Reinforcement Programming: A New Technique in Automatic Algorithm Development

Spencer Kesson White

Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Computer Sciences Commons

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/480
REINFORCEMENT PROGRAMMING: A NEW TECHNIQUE IN AUTOMATIC ALGORITHM DEVELOPMENT

by

Spencer K. White

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science
Brigham Young University
July 2006
This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date

Tony Martinez, Chair

Date

Dan Ventura

Date

Bill Barrett
As chair of the candidate’s graduate committee, I have read the thesis of Spencer K. White in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Tony Martinez  
Chair, Graduate Committee

Accepted for the Department

Parris Egbert
Graduate Coordinator

Accepted for the College

Earl M. Woolley
Dean, College of Physical and Mathematical Sciences
ABSTRACT

REINFORCEMENT PROGRAMMING: A NEW TECHNIQUE IN AUTOMATIC ALGORITHM DEVELOPMENT

Spencer K. White
Department of Computer Science
Master of Science

Reinforcement programming is a new technique for using computers to automatically create algorithms. By using the principles of reinforcement learning and Q-learning, reinforcement programming learns programs based on example inputs and outputs. State representations and actions are provided. A transition function and rewards are defined. The system is trained until the system converges on a policy that can be directly implemented as a computer program. The efficiency of reinforcement programming is demonstrated by comparing a generalized in-place iterative sort learned through genetic programming to a sorting algorithm of the same type created using reinforcement programming. The sort learned by reinforcement programming is a novel algorithm. Reinforcement programming is more efficient and provides a more effective solution than genetic programming in the cases attempted. As additional examples, reinforcement programming is used to learn three binary addition problems.
ACKNOWLEDGMENTS

First in my thanks and my life, my wife Keri-Ann. Without her, I doubt I would have had the confidence to complete my work. Her support and belief in my abilities has always been a bright light, even when debugging code at one in the morning. My children, Joseph and Sarah, have also been a constant inspiration to me. Their bright eyes and delightful laughter always kept me going. Next, I thank my parents. Their prayers and support have been answered and appreciated. I also thank my advisor, Tony Martinez, for helping me round out the edges. Further thanks go to the members of the Neural Networks and Machine Learning lab. Their questions and suggestions were a tremendous help. Lastly and most importantly, my thanks go to my Heavenly Father for teaching, inspiring, and guiding me to this point.
Acknowledgments vi

List of Tables ix

List of Figures x

1 Introduction 1

2 Genetic Programming and Sorting Algorithms 3
  2.1 Genetic Programming ................................. 3
  2.2 Evolving a General Sort ............................. 4

3 Reinforcement Programming 7
  3.1 Reinforcement Learning vs. Reinforcement Programming .... 7
  3.2 State Representation .................................. 8
  3.3 Action Selection ...................................... 10
  3.4 State Transition and the Role of Rewards ................ 10
  3.5 Policy Translation .................................... 11
  3.6 Reinforcement Programming vs. Genetic Programming .... 12

4 RPSort 13
  4.1 Q-Learning Specifics ............................... 13
  4.2 State Representation ................................ 14
  4.3 Actions and Transition ............................... 15
  4.4 Examples of RP Concepts Illustrated by RPSort .......... 16
  4.5 Policy to Program Conversion ........................ 16
  4.6 Training Data ..................................... 17
  4.7 Example of RPSort Training ........................... 17

5 Results 23
  5.1 The Learned Sorting Algorithm: RPSort .................. 23
  5.2 Comparison between Reinforcement Programming and Genetic Programming for Generating a Sort ........................................ 28
6  Conclusions and Future Work

A  Decision Tree Construction Example

B  Reinforcement Programming: Binary Addition

   B.1  RP: Full Adder .................................................. 33
       B.1.1  Training Set and State Representation ................. 35
       B.1.2  Actions and Transition .................................. 35
       B.1.3  Results and Policy Translation .......................... 36
   B.2  RP: Binary Incrementer ....................................... 38
       B.2.1  Training Set and State Representation ................. 38
       B.2.2  Actions and Transition .................................. 38
       B.2.3  Results and Policy Translation .......................... 39
   B.3  RP: General Binary Adder ..................................... 41
       B.3.1  Training Set and State Representation ................. 41
       B.3.2  Actions and Transition .................................. 42
       B.3.3  Results and Policy Translation .......................... 43
   B.4  Comparison to Random Program Generation .................. 43

Bibliography 48
# List of Tables

2.1 Functions and terminals used by Kinnear to evolve an iterative sort .......................... 6
4.1 Actions used in RPSort ......................................................................................... 15
5.1 RPSort Policy ........................................................................................................ 24
A.1 A Trivial Policy .................................................................................................... 30
B.1 Full Adder Inputs and Outputs ........................................................................... 35
B.2 Full Adder Action Set .......................................................................................... 35
B.3 Full Adder Policy ................................................................................................. 36
B.4 Binary Incrementer Actions ................................................................................ 39
B.5 Binary Incrementer Policy ................................................................................... 40
B.6 Binary Adder Actions ........................................................................................... 42
B.7 Binary Adder Policy ............................................................................................. 44
B.8 Summary of Binary Addition Possibilities ......................................................... 46
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A simple genetic program tree</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>Translated LISP code</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>First False decision tree</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>The basic RP program structure</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>State representation for RPSort</td>
<td>18</td>
</tr>
<tr>
<td>4.2</td>
<td>State transition example</td>
<td>19</td>
</tr>
<tr>
<td>4.3</td>
<td>Training Instance Modification Example</td>
<td>19</td>
</tr>
<tr>
<td>4.4</td>
<td>Non-determinism example</td>
<td>19</td>
</tr>
<tr>
<td>4.5</td>
<td>Initial state</td>
<td>19</td>
</tr>
<tr>
<td>4.6</td>
<td>First state transition</td>
<td>20</td>
</tr>
<tr>
<td>4.7</td>
<td>Second state transition</td>
<td>20</td>
</tr>
<tr>
<td>4.8</td>
<td>Third state transition</td>
<td>21</td>
</tr>
<tr>
<td>4.9</td>
<td>New state transition from the initial state</td>
<td>21</td>
</tr>
<tr>
<td>4.10</td>
<td>State transition leading back to original set</td>
<td>22</td>
</tr>
<tr>
<td>4.11</td>
<td>Penalized terminate example</td>
<td>22</td>
</tr>
<tr>
<td>4.12</td>
<td>Rewarded terminate example</td>
<td>22</td>
</tr>
<tr>
<td>5.1</td>
<td>RPSort</td>
<td>25</td>
</tr>
<tr>
<td>5.2</td>
<td>State transitions for sorting a list</td>
<td>26</td>
</tr>
<tr>
<td>5.3</td>
<td>Average case comparisons</td>
<td>27</td>
</tr>
<tr>
<td>A.1</td>
<td>Splitting on last action</td>
<td>31</td>
</tr>
<tr>
<td>A.2</td>
<td>Splitting on last action then ( i )</td>
<td>32</td>
</tr>
<tr>
<td>A.3</td>
<td>Splitting on ( i )</td>
<td>32</td>
</tr>
<tr>
<td>B.1</td>
<td>Circuit Diagrams for a Full Adder</td>
<td>34</td>
</tr>
<tr>
<td>B.2</td>
<td>Full Adder Decision Tree</td>
<td>37</td>
</tr>
<tr>
<td>B.3</td>
<td>Binary Incrementer Decision Tree</td>
<td>40</td>
</tr>
<tr>
<td>B.4</td>
<td>Binary Adder Decision Tree</td>
<td>45</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Generally speaking, at any given moment in execution a computer program is a collection of data forming a state. The program also specifies state transition - how the computer should modify the current state to transition to a different one [1, 2]. Through the simple concept of states and transition rules, computer programs can be created. The abstract notion of states and transitions is very common. People or computers examine a state, and alter it based on the changes they desire. Computers increment a variable. Humans put the bread dough in the oven. An agent changes the state it is in by transitioning to a different state.

Typically, in Reinforcement Learning [3, 4, 5, 6, 7] (RL), 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 rule stipulating what action to take in each state to maximize the agent’s reward.

Clearly, there are similarities between a basic computer program and the general form of reinforcement learning. The term Reinforcement Programming (RP), introduced by this thesis, refers to a new concept that exploits these similarities. RP works to automatically generate an algorithm to solve a task by creating a policy that is executed as a computer program.

Genetic Algorithms (GA’s) model Darwinian evolution to optimize a solution to a given problem [8, 9, 10, 11]. Genetic programming (GP) uses GA techniques to automatically create programs that solve problems. There are similarities between RP and GP, just as there are similarities between RL and GA’s. Both RL and GA’s are attempting to find an optimized solution through a combination of stochastic exploration and the exploitation of the properties of the problem. The difference between RL and GA’s 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.

The purpose of this thesis is to introduce RP and its basic concepts. In RP, a state representation is provided to the system. Atomic actions are also provided and used to define how to transition from state to state. One or more goal states are defined. Rewards for each transition are provided. 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 this program can generalize to new inputs.

