

Generating Three Binary Addition Algorithms using Reinforcement Programming

Spencer White
27921 NE 152nd St.
Duvall, WA, 98019
Skwhite314@gmail.com

Tony Martinez
Computer Science
Department
Brigham Young University
Provo, UT 84602
martinez@cs.byu.edu

George Rudolph
Department of Mathematics
and Computer Science
The Citadel
Charleston, SC 29409
george.rudolph@citadel.edu

ABSTRACT

Reinforcement Programming (RP) is a new technique for automatically generating a computer program using reinforcement learning methods. This paper describes how RP learned to generate code for three binary addition problems: simulate a full adder circuit, increment a binary number, and add two binary numbers. Each problem is presented as an extension of the one previous to it, which provides an introduction to the practical application of RP. Each solution uses a dynamic, episodic form of delayed Q-Learning algorithm. "Dynamic" means that grows the policy during learning, and prunes it before the policy is translated to source code. This is different from Q-Learning models that use fixed-size tables or neural net function approximators to store q-values associated with (state,action) pairs. The states, actions, rewards, other parameters, and results of experiments are presented for each of the three problems.

Categories and Subject Descriptors

I.2.2 [Artificial Intelligence]: Automatic Programming;
I.2.6 [Artificial Intelligence]: Learning

General Terms

Algorithms, Theory

Keywords

Reinforcement learning, automatic program generation, binary addition

1. INTRODUCTION

Reinforcement Programming (RP) is a new technique for automatically generating a computer program using reinforcement learning methods. This paper describes how RP learned to generate code for three binary addition problems: simulate a full adder circuit, increment a binary number, and

add two binary numbers. Each problem is presented as an extension of the one previous to it, which provides a gradual introduction to the practical application of RP.

Each solution uses a dynamic, episodic form of delayed Q-Learning algorithm. This paper assumes that the reader is familiar with Reinforcement Learning methods [6, 5], the Q-Learning model [4] and episodic training. Each solution is described in terms of states, actions, rewards, and a few other parameters that are used to tune the model for learning. The learning algorithm is dynamic in that it grows the policy during learning, and prunes it before translating it to source code. The algorithm uses delayed learning in that only a few actions have a non-zero reward or penalty. The system is trained many times with the same training data in the same training episode, in order to allow the effects of rewards and penalties to propagate from a terminal state back to a start state.

These problems are interesting because they deal with constructing generalized mappings using a representative subset of known pairings. The Binary Incrementer learns a one-to-one mapping, and the General Binary Adder learns a many-to-one mapping. Experimenting with solving these problems using RP was also an effort to explore and demonstrate the applicability of the RP approach. More details about the ideas discussed in this paper are found in [7]. The experimental results and generated solutions are interesting because of what they tell us about the complexity of learning problems like these. Code and results are discussed in the applicable sections of the paper.

Along with the states, actions, rewards and other parameters, results of experiments with each of the three problems are presented. Section 2 discusses related work. Section 3 discusses RP concepts and definitions. Section 4 discusses the Full Adder. Section 5 discusses the Binary Incrementer. Section 6 discusses the General Binary Adder. Section 7 gives results showing that RP uses directed learning, as opposed to conducting a random search. Section 8 is the conclusion.

2. RELATED WORK

RP is most closely related to Genetic Programming (GP) [3] and Reinforcement Learning (RL) [5]. A common way of implementing a GP system involves using Genetic Algorithms (GA) to modify a population of trees, where each tree is a complete program. Similarly, RP uses RL techniques (in this case the Q-Learning model) to generate programs. RP avoids some of the challenges that programmers face when

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACMSE '10 April 15-17, 2010, Oxford, MS, USA.

Copyright 2010 ACM 978-1-4503-0064-3/10/04 ...\$10.00.

using GP, and overcomes some of the limitations faced when using the classic Q-Learning model.

Both RL and GAs attempt to find a solution to a problem through a combination of stochastic exploration and the exploitation of the properties of the problem. The difference between RL and GAs lies in how problems are formulated and in the techniques used to solve the problem. Similarly, RP and GP both use stochastic exploration combined with exploitation, but the techniques and program representations differ dramatically. The need to balance exploration and exploitation is something that RP shares with GP and other evolutionary algorithms.

Despite its successes, GP is not without challenges. For example, Kinnear describes the challenges presented by using GP to evolve a truly generalized sort [2]. 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 and very large trees. The programmer formulates these aspects of the problem, not the machine. Typically, many program trees must be evaluated during a given iteration.

