Using Learning to Improve the Computational Performance of the Simulation of a Modeled Environment

Warren Armstrong

A subthesis submitted in partial fulfillment of the degree of
Bachelor of Science (Honours) at
The Department of Computer Science
Australian National University

October 2005

Typeset in Palatino by TEX and LATEX 2ε.

Except where otherwise indicated, this thesis is my own original work.

Warren Armstrong
30 October 2005

To my grandparents: Desmond, Basil, Doreen and Flower.

Acknowledgements

Several people have helped me over the course of this year, and I would like to use this space to thank them. In no particular order:

Thanks to Eric McCreath for supervising me, for his advice throughout the year, for reading and commenting on drafts of this thesis, and for generally being a lighthouse on the shores of academe.

Thanks to Weifa Liang for always having time to answer questions, and for inspirational words.

Thanks to my fellow students, with whom I have had many hours of interesting conversation - specifically: Jo Curtis, Michael Stevens, Nic Jean, Thomas Sutton, Josh Milthorpe, Adam Wagg, Eleanor Donovan, Lei Pu, Cameron Lamont, James Ring and Song Yang Guo.

Thanks to the denizens of the Discworld Multi User Dungeon, who have provided much laughter and learning.

Thanks to staff at Hancock Library, for maintaining a wealth of reference material.

Lastly, and most importantly, thanks to my family for far too much to say here.

Abstract

Decision trees are a common way to solve classification problems. Methods for learning such trees using information-theoretic heuristics are well known. However, existing learning methods do not consider the cost of evaluating the learned decision tree.

This thesis describes a learning algorithm which incorporates a new decision procedure for selecting nodes in a decision tree. This procedure imbues established information-gain methods with an awareness of the consequences of their choices in the terms of evaluation speed of the learned tree. The new algorithm chooses attributes for the tree based on a measure of their *bit-rate*, or how fast they produce information.

An empirical comparison was conducted between the new, speed-aware method, and a standard one based purely on information theory. It was found that significantly different trees are generated by the two methods. Evaluation of these trees was conducted in terms of two metrics: precision and speed. The new method produced trees which were significantly faster than those produced by the standard method - the gain ranged between 3% and 24%, depending on the number of objects in the world being classified.

Additionally, while the standard method could not outperform the baseline speed of the function it was approximating, the new method recorded such an improvement in speed that it was faster than executing the untutored function.

These gains in speed did not result in a decrease in accuracy over the traditional method.

Contents

List of Figures

List of Tables

Introduction

Consider a young zebra, living wild on the African savannah. From birth, this zebra must learn to recognise aspects of the environment, and to classify what is food, what is a threat, and so on. While classifications which are as accurate as possible are obviously beneficial, the time to arrive at the classification is also crucial. Spending too much time in deciding to run away from the lioness is not a viable survival strategy.

To take a more computing oriented example, imagine a simulation that needs to make many classifications. The time taken to run the simulation is proportional to the speed of the classification procedure. To make things more concrete, imagine a system which needs to check each possible pairing of 1000 objects. This needs 9.99×10^5 checks. If each check takes 10 milliseconds to calculate, the simulation needs a little under 3 hours to run. If each check only takes 5 milliseconds, the running time is cut in half. The algorithm is still $O(n^2)$, but runs much faster.

The main question, of course, is how to achieve such a speedup. One approach is to optimise the code for speed (for example, by using loop unrolling), and another is to change the algorithm. While the former could be done by humans, it is often entrusted to some extent to an optimising compiler. The latter is often done by humans, but it also can be somewhat mechanised.

The mechanisation can be performed by *machine learning*: given a specification of what should be achieved, a computer can sometimes come up with new ways to achieve it. The new ways may take advantage of patterns in the data which were not known to a human optimiser. This may be a drawback if these patterns are artifacts of a poorly chosen training set, but will be beneficial if the patterns are characteristic of the domain.

Just finding a new way isn't always useful, though, as it could be slower than the original. This thesis describes the design of an algorithm to learn a function guided by timing constraints. This function will be evaluated against an algorithm learned without reference to said constraints. The goal of this thesis is to show that utilising timing constraints results in faster evaluation without a loss of accuracy.

The remainder of this work is organised as follows: Chapter 2 provides background information. Chapter 3 describes the system used for the learning experiments. Chapters 4, 5 and 6 describe the new methodology and report on its performance. Chapter 7 draws together conclusions reached throughout the thesis, and gives directions for future research.