RP is compared to GP on the task of learning an iterative in-place sorting algorithm. Experiments performed using RP to learn a sorting algorithm that yields efficient sorting algorithm in fewer iterations than GP. This sorting algorithm is novel, and is more efficient than the sort learned by GP. The learned program is simple to understand. Unlike GP, the program does not grow more complex with each iteration. Instead, the learned program only grows complex enough to solve the problem efficiently. RP learns to make unnecessarily complex programs more efficient. In the comparisons performed, RP is more efficient than GP at finding a program, and finds more effective solutions.

This thesis assumes that the reader is already familiar with RL and Q-learning [12], and is also familiar with GA’s. In Chapter 2, GP is briefly described, along with a technique for using GP to learn a generalized in-place iterative sorting algorithm. A description of RP theory is presented in Chapter 3. In Chapter 4, a technique and the related issues for using RP to create a sorting algorithm are presented. This chapter also provides detailed diagrams and examples of the principles discussed and an example of a few training steps used to learn the sorting algorithm. In Chapter 5 results from the described sort experiments are discussed, explained, and compared to those from the discussed GP sort experiments. Future directions for RP are discussed in Chapter 6. As additional examples, RP is used to learn programs that solved three different binary addition problems. These applications are presented in Appendix B.
Chapter 2

Genetic Programming and Sorting Algorithms

This chapter overviews the basic concepts behind genetic programming (GP) and a technique for using it to a sorting algorithm. The benefits of GP are described, as well as its drawbacks. An example of a GP program tree is provided. The results of evolving a sort using GP are presented.

2.1 Genetic Programming

Overall, genetic programming has proven to be a flexible method for developing algorithms to solve tasks. Examples include evolving sorting networks [13, 14] and developing quantum algorithms [15, 16].

A common way of implementing a GP system involves using GA techniques to modify a population of trees [17, 18, 19]. 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 non-function). The entire tree represents a program. Figure 2.1 demonstrates 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.

As with GA’s, 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. Non-procedural programming languages,

![Figure 2.1: A simple genetic program tree](image)
such as LISP, are ideal for GP implementations, being able to naturally and easily express and execute the tree-like structure of the program.

One of the aspects of GP that makes it so powerful is that the number of possible functions that can be used by GP is vast. This means that the program trees that can be generated by GP is not limited by the usable functions. Instead, the number of possible trees is deliberately limited by selecting a subset of all usable functions. Another powerful aspect of GP is the ability for the tree structure to represent arbitrarily complex programs. Thus, the ability for GP to develop a suitable program is not terminally hampered by the complexity of the problem, only by the collection of functions made available to the GP system.

This arbitrary complexity is also a problem for GP. In standard GA’s the genome representation is typically set, with no change to its length or complexity. In GP, program trees can grow to arbitrary lengths. As a result, a common problem is the rather rapid growth in tree size among the program population. Consequently, methods must be devised and implemented, specific to the task being applied, to reduce program size without preventing the system from finding a program complex enough to solve the problem.

2.2 Evolving a General Sort

These advantages and problems are present when using GP to evolve a generalized sorting algorithm. Evolving a generalized sort is surprisingly 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.

In [20, 21], Kinnear describes the challenges presented by using GP to evolve a sort, as well as the techniques he used. In this chapter, these techniques are quickly reviewed as a means to compare evolved sorts with those derived via RP, as well as to provide an easy reference when discussing RP.

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 experimented with. Kinnear’s functions and terminals are described in Table 2.1. Not all of the functions were used concurrently; different subsets of the functions listed were used in different experiments, with different results. The simplest sort evolved is provided in Figure 2.2.

Obviously this simple program will sort any list of any size, although the efficiency is poor since this program is just a bubble sort. 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. This relationship implies that bubble sort, a very simple program with very little complexity in the corresponding GP tree, is the most fit of all the sorting algorithms representable by the functions and terminals chosen by Kinnear.

In an effort to decrease the size of the resulting programs, Kinnear includes a size penalty in the 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.

GP is somewhat effective for creating a general sorting algorithm. It does have several challenges, however. First of all, Kinnear’s experiments failed to always produce a general sorting algorithm without severe complexity penalties. Second, many different programs had to be evaluated at each iteration. Third, the program trees had a tendency to get large very rapidly, hurting generalization and increasing complexity. Fourth, the solution considered the most fit was a rather inefficient sorting algorithm. In the next chapter, these issues are discussed and overcome through the introduction of RP.
<table>
<thead>
<tr>
<th>Terminals</th>
<th>*len*</th>
<th>The length of the list being sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>index</td>
<td>An iterator variable taking on the value of the current index when in an iterator function, or zero otherwise</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arithmetic Functions</th>
<th>(n-m)</th>
<th>returns the value ((n - m))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(n+1)</td>
<td>returns the value ((n + 1))</td>
</tr>
<tr>
<td></td>
<td>(n-1)</td>
<td>returns the value ((n - 1))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence Comparison and Manipulation Functions</th>
<th>order(x, y)</th>
<th>puts the elements at indices (x) and (y) in the correct order</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>swap(x, y)</td>
<td>swaps the elements at indices (x) and (y)</td>
</tr>
<tr>
<td></td>
<td>wismaller(x, y)</td>
<td>returns the index of the smaller element at indices (x) and (y)</td>
</tr>
<tr>
<td></td>
<td>wibigger(x, y)</td>
<td>returns the index of the larger element at indices (x) and (y)</td>
</tr>
<tr>
<td></td>
<td>if-(&lt;)x, y, function)</td>
<td>performs (function) if the element at (x) is less than the element at (y)</td>
</tr>
<tr>
<td></td>
<td>less(x, y)</td>
<td>returns 1 if the element at (x) is less than the element at (y), 0 otherwise</td>
</tr>
<tr>
<td></td>
<td>if (test) function</td>
<td>performs (function) if (test) is non-zero</td>
</tr>
</tbody>
</table>

| Iterative Function | dobl(start, end, function) | performs a for-loop from start to end, executing \(function\) |

Table 2.1: Functions and terminals used by Kinnear to evolve an iterative sort
Chapter 3

Reinforcement Programming

This chapter reviews the general concepts of RP. First, it discusses the differences between RP and RL. This chapter then describes the general state representation, the role of actions, and the affect of state transitions. Lastly, it describes how the policy is translated into a program.

Throughout this chapter, a toy problem is used to illustrate the concepts discussed. Consider the task of taking an array of booleans of arbitrary length and finding the index of the first False value in the array, called the First False problem. The array is 0-based, meaning the first element in the array is referenced by array[0].

3.1 Reinforcement Learning vs. Reinforcement Programming

Just as GP is similar to GA’s, RP is similar to RL. States and state transitions are provided, and the system learns a policy that provides the action to perform in each state that maximizes reward (or minimizes cost).

There are some differences between RP and RL, however. First, RL does not have training instances, whereas RP does. This training instance in RP is a sample of the data that will be used by the algorithm the system is developing. This training instance is made a part of the state. A second difference is that in the most basic form of RL, the agent has full knowledge of its current state. Any non-determinism usually comes not from the state itself, but from the actions that can be performed. RP generally has completely deterministic actions. It is the state itself that provides any non-determinism, even in the simplest form of RP. This is particularly true when there are a variety of possible inputs. Fortunately, this can be resolved in the same manner as non-determinism in RL. An empirical measure of the likelihood of transitioning from one state to another via a particular action, along with an optional prior belief, can provide the expected value of an action performed in a given state. By tracking state transitions throughout the training process, this expected value will approach the correct value of the action leading to those state transitions. A third difference between RL and RP is that, in RL, all the possible states are typically already known to the agent. In RP, the collection of states begins with all the states the agent can start in. As actions are performed, new states are generated and added to the collection of states, and the transitions are tracked to deal with any non-determinism.
Once the RP policy is learned, it is executed in the same fashion as the RL policy. Until the agent is in the goal state, the state is referenced in the policy, and the corresponding action is executed. This action changes the state the agent is in. The RP policy can be simplified by converting it to a collection of rules based on the properties of the state, such as by creating a decision tree. These automatic methods can make the program structure easier to understand and faster to execute.

### 3.2 State Representation

The state is divided into two parts. The first is the *data-specific portion* of the state. The second is the *RP-specific portion*. 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. The RP-specific portion contains relationships between the data and variables that allow generalizability. In the case of the First False problem, the data-specific portion would contain the array of booleans, the length of the array (*len*), and a variable *i*. The RP-specific portion would contain the value of *array[i]*, whether or not *i* = 0 (the bottom index of the array) and whether or not *i* = *len* (the end of the array).

The conditionals and relationships contained in the RP-specific portion of the state become the conditionals that handle program control. In the First False problem, suppose the system learns to increment *i* until *array[i]* = *false* or *i* = *len*. The conditionals in the RP-specific portion become the comparisons that handle program control.

An important property of the state is *state equivalency*. Two states are considered equivalent when 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 does not allow for generalizability. In the case of the First False problem, consider two training instances *true, true, false* and *true, false*. The data-specific portion for state A has the first training instance, *i* = 1, and *len* = 3. It’s RP-specific portion has *i* \(\neq 0\), *i* \(\neq len\), and *array[i]* = *true*. The data-specific portion for state B has the second training instance, *i* = 1, and *len* = 2. It’s RP-specific portion has *i* \(\neq 0\), *i* \(\neq len\), and *array[i]* = *true*. Although the data-specific portions of the two states are not the same, the RP-specific portions are. Thus, state A and state B are equivalent.