In typical Reinforcement Learning (RL) [1, 5], 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, however, the learned policy is intended to be externalized and compiled as a program at some point, rather than simply continuously being executed by an agent. Hence, 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 classic Q-Learning algorithm has a number of characteristics that have to be dealt with in order to use it successfully in practical implementations. The most prominent of these is that table of (state, action) pairs becomes prohibitive if a lookup table is used. RP 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.

3. ON THE RP APPROACH

This section discusses the RP approach. This includes some Q-Learning model parameters, notions of state, policy translation, and policy execution.

With regard to the Q-Learning model used in this paper, the parameters, including rewards and penalties, were determined empirically. Holding the reward at 100 and varying the penalty and γ (a decay parameter), convergence occurred most rapidly with a penalty of -100 and a γ of 0.9 for all three problems. A *NOOP* action is provided as a starting “last state” for the initial state, when learning begins.

The RP notion of 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.

There are some basic rules about constructing the state, and determining what should be in the RP-specific portion of the state. A variable that directly references the input, or has an explicit purpose is called a *bound variable*. A *free variable* has no predefined purpose. 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. 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.

A number of different methods can be used to translate a learned policy into source code. The method we have used here 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, an 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.

4. FULL ADDER

4.1 Problem Description

The full adder is a circuit that has three inputs and two outputs. The inputs are two one-bit numbers and a carry in bit. It outputs the sum of those bits as a two-digit binary number, a one-bit sum and one-bit carry out. There are a number of ways to formulate the circuit and its associated logical representation. A formulation that uses only *AND*, *OR*, and *NOT* primitives is convenient for generating decision tree code.

$X1$, $X2$, and C_{in} are the inputs, and C_{out} and S are the outputs. The binary relational formulas are as follows:

$$\begin{aligned} C_{out} &= X1 * X2 + X1 * C_{in} + X2 * C_{in} \\ S &= (\overline{X1} * \overline{X2} * C_{in}) + (X1 * \overline{X2} * \overline{C_{in}}) \\ &\quad + (X1 * X2 * C_{in}) + (\overline{X1} * X2 * \overline{C_{in}}) \end{aligned} \quad (1)$$

4.2 Training Set and State Representation

A training instance for the full adder problem consists of

Table 1: Full Adder Inputs and Outputs

| C_{in} | X2 | X1 | C_{out} | S |
|----------|----|----|-----------|---|
| 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 2: Full Adder Action Set

| Action | Description |
|-------------------|------------------------------------|
| <i>NOOP</i> | Initial "last action". |
| <i>TERMINATE</i> | Ends policy execution |
| $C_{out} = True$ | Sets output C_{out} to true (1) |
| $C_{out} = False$ | Sets output C_{out} to false (0) |
| $S = True$ | Sets output S to true (1) |
| $S = False$ | Sets output S to false (0) |

three bits. The entire training set involved all eight possible instances the three bits could represent. The corresponding outputs are in Table 1.

Because the size of the state space is small, we can exhaustively represent all possible inputs, and the state representation is very simple. It consists of three boolean inputs, and two integer outputs. We use integer variables here rather than booleans because the output is tri-state, not binary. For each output, 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. Since the number of possible inputs is finite, there is no data-specific portion; the entire state is RP-specific.

4.3 Actions and Transitions

Five actions were made available to the system, and are described in Table 2. The goal states are any state in which the last action performed is the *TERMINATE* action. 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.

4.4 Results and Policy Translation

The system is trained on all training instances at each iteration. The most commonly learned policy contains 171 states. By executing the resulting policy on all the inputs and keeping only the visited states, the policy is pruned down to 24 states. The translation algorithm then translates the policy into the decision tree given as Listing 1. When the tree is executed as a program, decision tree is repeatedly executed until the *TERMINATE* action is performed. The code has been rewritten slightly to save space and help clarify its meaning for the reader. Nested if-then-else clauses have been replaced by a case statement in the code; nested if statements with single-variable clauses have been combined into the 16 statements shown in the listing. The reader can assume that each if statement is mutually

exclusive.