Background and Previous Work

This chapter states relevant previous work, presenting the background knowledge needed to understand the thesis. It also makes the contribution of this thesis explicit.

2.1 Background

2.1.1 Learning from Examples

Machine learning is the discipline of Artificial Intelligence which deals with programs that learn behaviours. There are three main branches of this field - supervised learning, reinforcement learning and statistical learning. Supervised learning involves the learner interacting with a tutor, which guides the learning process though the use of examples, or lessons. Reinforcement learning occurs when the learning agent has no feedback on its performance other than a single numerical score, which it acts to maximise. Statistical learning treats the world as uncertain, and attempts to learn the appropriate probability distributions to deal with it.

The learning algorithms used herein utilise examples provided and annotated by an external agent, and so are applications of supervised learning.

2.1.2 Collision Detection

Given that the algorithm described here performs supervised learning, it requires both a source of examples and a way to classify them. This requires that the learner be associated with a domain. For this thesis, the domain is that of collision detection between moving circular objects in a two dimensional frictionless world. This section sketches the derivation of the equation used to perform this detection - the same equation which classifies examples and assesses the accuracy of the learner. The equation is simple, further information on collisions can be found in most undergraduate physics texts (for example, Tipler's series [Tipler 1999]).

Consider the motion of such a system, which can be visualised as a pool table with two-dimensional balls - the frictionless surface means spin or curve on the balls is not an issue. The motion consists of a series of collisions, in between which balls travel in straight lines. This straight line travel is characterised by five values, two for direction,

two for position and one for size:

Let	x_1	denote the position of object 1 along an axis labelled x,
	y_1	denote the position of object 1 along an axis orthogonal to x labelled y,
	dx_1	denote the speed of object 1 in the x direction,
	dy_1	denote the speed of object 1 in the y direction,
	r_1	denote the radius of object 1,
	$x_2, y_2, dx_2, dy_2, r_2$	denote the same quantities as above for object 2

At some point in time t, the distance between object 1 and object 2 is given by:

$$\begin{aligned} d(t) &= \sqrt{\delta_x^2 + \delta_y^2} - (r_1 + r_2) \\ \text{Where} & \\ \delta_x &= (x_1 + dx_1 t) - (x_2 + dx_2 t) \\ \delta_y &= (y_1 + dy_1 t) - (y_2 + dy_2 t) \end{aligned}$$

A collision will occur when this distance drops to zero. To determine when this will be, it is necessary to solve:

$$0 = \sqrt{\delta_x^2 + \delta_y^2} - (r_1 + r_2)$$

which can be transformed into solving

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

Where

$$\begin{aligned} A &= (dx_1 - dx_2)^2 + (dy_1 - dy_2)^2 \\ B &= 2\big((dx_1 - dx_2)(x_1 - x_2) + (dy_1 - dy_2)(y_1 - y_2)\big) \\ C &= (x_1 - x_2)^2 + (y_1 - y_2)^2 - (r_1 + r_2)^2 \end{aligned}$$

2.1.3 Decision Trees

Decision trees are used for classification. They are analogous to a computer program written as a sequence of `if-then-else` statements. An example decision tree is presented in Figure 2.1, based on the earlier analogy to a grazing zebra. This tree is used to decide whether a situation merits running away, or if it's safe to continue

grazing. The leaf nodes contain the two possible classifications. To use the tree, the test in the root node is evaluated in the context of a situation. Depending on the result of the evaluation, one of two branches is taken. If the affirmative branch is taken, then the classification is complete. If the negative branch is taken, then another test is run, and depending on the result of this, the situation can be classified.