There are some general rules for determining what the RP-specific portion of the state should be. If there are a small number of possible inputs, then the RP-specific and data-specific portions become the same thing. If the First False problem only considered arrays of booleans up to size 3, then it becomes easy to exhaustively list all possible inputs (14 different inputs total). Thus, there is no need for generalization. The RP-specific portion would contain the value of each element
in the array, as well as the size of the array. When there are not a finite number of inputs, or when
the number of inputs is unreasonably large, generalization is required. There are some basic rules
for constructing the state. First, though, some definitions are required.

A variable that directly references the input (such as by indexing an element in a list) is
called a bound variable. $i$ in the First False problem is a bound variable. A variable that does not is
called a free variable. A free variable has no pre-set purpose, while bound variables have an explicit
purpose. If the example in this chapter contained a variable $j$ that did not reference the values in
the array at all, it would be a free variable. Bound variables should have booleans in the RP-specific
portion of the state that indicate if the variable is 0, the size of the input, or neither. Furthermore,
booleans should exist between each possible pairing of bound variables indicating which of the two
in the pair is the larger. If the First False problem contained a second bound variable, $k$, the RP-
specific portion would require booleans indicating whether or not $k = 0$, $k = \text{len}$, $i < k$ and $i > k$.
The element of the input that the bound variable references should also be in the RP-specific portion
of the state ($\text{array}[i]$ in the case of the First False problem). Each free variable has two booleans
in the RP-specific portion. The first is whether or not that free variable equals 0. The second is
whether or not it equals the size of the input. If the First False problem had the free variable $j$,
the RP-specific portion of the state would contain booleans indicating whether or not $j = 0$ and
$j = \text{len}$. 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 example, consider the Second False
problem (analogous to the First False problem). It has all the same state aspects as the First False
problem, but it also contains the free variable $m$. The system may learn to increment $m$ once at
each $\text{False}$ value encountered by $\text{array}[i]$. Thus, the system could learn to terminate at the first
$\text{array}[i] = \text{False}$ it encounters when already $j \neq 0$. This would be the second $\text{False}$ in the array.
As another example, in the sorting application described in Chapter 4, some of the solutions used
the free variable as a way of determining if there had been any swaps since the last time both bound
variables had equalled 0. The question of how many bound variables to use is an open question at
this time. The problem of stack equivalency mentioned in Chapter 6 answers this problem. The
RP-specific portion also typically contains the last action performed. This is to give some context
to the current state the system is in. Therefore, the RP-specific portion of the state for the First
False problem also contains the last action performed.

When it comes time to convert the 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
right sequence of functions to accomplish the goal. It is important to note that not all states will
necessarily be visited by the execution of the program. This allows for some pruning and program
simplification. This is discussed in Chapter 3.5.
3.3 Action Selection

As the above section implies, there should be very few, if any, conditionals associated with the actions. As many conditionals as possible should be provided by the state and not by the actions. The primary reason for this is to prevent forcing the system to learn a program preconceived by the one designing the system; this would defeat the whole purpose of RP. Of course, if this is not an issue (such as when no truly efficient algorithm is known to exist), then there is some leeway between how many conditionals should be provided by the state and how many should be provided by the actions. As a general rule of thumb, a conditional should only be included in an action if it results in a dramatic reduction in the number of possible states.

The fewer actions included in the system, the better. Obviously, the complexity of the state transitions is dependent on the number of actions. Furthermore, since 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.

Commonly, the list of actions available to the system includes the \textit{TERMINATE} function, which simply tells the system to stop traveling through the world (stop executing the program). This constitutes arriving at the goal state. This allows the system to learn to terminate at the best possible way.

In the case of the First False problem, there are two actions available to the system. \textit{INCI} increments \(i\) if \(i \neq \text{len}\). \textit{TERMINATE} indicates the end of the program (arrival at a goal state).

3.4 State Transition and the Role of Rewards

A state transition function defines how an action changes the 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 non-deterministic nature of the transitions is constructed. With the First False problem, consider a training instance of \([True, True, True, False]\). Let \(i = 1\), and the last action performed be \textit{INCI} (since the only other available action terminates the program). By incrementing \(i\) again, the RP-specific portion of the state does not change, and so the system considers the action \textit{INCI} to have transitioned to the same state. However, by incrementing \(i\) again, the value referenced by \texttt{array}[i] becomes \textit{False}. Thus, a change in the RP-specific portion of the state occurs, and the system considers it a different state.

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. For example, if the First False system performs \textit{TERMINATE} when \(i\) indexes the first \textit{False} value in the array,
a reward is provided. Otherwise, a penalty may be provided when TERMINATE is performed to make the system adverse to performing that action in the wrong state. Small awards 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. This does not require any knowledge of what the solution to the task ought to be. It merely requires knowledge of what the output is supposed to be for that training instance. Since every training instance requires a corresponding output this information is readily available. If the First False system performs INCI while array[i] = False, a small penalty may be provided. Notice that this does not specify which False value the penalty is incurred. It happens anytime the system increments i past a False value.

3.5 Policy Translation

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 table. It is important to remember that it is the RP-specific portion of the state that is used in the policy.

As stated before, RP (just like RL) simulates an agent traveling through a world. The agent travels until it reaches some goal state. Thus, the program produced by RP runs until a goal state is reached. At each step of the program, the policy is checked to determine the next action to be performed. Thus, the program is a while loop, with the body of the while loop being the policy. The conditional for the while loop is the check to see if the goal state has been reached. In the case of the First False problem, the while loop conditional would be whether or not the last action performed was TERMINATE.

There are several ways to represent the body of the while loop. The simplest method is to simply use the policy and the transition function. A more complicated, but more efficient way 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 transition function is still 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 can be used to construct the decision tree. The purpose behind the decision tree is primarily to speed up determining which action to perform next. Other rule-learning methods can also be implemented to transform large policy tables into smaller, easier-to-understand program instructions. In the case of the First False problem, consider that the policy consisted of performing INCI for each state where array[i] ≠ FALSE and the last action
if $i \neq \text{len}$
  if $\text{array}[i] \neq \text{False}$
    \text{INCI}
  else \text{TERMINATE}
else \text{TERMINATE}

Figure 3.1: First False decision tree

while $\text{currentstate} \neq \text{goalstate}$
  pass $\text{state}$ into decision tree to retrieve action
  perform action

Figure 3.2: The basic RP program structure

performed is not \text{TERMINATE}. In the other states, \text{TERMINATE} is performed. This results in the decision tree in Figure 3.1.

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 repeats until a goal state is reached. Figure 3.2 demonstrates this in pseudocode.

3.6 Reinforcement Programming vs. Genetic Programming

An advantage of RP over genetic programming is the style of the programs produced. Genetic programming constructs tree-like programs, presenting a structure somewhat more familiar to programmers. Reinforcement programming 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 genetic programming 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, non-generalizable programs is automatically provided for by RP.

A disadvantage to RP is that there is a lot of initial exploration involved. Due to the way information propagates backwards from the goal state to the starting state, it can take a lot of random exploring before there begins to be a well-defined policy 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.
Chapter 4

RPSort

In this chapter, an application for RP is described. Specific examples of the principles described in Chapter 3 are provided.

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 in Chapter 3, the actual resulting programs are dependent on the actions and state representation chosen for the system.

4.1 Q-Learning Specifics

Non-deterministic Q-learning is used for RPSort. The exact formula for the optimal policy learned via Q-learning is:

\[ \pi^*(s) = \arg\max_a Q(s, a) \]  

(4.1)

where the terms are defined as follows:

- \( s \) a state
- \( a \) an action
- \( Q(s, a) \) the expected Q-value for action \( a \) in state \( s \)
- \( \pi^*(s) \) the optimal policy for state \( s \) given current Q-values

\( Q(s, a) \) is calculated as follows:

\[ Q_n(s, a) = (1 - \alpha)Q_{n-1}(s, a) + \alpha(\sum_i p(r_i | s, a)r_i) + (\gamma \sum_j p(s_j | s, a) \arg\max_{a'} Q_{n-1}(s_j, a')) \]  

(4.2)
where:

\( p(r_i|s,a) \) the probability of receiving reward \( r_i \) when performing action \( a \) in state \( s \) (learned empirically)

\( p(s_j|s,a) \) the probability of transitioning to state \( s_j \) when performing action \( a \) in state \( s \) (learned empirically)

\( \gamma \) the discount factor, \( 0 < \gamma < 1 \)

\( r_i \) the \( i^{th} \) reward given for performing action \( a \) in state \( s \)

\( \alpha \) a decaying weight calculated as \( \frac{1}{1 + \text{visits}_n(s,a) \sigma} \)

\( \text{visits}_n(s,a) \) The number of times the state-action pair \( (s,a) \) has been visited

\( \sigma \) The rate of decay

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. This exploration/exploitation decision was made to allow for a lot of exploration, but still permit enough exploitation as to truly evaluate the policy.