Listing 1: Full Adder Algorithm

```

1 switch lastact
2 case (NOOP)
3   if (!Cin && !X1 && !X2) {S=0;}
4   if (!Cin && !X1 && X2) {S=1;}
5   if (!Cin && X1 && !X2) {Cout=0;}
6   if (!Cin && X1 && X2) {S=0;}
7   if (Cin && !X1 && !X2) {S=1;}
8   if (Cin && !X1 && X2) {S=0;}
9   if (Cin && X1 && !X2) {S=0;}
10  if (Cin && X1 && X2) {S=1;}
11 case (Cout=1)
12   {TERMINATE;}
13 case (Cout=0)
14   if (!X1) {TERMINATE;}
15   if (X1) {S=1;}
16 case (S=1)
17   if (!X1) {Cout=0;}
18   if (X1 && !Cin) {TERMINATE;}
19   if (X1 && Cin) {Cout=1;}
20 case (S=0)
21   if (!Cin && !X1) {Cout=0;}
22   if (!Cin && X1) {Cout=1;}
23   if (Cin) {Cout=1;}

```

It is straightforward to show that this code implements the mapping from Table 1. Initially, the lastact is always *NOOP*, so lines 2-10 cover each one of the 8 possible combinations of 3 boolean inputs. As an example, Line 3 should be read as "If lastact is NOOP and all inputs are 0, then set S to 0 and change lastact to action (S=0)." Lines 4-10 read similarly, each assuming the lastact is NOOP. Lines 2, 11, 14, 17 and 20 are not assignment statements—they are meant to designate the last action.

Although there are 8 possible combinations of inputs, there are only 4 distinct combinations of inputs: all 0's (000), one 1's (001, 010, 100), two 1's (011, 101, 110) and all 1's (111). In the all 0's case, line 3 will execute, then line 21, then line 16. In the one 1's case, line 4, 5 or 7, is followed by line 15, then [it does not terminate]. In the two 1's case, line 6, 8 or 9 is followed by line 12 and then line 22. In the all 1's case, line 10 is followed by line 18, and then line 13. In all cases, S and C_{out} have the correct values when the algorithm terminates.

5. BINARY INCREMENTER

5.1 Problem Description

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 that the input number is all 1's, the result is all 0's. In effect this implements modular addition, but keeps the number of bits in the output the same as those in the input.

This problem has an interesting abstraction. The binary incrementer problem is an instance of the taking of a single input and mapping it to the corresponding output. The task of the RP system is to learn the one-to-one mapping between the inputs and the outputs. This application can be extended to several domains. One of note is the *successor problem*, the task of taking any input and generating its successor in an ordered list. The successor for each input

Table 3: Binary Incrementer Actions

| Action | Description |
|------------------|-----------------------|
| <i>NOOP</i> | Initial "last action" |
| <i>TERMINATE</i> | Ends policy execution |
| <i>INCI</i> | Increments i |
| <i>SETZERO</i> | Sets $i = 0$ |
| <i>INCJ</i> | Increments j |
| <i>SETJZERO</i> | Sets $j = 0$ |
| <i>SETTRUE</i> | Sets $dest[i]$ to 1 |
| <i>SETFALSE</i> | Sets $dest[i]$ to 0 |

may be known, but not the actual mapping. A system that can automatically generate that mapping is a useful tool.

5.2 Training Set and State Representation

The training instances consists of the entire set of 3-digit binary numbers. This was the smallest number of bits in the number that could still give the full range of states needed to make a general binary incrementer. Experiments with binary numbers with fewer than 3 digits led to non-general solutions. Numbers with more than 3 digits led to the same solution as that learned using 3-digit binary numbers.

The state consists of a data-specific and an RP-specific portion. The data-specific portion contains two integers: i and j . i is a bound variable, and j is a free variable. The data-specific portion also contains the training instance (*source*), as well as the destination binary number (*dest*) which is initialized to the training instance. Lastly, it has a variable *len*, storing how many bits are in the training instance. The RP-specific portion contains several boolean flags—whether or not $i = 0$, $j = 0$, $i = len$, and $j = len$ —the last action performed, and the values of the following bits: $source[i]$, $dest[i]$, $source[i - 1]$, $dest[i - 1]$. Both binary numbers were 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 $dest[i] = 0$. These constraints avoid potential array out-of-bounds errors in the implementation.

Experiments performed without including $source[i - 1]$ and $dest[i - 1]$ failed to lead to convergence. This suggests that the problem is a Partially Observable Markov Decision Process (POMDP) [1]—that is, the conditional probability of reaching future states depends on observing only part of the current state, not the full state or past states. The additional state information allows the system to learn using a model that satisfies the Markov Property, which is a requirement for using Q-learning.

5.3 Actions and Transitions

Table 3 contains the actions used for this problem. Goal states are any state with *TERMINATE* as the last action performed. 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.