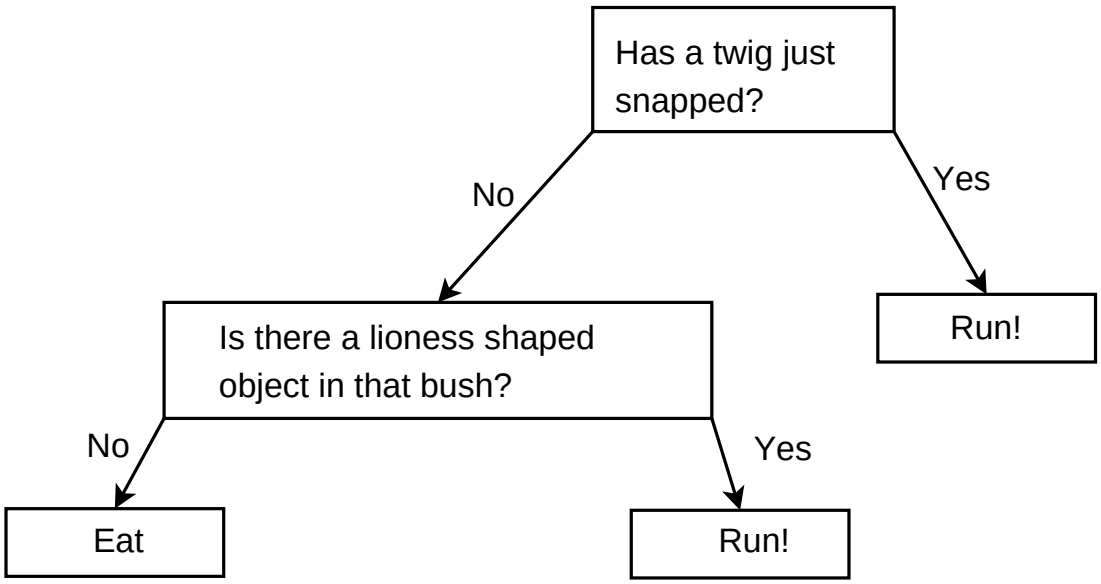

Figure 2.1: Example Decision Tree for Zebra Survival

The example decision tree was simple. There are several possible complications, two of which are noise and multivalued attributes. Noise is a problem which is not just restricted to supervised machine learning, but in that context it refers to examples which have been incorrectly classified by the assisting agent. To deal with such a situation, the learner must first of all be aware of it, and then may use statistical methods to ignore certain examples which are likely mistakenly classified. Multivalued attributes mostly cause problems because of their potential to bias construction in certain ways - for example, a test with eight possible outcomes may be selected before a test with two, even if the test with two outcomes will produce a more accurate tree. Countering this bias involves changing the methods used to build the tree.

There are various metrics for ranking decision trees. The most common one is the accuracy of the tree. This is measured on data that were not used in building the tree (known as the *test set*), and is frequently quoted as a confidence interval. A confidence interval is a measure of how representative the test set was of unseen data in general. For example, an accuracy of 90% with a 95% confidence interval of 2% is a statement that there is a 95% chance that any unseen data will be classified with an accuracy between 88% and 92% inclusive. Further details on confidence intervals can be found in Section 3.6.1.2 and in Mitchell [Mitchell 1997].

Another metric is speed of evaluation. This was used in Quinlan [Quinlan 1983] to measure the speed of decision trees for predicting chess end games. Both the accuracy

and speed metrics are used to evaluate trees produced in this thesis.

2.1.4 Information Theory

The material presented here is based on the introduction given in Aczél and Darócyz's book [Aczel and Daroczy 1975]. It presents the terms and concepts of information theory in as much detail as is needed to understand their use in this thesis.

Information is defined on an event e, which occurs with a certain probability $P(e)$. Consider the event to be linked to measurement of the state of some system - say, the average daily maximum temperature for Canberra in October 2005. Imagine a student equipped with a list of daily maxima, and who knows the mean temperature. Picking a random example from the list, there is a probability that it is close to the mean. If this is indeed the case, then the value is expected, it yields little information beyond what was already known. If, however, the value is far away from the mean, then it could not have been guessed at using only data previously available. Hence it has a greater information yield. A scientific corollary is that, if fifty experiments support a theory, more information is yielded if the fifty-first experiment disproves the theory than if it provides another confirmation.

Information, then, is a measure of the unexpected. There are some other properties which it is usual to assume should hold for some measure of information. If $I(P(e))$ denotes the information gained from an event e with probability P, these properties can be stated as:

1. $I(P(e)) \geq 0$,

2. $I(P(e_1)P(e_2)) = I(P(e_1)) + I(P(e_2))$

.

The first property says that information gained does not reduce what is known, and the second says that the information gained from knowing two events occurred is the same as knowing each individual event occurred - this obviously requires the events to be independent. These two requirements are satisfied by using the logarithm function:

$$I(P(e)) = -log_2 P(e), \qquad P(e) \in (0, 1]$$

The negation is needed as probabilities are less than or equal to one, and hence have negative logarithms. The base of two allows an interpretation of information as the number of binary digits required to encode the unexpectedness of the event.