Values for \( \sigma, \gamma, \) the reward, and the penalty were experimented with to see how robust the system was. Experiments indicate that the actual values of the reward and penalty do not matter as much as the ratio between them (i.e. \( \frac{\text{reward}}{\text{penalty}} \) gave exactly the same results as \( \frac{10 \times \text{reward}}{10 \times \text{penalty}} \)). Further experiments held the reward constant at 100, and varied \( \gamma \) and the penalty. A \( \gamma \) of 0.6 to 0.9 with a penalty of \(-160\) to \(-180\) leads to convergence the most often (between 90 and 95% of the time convergence occurred within 200,000 iterations). Experiments with \( \sigma \) indicated that smaller values lead to convergence faster than larger values. This is likely due to the fact that the system is not only exploring the state space, but empirically learning the probabilities of the rewards and transitions.

4.2 State Representation

In this particular application of RP, a training instance is a list of integers. The list is modified when some actions are performed. Thus, the data contained in the list has an affect on the state transition. The state itself has both the data-specific portion and the RP-specific portion. The data-specific portion of the state contains three variables, \( i, j, \) and \( k \). \( i \) and \( j \) are both bound variables. \( k \) is a free variable. The data-specific portion also contains the length of the current list being sorted (\( \text{len} \)), and the list itself (\( \text{list} \)). The RP-specific portion of the state contains primarily booleans. It contains whether or not \( i = 0, j = 0, k = 0, i < j, i > j, i = \text{len}, j = \text{len}, k = \text{len}, \) and \( \text{list}[i] > \text{list}[j] \). It also contains the last action performed. The list is indexed as a 0-based array, (0 to \( \text{len} - 1 \)). If \( i = \text{len} \) or \( j = \text{len} \), then as a default relationship \( \text{list}[i] < = \text{list}[j] \) to avoid
Table 4.1: Actions used in RPSort

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOP</td>
<td>A “non-action”. Not selectable for execution.</td>
</tr>
<tr>
<td>TERMINATE</td>
<td>Ends policy execution</td>
</tr>
<tr>
<td>INCI</td>
<td>Increments i</td>
</tr>
<tr>
<td>INCJ</td>
<td>Increments j</td>
</tr>
<tr>
<td>INCK</td>
<td>Increments k</td>
</tr>
<tr>
<td>SETIZERO</td>
<td>Sets $i = 0$</td>
</tr>
<tr>
<td>SETJZERO</td>
<td>Sets $j = 0$</td>
</tr>
<tr>
<td>SETKZERO</td>
<td>Sets $k = 0$</td>
</tr>
<tr>
<td>SWAP</td>
<td>Swaps the values in $list[i]$ and $list[j]$</td>
</tr>
</tbody>
</table>

out-of-bounds array errors. The data-specific portion contains all the information needed to actually allow a list to be sorted. The RP-specific portion is general enough that lists longer than those the system is trained on can be passed in as data and still be handled by the state representation.

4.3 Actions and Transition

The actions were chosen to be as atomic as possible. Table 4.1 lists the actions that are available to the RP system. 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.

The goal states are any states in which the last action performed is the TERMINATE action. There is a reward for terminating when the list is sorted, and a penalty for terminating when the list is unsorted. The list is checked for whether or not it is sorted once the TERMINATE action is chosen. It is important to note that determining whether the list is sorted or not is used only at the end to determine if the objective has been reached (similar to the way in which Kinnear checked to see if the list that passed through a particular population member was sorted). It is not used in the actual sorting algorithm in any way. In fact, a sorting algorithm is not even required to determine if the list is sorted. All that is required when evaluating the list is to determine whether or not all the elements are in ascending order.

There is one other action that has a reward and penalty associated with it. SWAP gives a reward of 10 if $i < len$, $j < len$, $i \neq j$, 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 is to reward swaps that progress the list towards a solution, and to penalize swaps that have no effect, result in an out-of-bounds array error, or move the list away from being sorted. This reward and penalty were varied, and it was found that small rewards and penalties led most easily to convergence.
By way of comparison, this set of actions is more atomic than those used by Kinnear. There are no explicit “order” instructions. These are not as likely to automatically lead to a sorting algorithm because there is a lack of explicit iteration. The only explicitly defined iteration is the main while-loop, which is only necessary for repeated executions of the policy itself. Any iteration resulting in the sorting algorithm is learned by the system through $\text{INCI}$, $\text{INCJ}$, $\text{SETZERO}$, and $\text{SETJZERO}$.

### 4.4 Examples of RP Concepts Illustrated by RPSort

The first elaboration of the above concepts as used by RPSort is shown in Figure 4.1. It contains the general structure of the state representation. It is divided into the data-specific and RP-specific portions (data-specific on the top, RP-specific on the bottom). Question marks indicate values that are set during execution (because this is the general state structure, not a specific example of the state). The next figure, Figure 4.2, 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 4.3 demonstrates the way an action can affect the training instance. In the case of RPSort, $\text{SWAP}$ is the only action that modifies the training instance in any way. As before, the arrow indicates a transition and the label above the arrow indicates the action. The last figure, Figure 4.4, demonstrates some of the non-determinism present in the RPSort system. Notice that for both $\text{state 1a}$ and $\text{state 1b}$ the RP specific portion is exactly the same. The only difference is in the value of $i$ in the data specific portion. Because the RP specific part of both states are the same, states $1a$ and $1b$ are considered equivalent. The same action is performed in both cases, but the resulting states are different. From the perspective of the learning system, the action of $\text{INCI}$ is non-deterministic because that action led from one state to two different states.

### 4.5 Policy to Program Conversion

The policy is represented as a set of states and actions. As explained in Section 4.1, the action selected is the action that provides the largest Q-value. The set of states is comprised of all states with distinct RP-specific portions. Once the policy is learned, unneeded states are pruned. This is achieved by using the policy to sort all of the lists that are the same size as those used in training and tracking which states are not visited. The unvisited states are discarded.

Next, a decision tree is formed. Because there are a relatively small number of resulting states, an exhaustive search is performed to create the tree with the fewest leaf nodes. This results in an easy-to-understand and rapidly-executing program. This process takes place in the same fashion as described in Appendix A. Program execution takes place as shown in Figure 3.1. The state
is passed through the decision tree until a leaf node is reached. The action at that leaf node is performed, modifying the state. This process repeats until the \textit{TERMINATE} action is performed.

### 4.6 Training Data

Training is simple. A list length is provided (\textit{len}), and an exhaustive set of lists is created (all possible permutations of the numbers 1 to \textit{len}). These lists are ordered lexicographically. Experiments to determine the size of \textit{len} needed to achieve generality are discussed in Chapter 5.

The training set of lists initially starts out with the backwards list (the last list in the lexicographical ordering). The reason for this choice is that 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 \textit{len} is large.

### 4.7 Example of RPSort Training

This section provides specific examples of the RP concepts discussed in Chapter 3. The initial state will be presented, as well as the manner in which the state space 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 with RL and Q-Learning. The values and formulas described in Section 4.1 are used.

Figure 4.5 displays the initial state. Figures 4.7 through 4.9 demonstrate a possible route taken at the very beginning of training and the way in which the state space grows with the transitions. A random walk occurs because there is no reward information available for any of the actions. Figure 4.6 demonstrates the new state derived by the action \textit{SETZERO}. The next action executed is \textit{INCI}, and is shown in Figure 4.7. The next action chosen randomly is \textit{SWAP}, the first action chosen that has an affect on the training instance (Figure 4.8). The action sorts list elements 0 and 1 with respect to each other, providing a reward of 10. The random walk continues until \textit{TERMINATE} is chosen.
Consider now that the world has been explored until a goal state is reached (the \textit{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. The same training instance is used in the state to further explore and grow the state space. Figure 4.9 shows the first and last state explored in Figures 4.5 through 4.8. The ellipses indicate the other states visited between the two shown in Figure 4.9. This figure also shows that the first action performed from the initial state is \textit{INCJ}, transitioning to the new state shown in the figure. Since the state is new, there is no reward information to guide the search, so an action is selected at random. The next action chosen, shown in Figure 4.10, is the \textit{SWAP} action. As the figure shows, this transitions to the last state described in the earlier example. Training again continues until the terminate action is performed. This 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.

Next, two different executions of the \textit{TERMINATE} action are demonstrated. Figure 4.11 shows the execution of the \textit{TERMINATE} action when the list is not sorted. The reward for executing the \textit{TERMINATE} action in that situation is $-160$. The system rapidly learns that the \textit{TERMINATE} action should not be used in that state. In Figure 4.12, the list is sorted when \textit{TERMINATE} is performed. Thus, the reward for performing the \textit{TERMINATE} action in that state is 100.

![Figure 4.1: State representation for RPSort](image)

Figure 4.1: State representation for RPSort
Figure 4.2: State transition example

Figure 4.3: Training Instance Modification Example

Figure 4.4: Non-determinism example

Figure 4.5: Initial state
Figure 4.6: First state transition

Figure 4.7: Second state transition
Figure 4.8: Third state transition

Figure 4.9: New state transition from the initial state
Figure 4.10: State transition leading back to original set

Figure 4.11: Penalized terminate example

Figure 4.12: Rewarded terminate example
Chapter 5

Results

This chapter describes the results obtained from the system described in Chapter 4. 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.