The system provides a reward when the agent uses the *TERMINATE* action when the *dest* binary number has all its bits set to the correct value. If the agent uses the *TERMINATE* action in any other situation, a penalty is provided.

5.4 Results and Policy Translation

The system is trained on all training instances at each

iteration. The most commonly learned policy contains 254 states. The pruning algorithm prunes this policy to 9 states, and the translation algorithm translates it into the decision tree given in Listing 2. The generated code has been rewritten to accommodate the 2-column format. Line 3 should be read as "If i is not equal to length, and bit i of *source* is 0, and bit i of *dest* is zero, take action SETTRUE."

It is interesting to note two things: First, the variable j provided to the state was neither needed nor used. Second, although convergence would not occur without having $source[i - 1]$ and $dest[i - 1]$ as part of the state, the resulting decision tree uses neither value. Assume that the if statements are mutually exclusive—the else's are omitted for convenience. Indentation indicates nested ifs.

Listing 2: Binary Incrementer Algorithm

```

1 if (  $i \neq len$  )
2   if (  $source[i] == 0$  )
3     if (  $dest[i] == 0$  ) {SETTRUE;}
4     if (  $dest[i] == 1$  ) {TERMINATE;}
6   if (  $source[i] == 1$  )
7     if (  $dest[i] == 0$  ) {INCI;}
8     if (  $dest[i] == 1$  ) {SETFALSE;}
11  if (  $i == len$  ) {TERMINATE;}

```

The variable i starts at 0 and is incremented only when line 7 is executed. Recall also that *dest* is initialized to the same value as *source*. Every number falls into one of three possible cases: First, bit 0 is 0. Second, there is some value $k < len$ such that all the bits from position 0 to $k - 1$ are 1, and bit k is 0. Third, the number is all 1's. It is straightforward to show that the algorithm is correct for each of the three cases.

Consider the first case. Line 3 will cause $dest[0]$ to be set to 1. Next, because $dest[0]$ is 1, the policy will execute line 4, which causes the algorithm to terminate. This behavior increments the original value by 1, which is correct.

Consider the second case. Since $source[0]$ and $dest[0]$ is 1, line 8 will execute first. This will change $dest[0]$ to 0. Next, line 7 will execute, which changes the value of i to 1. Line 8 will execute on bit 1, which will change $dest[1]$ to 0, and then line 7 will execute, again incrementing i . Lines 8 and 7 will alternately execute until the first 0 is encountered in *source* and *dest* at position $i == k$. This will cause line 3 to execute, which will change $dest[k]$ to 1. Then line 4 will execute, which terminates the algorithm. This is the correct behavior.

Consider the third case. Lines 8 and 7 will alternate execution until there are no more bits $i == len$. All of the 1's have been inverted to 0's. At this point, line 11 will execute, and the algorithm will terminate. Again, this is the correct behavior.

What the learned program does is scan the binary number, starting at the least-significant bit. 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 any binary number.

6. GENERAL BINARY ADDER

6.1 Problem Description

A general binary adder takes in n -bit two binary numbers and returns an n -bit binary number representing their sum.

It is assumed that both inputs have the same number of bits. If not, the shorter number is padded with leading 0's to the proper length. In the even of an overflow, the overflow is ignored—in effect, the sum is mod 2^n . For example, $110 + 101 = 1011$, however the binary adder would return $110 + 101 = 011$. This was an implementation decision.

This problem has an interesting abstraction. This problem creates a many-to-one mapping (2^{2n} inputs mapping to 2^n outputs, where n is the number of bits). One useful application for a many-to-one mapping is a hash table. Typically, the hashing function is created beforehand, and modified until there is an even distribution across the mapping. Instead, the mapping can be created beforehand on a subset of the possible inputs. RP could then be used to create an appropriate hashing function. The general binary adder problem will be viewed from the perspective of learning a generalized function to map inputs to an output based on examples.

6.2 Training Set and State Representation

The training set consists of the cross product of the entire set of 4-digit binary numbers with itself (because two binary number inputs are needed). Size 4 is the smallest size that still visits every state necessary to learn a generalized binary adder. Smaller and larger sizes were experimented with, and size 4 fit the criteria. Size 3 did not lead to a generalized adder, while sizes 4 and up did.

The state representation consists of a data-specific portion and an RP-specific portion. The data-specific portion 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 , an array of integers the same length as the input numbers, called $dest$, and a variable called len , which is set to the length 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 contains four boolean flags, and integers $zcount$, $ocount$, $lastzcount$, $lastocount$, $dloc$ and $lastdloc$, and the last action performed. The boolean flags indicate whether $i = 0$, $i = len$, $j = 0$, and $j = len$. $zcount$ indicates how many zeros are indexed by $source1[i]$ and $source2[i]$ (either 0, 1, or 2 zeros). $ocount$ indicates how many ones are indexed by $source1[i]$ and $source2[i]$. $lastzcount$ and $lastocount$ indicate 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.