5.1 The Learned Sorting Algorithm: RPSort

One of the most interesting and surprising results is the number and size of the lists needed to learn a general sorting algorithm. Lists of size 4 are sufficient to create a general sorter, and often only the list \([4, 3, 2, 1]\) is needed. Longer lists generally came up with the same solution (with some minor random variations due to the non-deterministic nature of the technique). Another interesting result was the lack of any use of the 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.

Before pruning unnecessary states, the policy most commonly found contains 226 states. By executing the policy on all the lists of size 4 and keeping only the states visited, the policy is reduced to 17 states. Table 5.1 shows the pruned policy. Note that the ID column of Table 5.1 is not a part of the policy. It is present for easy reference. Figure 5.1 displays the program consistently produced by RPSort. This program was derived in the manner described in Appendix A. The main while-loop keeps the policy executing until a goal state is reached. Since the goal state is any state in which the last action performed was \(TERMINATE\), this forms the conditional for the while-loop. The body of the loop is the decision tree (in pseudocode) produced automatically from the learned policy using the technique demonstrated in Appendix A. 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. Figure 5.2 demonstrates the state transitions that occur while sorting the list \([2, 3, 1]\). Each state is labelled according to the ID column in Table 5.1. Program execution occurs as discussed in Section 4.5.

It is impossible to exhaustively check all possible inputs to determine algorithm correctness. An analysis of the algorithm is needed to determine if it is, in fact, a generalized sorting algorithm.
The first thing to note is that the RP system learned to only terminate when \( i = \text{len} \) and \( j = \text{len} \). Assume the algorithm has been passed a sorted list. Whenever \( i = j \), \textit{INCJ} is performed. Whenever \( \text{list}[i] \leq \text{list}[j] \) and \( \text{lastact} = \text{INCJ} \), \textit{INCI} is performed. Since the list is sorted, the program alternates between performing \textit{INCJ} and \textit{INCI}. When \( j = \text{len} \) and \( \text{lastact} = \text{INCJ} \), \textit{INCI} is performed. Because both variables \( i \) and \( j \) have been incremented an equal number of times it is obvious that at this point \( i = \text{len} \) and \( j = \text{len} \). The program terminates correctly when the list is sorted.

Now assume the algorithm has been passed an unsorted list. There must therefore be at least two adjacent elements somewhere in the list that are out of order with respect to each other. The entire list before that pair can be considered sorted. Thus, the behavior of \( i \) and \( j \) will be exactly the same as if the entire list was sorted, until that adjacent out-of-order pair is reached. When that pair is reached, the last action performed will have been \textit{INCJ} (due to the way the algorithm operates). Prior to that action, \( i = j \). Thus, when that pair is reached, \( i = j - 1 \). The adjacent out-of-order pair is swapped, putting them in order with respect to each other. \textit{INCJ} is then performed. At this point, \( i = j - 2 \). This gap increases each time \( \text{list}[i] > \text{list}[j] \). When \( j = \text{len} \), the last action performed is \textit{INCJ}. The algorithm specifies that \textit{INCI} is performed next. Since, at best, \( i = j - 2 \) before this action is performed, incrementing \( i \) will not make \( i = \text{len} \). The algorithm then sets \( i = 0 \) and \( j = 0 \) and starts over. Thus, if there are even only two elements that are out of place, the algorithm will not terminate until at least the next pass through the list. Therefore, RPSort is a generalized sort.

<table>
<thead>
<tr>
<th>ID</th>
<th>last action</th>
<th>( i )</th>
<th>( j )</th>
<th>( k )</th>
<th>( i ) vs. ( j )</th>
<th>list[( i )] vs. list[( j )]</th>
<th>best action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\textit{NOOP}</td>
<td>( i = 0 )</td>
<td>( j = 0 )</td>
<td>( k = 0 )</td>
<td>( i = j )</td>
<td>list[( i )] \leq list[( j )]</td>
<td>\textit{INCJ}</td>
</tr>
<tr>
<td>2</td>
<td>\textit{INCI}</td>
<td>( i =? )</td>
<td>( j =? )</td>
<td>( k = 0 )</td>
<td>( i = j )</td>
<td>list[( i )] \leq list[( j )]</td>
<td>\textit{INCJ}</td>
</tr>
<tr>
<td>3</td>
<td>\textit{INCI}</td>
<td>( i =? )</td>
<td>( j =? )</td>
<td>( k = 0 )</td>
<td>( i &lt; j )</td>
<td>list[( i )] &gt; list[( j )]</td>
<td>\textit{SWAP}</td>
</tr>
<tr>
<td>4</td>
<td>\textit{INCI}</td>
<td>( i =? )</td>
<td>( j =? )</td>
<td>( k = 0 )</td>
<td>( i &lt; j )</td>
<td>list[( i )] \leq list[( j )]</td>
<td>\textit{INCJ}</td>
</tr>
<tr>
<td>5</td>
<td>\textit{INCI}</td>
<td>( i =? )</td>
<td>( j = \text{len} )</td>
<td>( k = 0 )</td>
<td>( i &lt; j )</td>
<td>list[( i )] \leq list[( j )]</td>
<td>\textit{SETJZERO}</td>
</tr>
<tr>
<td>6</td>
<td>\textit{INCI}</td>
<td>( i = \text{len} )</td>
<td>( j = \text{len} )</td>
<td>( k = 0 )</td>
<td>( i = j )</td>
<td>list[( i )] \leq list[( j )]</td>
<td>\textit{TERMINATE}</td>
</tr>
<tr>
<td>7</td>
<td>\textit{INCI}</td>
<td>( i =? )</td>
<td>( j =? )</td>
<td>( k = 0 )</td>
<td>( i &lt; j )</td>
<td>list[( i )] &gt; list[( j )]</td>
<td>\textit{SWAP}</td>
</tr>
<tr>
<td>8</td>
<td>\textit{INCI}</td>
<td>( i =? )</td>
<td>( j =? )</td>
<td>( k = 0 )</td>
<td>( i &lt; j )</td>
<td>list[( i )] \leq list[( j )]</td>
<td>\textit{INCI}</td>
</tr>
<tr>
<td>9</td>
<td>\textit{INCI}</td>
<td>( i =? )</td>
<td>( j = \text{len} )</td>
<td>( k = 0 )</td>
<td>( i &lt; j )</td>
<td>list[( i )] \leq list[( j )]</td>
<td>\textit{INCI}</td>
</tr>
<tr>
<td>10</td>
<td>\textit{INCI}</td>
<td>( i = 0 )</td>
<td>( j =? )</td>
<td>( k = 0 )</td>
<td>( i &lt; j )</td>
<td>list[( i )] \leq list[( j )]</td>
<td>\textit{SWAP}</td>
</tr>
<tr>
<td>11</td>
<td>\textit{INCI}</td>
<td>( i = 0 )</td>
<td>( j = \text{len} )</td>
<td>( k = 0 )</td>
<td>( i &lt; j )</td>
<td>list[( i )] \leq list[( j )]</td>
<td>\textit{INCI}</td>
</tr>
<tr>
<td>12</td>
<td>\textit{INCI}</td>
<td>( i = 0 )</td>
<td>( j = 0 )</td>
<td>( k = 0 )</td>
<td>( i = j )</td>
<td>list[( i )] \leq list[( j )]</td>
<td>\textit{SETJZERO}</td>
</tr>
<tr>
<td>13</td>
<td>\textit{SETJZERO}</td>
<td>( i = 0 )</td>
<td>( j = 0 )</td>
<td>( k = 0 )</td>
<td>( i = j )</td>
<td>list[( i )] \leq list[( j )]</td>
<td>\textit{INCI}</td>
</tr>
<tr>
<td>14</td>
<td>\textit{SETJZERO}</td>
<td>( i =? )</td>
<td>( j = 0 )</td>
<td>( k = 0 )</td>
<td>( i &gt; j )</td>
<td>list[( i )] &gt; list[( j )]</td>
<td>\textit{SETJZERO}</td>
</tr>
<tr>
<td>15</td>
<td>\textit{SETJZERO}</td>
<td>( i = 0 )</td>
<td>( j = 0 )</td>
<td>( k = 0 )</td>
<td>( i = j )</td>
<td>list[( i )] \leq list[( j )]</td>
<td>\textit{INCI}</td>
</tr>
<tr>
<td>16</td>
<td>\textit{SWAP}</td>
<td>( i =? )</td>
<td>( j =? )</td>
<td>( k = 0 )</td>
<td>( i &lt; j )</td>
<td>list[( i )] \leq list[( j )]</td>
<td>\textit{INCI}</td>
</tr>
<tr>
<td>17</td>
<td>\textit{SWAP}</td>
<td>( i = 0 )</td>
<td>( j =? )</td>
<td>( k = 0 )</td>
<td>( i &lt; j )</td>
<td>list[( i )] \leq list[( j )]</td>
<td>\textit{INCI}</td>
</tr>
</tbody>
</table>

Table 5.1: RPSort Policy
list[] ← list to be sorted
belief ← false
len ← length of list[]
i ← 0
j ← 0
lastact ← NOOP
while lastact ≠ (TERMINATE) do
  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 ← (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 ++
  else
    if i ≠ 0
      lastact ← (SETIZERO)
      i ← 0
    else
      lastact ← (INCJ)
      j ++

Figure 5.1: RPSort
Figure 5.2: State transitions for sorting a list
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 5.3 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-trivial lists, RPSort is superior to the other three algorithms.

Random Program Generation (RPG) refers to randomly creating a computer program using the same state representation used by RP. The reason for RPG is to ascertain whether or not RP is just randomly creating programs until a successful program is found, instead of performing a directed search. 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 was able to successfully sort each 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.

5.2 Comparison between Reinforcement Programming and Genetic Programming for Generating a Sort

RP appears more efficient at generating a sorting algorithm than GP when comparing the results in this chapter 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 system (according to the training technique described above), 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 a solution, one 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.
Chapter 6

Conclusions and Future Work

Reinforcement programming is able to generate a fast, efficient, general sorting algorithm, competitive with the most common iterative sorting algorithms available. It is able to come up with a more efficient algorithm than genetic programming, using more atomic operations than were used to evolve bubblesort through GP. The state representation and transition is more “natural” to a computer than the tree structure used by GP.

Future applications for RP are taken from [18]. They include creating a parity-checking function, an algorithm for solving two-variable linear systems, and a function for 1-dimensional pattern recognition. While these applications are trivial, they will further explore the strengths and limitations of reinforcement programming. It is expected that as RP is further explored, it will be able to handle real-world 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. Future work also involves exploring different techniques of handling POMDP and non-markovian reinforcement learning problems and applying those techniques to reinforcement programming. An important future task involves stack equivalency. In order 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.

Additional work involves actual implementation. As with many RL applications the state space can rapidly get large. If there are currently $n$ bound variables, a new bound variable added to the state adds $n$ booleans in order to reference its value with the values of all the other bound variables. This results in a $2^n$-fold increase in the number of states. Work to simplify this situation will improve the scalability of 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.
Appendix A

Decision Tree Construction Example

For the sake of ease, the RP-specific portion of the state will be referred to simply as the state unless explicitly stated. This is done while acknowledging that the entire state is, in truth, made up of the RP-specific and the data-specific. The policy consists of the set of (possibly pruned) states where each RP-specific portion is unique. A split refers to dividing a set of states into groups according to a specified 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 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 computer code.

Table A.1 shows a trivial policy. The state number is not an aspect of the state, but rather is an index for easy referencing. This program increments a variable \(i\) until it equals 2. The available actions are NOOP (not selectable for execution), INCI which increments \(i\), and TERMINATE which ends policy execution. The step-by-step process for constructing the decision trees used in this thesis will be demonstrated using this policy. It is important to note that, in this case, there is no data-specific portion of the state, just an RP-specific portion containing the value of the variable \(i\) and the last action performed.

For the sake of this example, the first aspect of the state used in the exhaustive search is the last action performed. Figure A.1 shows the distribution caused by this particular split.

<table>
<thead>
<tr>
<th>State Number</th>
<th>last action</th>
<th>(i)</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NOOP</td>
<td>0</td>
<td>INCI</td>
</tr>
<tr>
<td>2</td>
<td>INCI</td>
<td>1</td>
<td>INCI</td>
</tr>
<tr>
<td>3</td>
<td>INCI</td>
<td>2</td>
<td>TERMINATE</td>
</tr>
</tbody>
</table>

Table A.1: A Trivial Policy
state 1 is all by itself after this split, there is only one “best action” in that group. Thus, that node becomes a leaf node. The other node contains states 2 and 3, both of which have a different “best action”. Thus, another split must be performed. The only remaining aspect of the state that has not been split on is the value of $i$. Figure A.2 shows the results of the split. Because of the split, all the states have entered leaf nodes, resulting in a complete tree. This tree has 3 leaf nodes and a depth of 2. If there were more than two features in the state, the system would store this tree, backup to the tree created before the split on $i$, and then split on another aspect of the state.

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 the next aspect of the state, in this case $i$. Figure A.3 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, the same number as the first tree created. However, it has a depth of 1. Since 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.
Figure A.2: Splitting on last action then $i$

Figure A.3: Splitting on $i$
Appendix B

Reinforcement Programming: Binary Addition

This appendix reviews the principles of RP by introducing three applications of reinforcement programming: learning to simulate a full adder circuit, learning to perform binary incrementing, and learning to perform binary addition. Each application is an extension of the one previous to it, and provides a gradual introduction to the practical application of RP. Each section will review one of the aforementioned applications, as well as the state representation for each one, the actions chosen, and the parameters used. Each application is also accompanied by the results of the experiments.

For each application, non-deterministic Q-learning is used. The same formulas and techniques described in Chapter 4.1 are used. By exploring different rewards and penalties, it became clear that the ratio between the reward and the penalty, rather than the reward and penalty values themselves, were important. By holding the reward at 100 and varying the penalty and $\gamma$, it was determined that convergence occurred most rapidly with a penalty of $-100$ and a $\gamma$ of $0.9$.

As with RPSort, a NOOP action is provided as a starting “last state” for the state the training instance starts in. This is made available to each application.

B.1 RP: 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. Figure B.1 shows the basic circuit diagram using only AND, OR, and NOT gates. The gate symbols came from http://hyperphysics.phy-astr.gsu.edu/hbase/hframe.html.

$X_1$, $X_2$, and $X_3$ are the inputs, and $A_1$ and $A_0$ are the outputs, with $A_i$ representing the $2^i$ bit. The binary relational formulas represented in Figure B.1 are as follows:

$$A_1 = (X_1 \text{ AND } X_2) \text{ OR } (X_1 \text{ AND } X_3) \text{ OR } (X_2 \text{ AND } X_3) \quad (B.1)$$

$$A_0 = \begin{cases} 
(X_1 \text{ AND } \overline{X_2} \text{ AND } X_3) \text{ OR } \\
(X_1 \text{ AND } \overline{X_2} \text{ AND } \overline{X_3}) \text{ OR } \\
(X_1 \text{ AND } X_2 \text{ AND } X_3) \text{ OR } \\
(\overline{X_1} \text{ AND } X_2 \text{ AND } \overline{X_3}) 
\end{cases} \quad (B.2)$$
Figure B.1: Circuit Diagrams for a Full Adder
### Table B.1: Full Adder Inputs and Outputs

<table>
<thead>
<tr>
<th>X3</th>
<th>X2</th>
<th>X1</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table B.2: Full Adder Action Set

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOP</td>
<td>A “non-action”. Not selectable for execution</td>
</tr>
<tr>
<td>TERMINATE</td>
<td>Ends policy execution</td>
</tr>
<tr>
<td>A1 = True</td>
<td>Sets output A1 to true (1)</td>
</tr>
<tr>
<td>A1 = False</td>
<td>Sets output A1 to false (1)</td>
</tr>
<tr>
<td>A0 = True</td>
<td>Sets output A0 to true (0)</td>
</tr>
<tr>
<td>A0 = False</td>
<td>Sets output A0 to false (0)</td>
</tr>
</tbody>
</table>

### B.1.1 Training Set and State Representation

A training instance for the full adder problem consists of three bits. The entire training set involved all eight possible instances the three bits could represent. The corresponding outputs are in Table B.1.

Because it is possible to exhaustively list all inputs, the state representation is very simple. It consists of three booleans labeled to correspond to the inputs. It also has two integer variables labeled to correspond to the outputs. 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.

### B.1.2 Actions and Transition

Five actions were made available to the system, and are described in Table B.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.
Table B.3: Full Adder Policy

<table>
<thead>
<tr>
<th>Last action</th>
<th>X3</th>
<th>X2</th>
<th>X1</th>
<th>A1</th>
<th>A0</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOP</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>A0=True</td>
</tr>
<tr>
<td>NOOP</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>?</td>
<td>?</td>
<td>A1=False</td>
</tr>
<tr>
<td>NOOP</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>A0=True</td>
</tr>
<tr>
<td>NOOP</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>?</td>
<td>?</td>
<td>A1=True</td>
</tr>
<tr>
<td>NOOP</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>A0=True</td>
</tr>
<tr>
<td>NOOP</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>?</td>
<td>?</td>
<td>A0=False</td>
</tr>
<tr>
<td>NOOP</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>A1=True</td>
</tr>
<tr>
<td>NOOP</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>?</td>
<td>?</td>
<td>A0=True</td>
</tr>
<tr>
<td>A1=True</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>?</td>
<td>A0=False</td>
</tr>
<tr>
<td>A1=True</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>TERMINATE</td>
</tr>
<tr>
<td>A1=True</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>?</td>
<td>A0=True</td>
</tr>
<tr>
<td>A1=True</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>TERMINATE</td>
</tr>
<tr>
<td>A1=False</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>TERMINATE</td>
</tr>
<tr>
<td>A1=False</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>?</td>
<td>A0=True</td>
</tr>
<tr>
<td>A1=False</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>TERMINATE</td>
</tr>
<tr>
<td>A0=True</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>1</td>
<td>A1=False</td>
</tr>
<tr>
<td>A0=True</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>?</td>
<td>1</td>
<td>A1=False</td>
</tr>
<tr>
<td>A0=True</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>0</td>
<td>A1=False</td>
</tr>
<tr>
<td>A0=False</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>TERMINATE</td>
</tr>
<tr>
<td>A0=False</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>?</td>
<td>0</td>
<td>A1=True</td>
</tr>
<tr>
<td>A0=False</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>TERMINATE</td>
</tr>
</tbody>
</table>

B.1.3 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, and is shown in Table B.3. Once the pruned policy is obtained, an exhaustive search is performed to create a decision tree with the least number of leaf nodes. In the event that two decision trees gave the same number of leaf nodes, the tree with the shallowest depth was used. An example of how this tree is formed is presented in Appendix A. The decision tree itself is given in Figure B.2. This decision tree is repeatedly executed until the TERMINATE action is performed.