Experiments performed without $lastzcount$, $lastocount$, and $lastdloc$ failed to converge. Adding these variables allowed the system to converge. This suggests that the general binary adder problem is a Partially Observable Markov Decision Process, like the binary incrementer (see section 5.4). The elements of $source1$, $source2$, and $dest$ 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 out-of-bounds array errors in the implementation.

6.3 Actions and Transitions

The actions used by the general binary adder system are

the same as the actions used in the Binary Incrementer, in table 3. The goal states are any state where the last action performed is the *TERMINATE* action. A reward is given when the system terminates with $dest$ initialized and set to the correct answer. Otherwise, a penalty is given.

Smaller rewards and penalties are given for partial results that are correct, or incorrect, as an aid to convergence. The values are set empirically, by trial-and-error. If the values are too high, they will unfairly bias the system toward (or away from) certain (*state, action*) pairs. If they are too low, they don't bias learning enough to make a difference. If *SETTRUE* or *SETFALSE* sets $dest[i]$ to match $map[i]$, a small reward (as compared to the termination reward) is given. Otherwise, a small penalty (as compared to the termination penalty) is given. Note that the learning algorithm is not using an addition algorithm to learn an addition algorithm. The system is learning a mapping between given inputs and outputs, and is rewarded for finding a partial match.

There are restrictions placed on the actions that can be selected, depending on the state the agent is in. *SETTRUE* and *SETFALSE* cannot be performed if $i = len$ to avoid out-of-bound array errors. If $dloc = 1$ then *SETTRUE* cannot be performed. If $dloc = 0$ then *SETFALSE* cannot be performed. These restrictions are necessary to avoid the infinite reward the agent can get by repeatedly performing a reward-giving action.

6.4 Results and Policy Translation

The system is trained on all training instances for each iteration. The most commonly learned policy contains 913 states. After executing the policy on all the training instances, the policy is pruned to 48 states. A table showing the pruned policy can be found in [7]—it too large to include here. The policy is then translated into the decision tree presented as Listing 3. The generated code has been rewritten to accommodate the 2-column format. Line 3, where the first action is specified, should be read, "If $i < len$ and $dloc < 0$ and $zcount$ is 0 and $lastzcount$ is 0, then take action SETTRUE." It means "if not done, and $dest[i]$ is undefined, and both $source1[i]$ and $source2[i]$ are 1, and both $source1[i-1]$ and $source2[i-1]$ are 1, then set $dest[i]$ to 1." Other lines read similarly. If statements at the same indentation level are mutually exclusive.

Listing 3: Binary Adder Algorithm

```

1 if ((i != len)&&(dloc < 0))
2   if (zcount == 0)
3     if (lastzcount == 0) {SETTRUE;}
4     if (lastzcount == 1)
5       if (lastdloc == 0) {SETTRUE;}
6       if (lastdloc == 1) {SETFALSE;}
7     if (lastzcount == 2) {SETFALSE;}
8   if (zcount == 1)
9     if (lastzcount == 0) {SETFALSE;}
10    if (lastzcount == 1)
11      if (lastdloc == 0) {SETFALSE;}
12      if (lastdloc == 1) {SETTRUE;}
13    if (lastzcount == 2) {SETTRUE;}
14  if (zcount == 2)
15    if (lastzcount == 0) {SETTRUE;}
16    if (lastzcount == 1)
17      if (lastdloc == 0) {SETTRUE;}
18      if (lastdloc == 1) {SETFALSE;}
19    if (lastzcount == 2) {SETFALSE;}

```

```

20 if ((i!=len) && (dloc = 0)) {INCI;}
21 if ((i!=len) && (dloc = 1)) {INCI;}
22 if (i==len){TERMINATE;}

```

This algorithm does not use a carry bit. Instead, it uses bit counts (*zcount* and *lastzcount*), and the previously computed bit in the result (*lastdloc*) to determine how to set the current bit (*dloc*). Using bit counts reduces the number of unique combinations of bits in the inputs to nine, because *zcount* and *lastzcount* can only be 0, 1, or 2. When *lastzcount* is 1, it is necessary to look at *lastdloc* to determine whether there was a carry. Thus there are 12 cases, rather than nine, with associated actions in lines 2-19. Lines 20 and 21 increment *i* when *dloc* has been set to a value. The algorithm terminates only when all bits in the result have been set (line 22).