Since it is possible to exhaustively enumerate all possible inputs, it was possible to test all inputs to ensure that the above program does, in fact, perform the same function as a full adder circuit. In fact, by performing an analysis of the program to transform it into a binary circuit the exact same circuit as is presented in Figure B.1 is generated.
if (last act == NOOP)
  if (!X1)
    if (!X2)
      if (!X3)
        A0 = False
      else
        A1 = False
    else
      if (!X3)
        A1 = False
      else
        A1 = True
  else
    if (!X2)
      if (!X3)
        A1 = False
      else
        A1 = True
    else
      if (!X3)
        A1 = True
      else
        A0 = True
else if (last act == (A1=True))
  if (A0 < 0)
    A0=False
  else if (A0==1)
    Terminate
else if (last act == (A1=False))
  if (A0 < 0)
    A0=True
  else if (A0==0)
    TERMINATE
else if (last act == (A0=True))
  if (A1 < 0)
    A1=True
  else if (A1 == 1)
    TERMINATE
else if (last act == (A0=False))
  if (A1 < 0)
    A1 = False
  else if (A1 == 1)
    TERMINATE

Figure B.2: Full Adder Decision Tree
B.2 RP: 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 that the input number has nothing but bits set to 1, the resulting number has nothing but 0’s. This is purely an implementation decision to keep the number of bits in the output the same as those in the input.

This problem has an interesting abstraction. In essence, the binary incrementer problem is 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 it’s successor in an ordered list. The successor for each input may be known, but not the actual mapping. A system that can automatically generate that mapping is a useful tool.

B.2.1 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 itself 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 booleans: whether or not \(i = 0\), \(j = 0\), \(i = len\), and \(j = len\). The last action performed is also a part of the RP-specific portion of the state. It also contains 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 \(dest[i] = 0\). Both of these stipulations are to deal with array out-of-bounds situations. Experiments performed without including \(source[i-1]\) and \(dest[i-1]\) failed to lead to convergence. This suggests that the problem is at best a partially observable markov decision process (POMDP) [5] and at worst entirely non-markovian. By giving the state the additional information, the system was close enough to being markovian that Q-learning worked correctly.

B.2.2 Actions and Transition

Table B.4 contains the actions used by this RP system. As with the last example, the goal states are any state with \(TERMINATE\) as the last action performed. The following restriction is
<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOP</td>
<td>A &quot;non-action&quot;. Not selectable for execution</td>
</tr>
<tr>
<td>TERMINATE</td>
<td>Ends policy execution</td>
</tr>
<tr>
<td>INCI</td>
<td>Increments $i$</td>
</tr>
<tr>
<td>SETIZERO</td>
<td>Sets $i = 0$</td>
</tr>
<tr>
<td>INCJ</td>
<td>Increments $j$</td>
</tr>
<tr>
<td>SETJZERO</td>
<td>Sets $j = 0$</td>
</tr>
<tr>
<td>SETTRUE</td>
<td>Sets $\text{dest}[i]$ to 1</td>
</tr>
<tr>
<td>SETFALSE</td>
<td>Sets $\text{dest}[i]$ to 0</td>
</tr>
</tbody>
</table>

Table B.4: Binary Incrementer Actions

placed on action selection: if $i = \text{len}$, then \text{SETFALSE} and \text{SETTRUE} cannot be chosen. This is to avoid out-of-bounds array errors.

A reward is provided when the \text{TERMINATE} action is performed while the \text{dest} binary number has all its bits set to the correct value. If the the \text{TERMINATE} action is executed in any other situation, a penalty is provided.

B.2.3 Results and Policy Translation

The system is trained on all training instances at each iteration. The most commonly learned policy contains 254 states. After executing the policy on all the training instances, the policy is pruned to 9 states and is presented in Table B.5. Note that an entry of $i = ?$ indicates that $i$ is somewhere between 0 and $\text{len}$. An exhaustive search to create the tree with the fewest leaves and shallowest depth is executed, again using the technique demonstrated in Appendix A. The resulting decision tree is presented in Figure B.3. It is interesting to note that the variable $j$ provided to the state was neither needed nor used. Furthermore, although convergence would not occur without having $\text{source}[i-1]$ and $\text{dest}[i-1]$ as part of the state, the resulting decision tree uses neither value. This suggests that while the decision tree generated from the policy did not make use of those values, the RP system required those values to put the rewards the system received into proper context. Given that convergence to a solution does not occur without those values as part of the state, and that convergence does occur with those values as a part of the state, it is clear that this problem is either a POMDP or completely non-markovian. The decision tree generated from the learned policy is repeatedly executed until the goal state is reached (the \text{TERMINATE} action is performed).

Because there is no exhaustive set of all possible inputs like there was for the full adder application, this learned program must be analyzed to prove correctness. 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 $i = \text{len}$. This is the exact procedure for
<table>
<thead>
<tr>
<th>last action</th>
<th>$i$</th>
<th>$j$</th>
<th>source[$i$]</th>
<th>source[$i$-1]</th>
<th>dest[$i$]</th>
<th>dest[$i$-1]</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOP</td>
<td>$i = 0$</td>
<td>$j = 0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SETTRUE</td>
</tr>
<tr>
<td>NOOP</td>
<td>$i = 0$</td>
<td>$j = 0$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SETFALSE</td>
</tr>
<tr>
<td>INCI</td>
<td>$i = ?$</td>
<td>$j = 0$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SETTRUE</td>
</tr>
<tr>
<td>INCI</td>
<td>$i = ?$</td>
<td>$j = 0$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>SETFALSE</td>
</tr>
<tr>
<td>INCI</td>
<td>$i = \text{len}$</td>
<td>$j = 0$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>TERMINATE</td>
</tr>
<tr>
<td>SETTRUE</td>
<td>$i = ?$</td>
<td>$j = 0$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>TERMINATE</td>
</tr>
<tr>
<td>SETTRUE</td>
<td>$i = 0$</td>
<td>$j = 0$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>TERMINATE</td>
</tr>
<tr>
<td>SETFALSE</td>
<td>$i = ?$</td>
<td>$j = 0$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>INCI</td>
</tr>
<tr>
<td>SETFALSE</td>
<td>$i = ?$</td>
<td>$j = 0$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>INCI</td>
</tr>
</tbody>
</table>

Table B.5: Binary Incrementer Policy

if $i \neq \text{len}$
if source[$i$] = 0
    if dest[$i$] = 0
        SETTRUE
    else
        TERMINATE
else
    if dest[$i$] = 0
        INCI
    else
        SETFALSE
else
    TERMINATE

Figure B.3: Binary Incrementer Decision Tree
incrementing a binary number.

B.3 RP: General Binary Adder

A general binary adder takes in two binary numbers of any length (number of digits) and returns a binary number representing the sum of those two binary numbers. It is assumed that both input binary numbers are the same length (if they are not, the shorter one can have 0’s added to the beginning to extend it to the same length as the longer one). The output number is also the same length as the two input numbers. 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, $110 + 101 = 1011$, however the binary adder would return $110 + 101 = 011$. As with the binary incrementer, it was an implementation decision to require the solution to have the same number of bits as the inputs.

As with the binary incrementer, this problem has an interesting abstraction. This problem creates a many-to-one mapping ($2^n$ 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.