When the tree is executed as a program, decision tree is repeatedly executed until the *TERMINATE* action is performed. Initially, *i* is 0, *lastzcount* is 2, and *lastdloc* is -1. For brevity, we consider one of the 12 possible cases, to show that the algorithm behaves correctly. The other cases could be discussed similarly.

Consider the case where *source1*[*i*] and *source2*[*i*] are both 1, and one or the other of *source1*[*i* - 1] and *source2*[*i* - 1] are 0 (but not both), and *lastdloc* is 0. Line 5 is executed. Because *lastdloc* is 0, but *lastzcount* is 1, there must have been (in effect) a carry from the addition of bits at *i* - 2. *dloc* is therefore set to 1. Next, line 21 is executed, incrementing the *i* by 1, and *dloc* now references the next bit in *dest*. *lastzcount* now changes to 0. We know that line 3, line 9 or line 15 must execute next, depending on the new *zcount*. If line 3, *dloc* should be set to 1, passing along the previous carry. If line 9, *dloc* should be set to 0, passing along a carry. If line 15, *dloc* is set to 1, absorbing the previous carry. Next, line 20 is executed, incrementing the value of *i*.

Assume that the algorithm has the correct behavior for every possible combination of *source1*[*i* - 1], *source2*[*i* - 1], *source1*[*i*], *source2*[*i*], and *dest*[*i* - 1] (this requires enumerating all 12 of the cases discussed above), for some *i*. It is straightforward to verify that the algorithm works correctly for all 1- and 2-bit numbers. Now assume an *n*-bit number, $n > 2$, and choose some $k < n$. Assume that the algorithm behaves correctly for the first $k - 1$ bits. Let $k = i$. The algorithm above will generate the correct values for bits k and $k + 1$. Therefore, by induction, this algorithm would generate the correct result for any *n*-bit number.

7. IS RP REALLY LEARNING?

Is RP just randomly creating programs until a successful program is found, or performing a directed search? Random Program Generation (RPG) was used to randomly create a computer program using the same state representation used by RP, for each of the three applications presented in this paper. The entire state space explored by a given RP application is iterated over, and each state is assigned a random action. This process creates an entire policy 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 generated for each application. No random policy performed correctly on any entire training set. Each of three RP algorithms described in this

paper were generated in 10,000 iterations or less. Thus, it is clear that RP performed much better than RPG for these applications.

8. CONCLUSIONS

This paper presented Reinforcement Programming (RP) as a new technique for automatically generating programs using reinforcement learning. We used RP to generate iterative algorithms for three binary addition problems: Simulate a Full Adder Circuit, a Binary Incrementer, and a Binary Adder.

Results showed that RP generates a correct, efficient algorithm, and generates it quickly. Each algorithm was generated in 10,000 iterations or less. RP uses a lookup table that grows as needed, rather than a fixed-size lookup table, or a neural network function approximator. The number of actual states explored is much less than the total possible number of states, so this seems like a good choice, as it overcomes some of the weaknesses of the standard Q-Learning model. Results also show that RP is performing a directed search, not just randomly creating programs until a successful program is found.

One of the goals in using these problems is to explore and demonstrate the applicability of the RP approach. The results given in this paper algorithms extend initial work on RPSort, which uses RP to generate in-place, iterative sorting algorithms (currently under review for publication). Future possible work includes simulating a Turing-complete language and using RP to find a Turing machine that sorts any list; exploring real-world applications beyond canonical benchmark applications; working to formalize the state representation process; contribute toward an effort to look at the higher-level question of how to judge a particular algorithm's or method's fitness for a particular task. Further implementation experiments may 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.

9. REFERENCES

- [1] T. Jaakkola, S. P. Singh, and M. I. Jordan. 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, pages 345-352. The MIT Press, 1995.
- [2] K. E. Kinneer. Evolving a Sort: Lessons in Genetic Programming. In Proceedings of the 1993 International Conference on Neural Networks, volume 2, pages 881-888. IEEE Press, 1993.
- [3] J. R. Koza. Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers, 2003.
- [4] T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.
- [5] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
- [6] C. J. Watkins. *Learning from delayed rewards*. PhD thesis, Cambridge university, 1989.
- [7] S. K. White. Reinforcement programming: A new technique in automatic algorithm development. Master's thesis, Brigham Young University, 2006.