B.3.1 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 $j$. $i$ is a bound variable and $j$ is a free variable. The data-specific portion also has the two input binary numbers, $source_1$ and $source_2$. It also contains the binary number that $source_1$ and $source_2$ maps to, called $map$. It is important to remember that this system is being modelled as a mapping-learning problem (similar to standard machine learning problems). Lastly, it contains $dest$, an array of integers the same length as the input numbers. 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 data-specific portion also has a variable called $len$, which is set to the length of the input numbers. The RP-specific portion contains four booleans: whether or not $i = 0$, $i = len$, $j = 0$, and $j = len$. The last action performed is also a part of the RP-specific portion. It also contains
<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOP</td>
<td>A &quot;non-action&quot;. Not selectable for execution</td>
</tr>
<tr>
<td>TERMINATE</td>
<td>Ends policy execution</td>
</tr>
<tr>
<td>INCi</td>
<td>Increments i</td>
</tr>
<tr>
<td>SETiZERO</td>
<td>Sets $i = 0$</td>
</tr>
<tr>
<td>INCi</td>
<td>Increments j</td>
</tr>
<tr>
<td>SETjZERO</td>
<td>Sets $j = 0$</td>
</tr>
<tr>
<td>SETTRUE</td>
<td>Sets $dest[i] = 1$</td>
</tr>
<tr>
<td>SETFALSE</td>
<td>Sets $dest[i] = 0$</td>
</tr>
</tbody>
</table>

Table B.6: Binary Adder Actions

A variable that indicates how many zeros are indexed by $source_1[i]$ and $source_2[i]$ (either 0, 1, or 2 zeros) called $zcount$. Similarly, it contains a variable that indicates how many ones are indexed by $source_1[i]$ and $source_2[i]$ called $ocount$. Similar variables, called last$zcount$ and last$ocount$ track how many zeros and ones (respectively) are referenced by $source_1[i-1]$ and $source_2[i-1]$. Lastly, two variables $dloc$ and last$dloc$ reference the value in $dest[i]$ and $dest[i-1]$ respectively. Experiments performed without last$zcount$, last$ocount$, and last$dloc$ failed to converge, suggesting that general binary adder problem is either POMDP or non-markovian. Making those variables a part of the RP-specific portion of the state made the entire problem close enough to being markovian that the system could converge. The elements of $source_1$, $source_2$, and $dest$ are referenced as a zero-based array (0 to $len-1$). If $i = 0$, then last$zcount = 2$, last$ocount = 0$, and last$dloc = -1$. If $i = len$, then $zcount = 2$, $ocount = 0$, and $dloc = -1$. These implementation decisions were put in place to handle out-of-bound array errors.

B.3.2 Actions and Transition

The actions used by the general binary adder system are in Table B.6. As usual, 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. Furthermore, 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. Again, it is important to note that this was not using an addition algorithm 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.

There are restrictions placed on the actions that can be selected, depending on the current state. 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 achievable by repeatedly
performing a reward-giving action.

B.3.3 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. The policy is presented in Table B.7. Note that an entry of $i=?$ indicates that the value of that variable is somewhere between 0 and $len$. Also note that, although $j$ was part of the state, it is never used. Thus, in an endeavor to present the state in a format that will fit on a single page, the $j$ aspect of the RP-specific portion of the state has been removed. The same approach used to create the decision tree for the full adder and the binary incrementer is used to create the decision tree for the binary adder. The decision tree is repeatedly executed until the $TERMINATE$ action is performed. This decision tree is presented in Figure B.4.

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 easy to tell that the learned program follows the proper technique for adding together two binary numbers. Table B.8 contains a case-by-case summary of how $dest[i]$ is set. By noting that the algorithm only terminates when $i = len$, and never increments $i$ when $dloc = ?$, it becomes clear that this algorithm sets every bit. Combined with the data from Table B.8, it is obvious that this program is indeed a generalized binary adder.

B.4 Comparison to Random Program Generation

The simplicity of the resulting programs can imply that random program generation (RPG) could lead to a solution just as quickly as RP. Consequently, the above three applications were compared to RPG. Each random policy that was created was tested using the entire training set that had been used by the corresponding RP application. One billion trials were performed for each application. Not even after a billion trials was a program randomly generated that was able to perform correctly on the entire training set. Since each of the three RP applications mentioned in this appendix were found in 50,000 iterations or less, it is clear that RP performed much better than random in these instances.
<table>
<thead>
<tr>
<th>last action</th>
<th>i</th>
<th>zcount</th>
<th>ocount</th>
<th>dloc</th>
<th>lastzcount</th>
<th>lastocount</th>
<th>lastdloc</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOP</td>
<td>i = 0</td>
<td>0</td>
<td>2</td>
<td>?</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>SETFALSE</td>
</tr>
<tr>
<td>NOOP</td>
<td>i = 0</td>
<td>1</td>
<td>1</td>
<td>?</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>SETTRUE</td>
</tr>
<tr>
<td>NOOP</td>
<td>i = 0</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>SETFALSE</td>
</tr>
<tr>
<td>INCI</td>
<td>i = ?</td>
<td>0</td>
<td>2</td>
<td>?</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>SETTRUE</td>
</tr>
<tr>
<td>INCI</td>
<td>i = ?</td>
<td>0</td>
<td>2</td>
<td>?</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>SETTRUE</td>
</tr>
<tr>
<td>INCI</td>
<td>i = ?</td>
<td>0</td>
<td>2</td>
<td>?</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>SETTRUE</td>
</tr>
<tr>
<td>INCI</td>
<td>i = ?</td>
<td>0</td>
<td>2</td>
<td>?</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>SETFALSE</td>
</tr>
<tr>
<td>INCI</td>
<td>i = ?</td>
<td>0</td>
<td>2</td>
<td>?</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>SETFALSE</td>
</tr>
<tr>
<td>INCI</td>
<td>i = ?</td>
<td>1</td>
<td>1</td>
<td>?</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>SETFALSE</td>
</tr>
<tr>
<td>INCI</td>
<td>i = ?</td>
<td>1</td>
<td>1</td>
<td>?</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>SETFALSE</td>
</tr>
<tr>
<td>INCI</td>
<td>i = ?</td>
<td>1</td>
<td>1</td>
<td>?</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>SETTRUE</td>
</tr>
<tr>
<td>INCI</td>
<td>i = ?</td>
<td>1</td>
<td>1</td>
<td>?</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>SETTRUE</td>
</tr>
<tr>
<td>INCI</td>
<td>i = ?</td>
<td>1</td>
<td>1</td>
<td>?</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>SETTRUE</td>
</tr>
<tr>
<td>INCI</td>
<td>i = ?</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>SETTRUE</td>
</tr>
<tr>
<td>INCI</td>
<td>i = ?</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>SETTRUE</td>
</tr>
<tr>
<td>INCI</td>
<td>i = ?</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>SETFALSE</td>
</tr>
<tr>
<td>INCI</td>
<td>i = ?</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>SETFALSE</td>
</tr>
<tr>
<td>INCI</td>
<td>i = ?</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>TERMINATE</td>
</tr>
<tr>
<td>INCI</td>
<td>i = ?</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>TERMINATE</td>
</tr>
<tr>
<td>INCI</td>
<td>i = ?</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>TERMINATE</td>
</tr>
<tr>
<td>SETTRUE</td>
<td>i = ?</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>INCI</td>
</tr>
<tr>
<td>SETTRUE</td>
<td>i = ?</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>INCI</td>
</tr>
<tr>
<td>SETTRUE</td>
<td>i = ?</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>INCI</td>
</tr>
<tr>
<td>SETTRUE</td>
<td>i = ?</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>INCI</td>
</tr>
<tr>
<td>SETTRUE</td>
<td>i = ?</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>INCI</td>
</tr>
<tr>
<td>SETTRUE</td>
<td>i = ?</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>INCI</td>
</tr>
<tr>
<td>SETTRUE</td>
<td>i = ?</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>INCI</td>
</tr>
<tr>
<td>SETTRUE</td>
<td>i = ?</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>INCI</td>
</tr>
<tr>
<td>SETTRUE</td>
<td>i = ?</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>INCI</td>
</tr>
<tr>
<td>SETTRUE</td>
<td>i = 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>?</td>
<td>INCI</td>
</tr>
<tr>
<td>SETFALSE</td>
<td>i = ?</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>INCI</td>
</tr>
<tr>
<td>SETFALSE</td>
<td>i = ?</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>INCI</td>
</tr>
<tr>
<td>SETFALSE</td>
<td>i = ?</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>INCI</td>
</tr>
</tbody>
</table>

Table B.7: Binary Adder Policy
if $i \neq len$
  if $dloc < 0$
    if $zcount = 0$
      if $lastzcount = 0$
        SETTRUE
      else if $lastzcount = 1$
        if $lastdloc = 0$
          SETTRUE
        else if $lastdloc = 1$
          SETFALSE
      else if $lastzcount = 2$
        SETFALSE
    else if $zcount = 1$
      if $lastzcount = 0$
        SETFALSE
      else if $lastzcount = 1$
        if $lastdloc = 0$
          SETFALSE
        else if $lastdloc = 1$
          SETTRUE
      else if $lastzcount = 2$
        SETTRUE
    else if $zcount = 2$
      if $lastzcount = 0$
        SETTRUE
      else if $lastzcount = 1$
        if $lastdloc = 0$
          SETTRUE
        else if $lastdloc = 1$
          SETFALSE
      else if $lastzcount = 2$
        SETFALSE
    else if $dloc = 0$
      INC1
    else if $dloc = 1$
      INC1
  else
    TERMINATE

Figure B.4: Binary Adder Decision Tree
<table>
<thead>
<tr>
<th>source1[i]</th>
<th>source1[i-1]</th>
<th>source2[i]</th>
<th>source2[i-1]</th>
<th>dest[i-1]</th>
<th>set dest[i] to</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table B.8: Summary of Binary Addition Possibilities
Bibliography