Lastly, information is often referred to as *entropy*. An event is high in entropy if there is little data available on it. This parallels the usual use of the word as a measure of disorder - a highly disordered system could do anything, while a very ordered system (for example, the one reading these words) is constrained to behave in certain ways, *e.g.* not to suddenly float through the ceiling.

2.2 Previous Work

2.2.1 Selecting Attributes

Exactly which attributes are used to build a decision tree has been the focus of much research over the years. An early approach by Hunt *et al* [Hunt et al. 1966] documented the performance of nine variants of a Concept Learning System, or CLS. Each variant used different attribute evaluation methods or operated under different memory constraints. The methods used were:

CLS 1-5 : Choose the attribute which appears most often in positive examples,

CLS 6-8 : Maximise the difference between an attribute's positive and negative occurrences,

CLS 9 : Selects the attribute which does the best job of partitioning examples into classes.

While they note that the cost of using the tree is a factor, none of their learning systems consider this when selecting attributes.

Hunt *et al* also mentioned information theory as a possible selection method, but did not use it due to computational effort required. Quinlan introduced this method with his system `ID3`, described in [Quinlan 1987]. `ID3` uses the information function from Section 2.1.4, extended to handle two events rather that one. This is achieved by weighting the information from each event's probability by the probability that the event occurs, so an unlikely event, whilst very informative, is rare. `ID3`'s information function is thus

$$I(p,n) = \frac{-p}{p+n} - log_2\frac{p}{p+n} - \frac{n}{p+n} - log_2\frac{n}{p+n}$$

where p and n denote the number of examples belonging to two classes P and N. The above formula is applied to each possible value of each candidate attribute to determine how much more information is needed to complete the classification. Summing over all the possible values the attribute can take on gives the average amount of information required to finish classifying after the attribute is tested:

$$E(A) = \sum_{i=1}^{v} \quad \frac{p_i + n_i}{p+n} I(p_i, n_i)$$

where there are v different values of attribute A, and there are p_i and n_i elements of classes P and N for the ith attribute. The best attribute to test is thus the one which minimises E, that is, the one which leaves the least amount of information still needed.

This approach to attribute selection has proved popular and is used in many other sources, such as Russell and Norvig's text [Russel and Norvig 2003]. Other methods have also been developed, including the χ^2 measure which examines associations between variables, the G statistic, which is related to both the χ^2 measure and the

information theoretic measure, the GINI index, which measures the mix of classes under different values of an attribute, and the Gain-ratio measure, which modifies the above information-gain measure by considering the spread of examples over attribute values. These measures are all discussed in Mingers [Mingers 1989].

2.2.2 Measuring Tree Performance

Traditionally, decision trees with equal error rates are assessed by their size.. The main justification for this is that a simpler tree is more likely to generalise. In 1989, Mingers evaluated five attribute selection techniques over four different domains, and found that not only was the accuracy achieved invariant over the gain ratio, χ^2, G or GINI methods, but that selecting attributes randomly achieved the same levels of accuracy. Mingers concluded that the primary difference between methods was the size of the produced trees. [Mingers 1989]

2.3 Contribution

The contribution of this thesis is the notion that *speed matters*. It presents the design and experimental evaluation of an algorithm which constructs decision trees in a non-standard way. Instead of trying to minimise decision tree depth, the algorithm used assumes different nodes can execute at different speeds, and tries to minimise the average computation cost for classifying unseen objects.

System

This section describes the software which was used to gather the results for this thesis.

3.1 Introduction

As noted previously, the experiments reported in this thesis are carried out in the domain of collision detection in a two dimensional world. The software consists of several key components, namely:

- situation generation,
- naive collision detection,
- learned collision detection,
- learning algorithms,
- accuracy evaluation and
- speed evaluation.

Each component is described in detail below. The overall structure of the software is irrelevant, but enough information is given to enable reproduction of the results. Some of the explanations make use of pseudocode to outline the algorithms used, this pseudocode is written in a fairly common syntax, details of which are available in Appendix A.

3.2 Situation Generation

Central to supervised learning is the provision of examples, which are situations annotated with the correct classification. For this thesis, the situations used were generated randomly on request, and then annotated by a separate procedure. This allows the training set of examples and the test set of unannotated situations to be generated in the same place.

Each example consisted of a specified number of identical moving objects enclosed within four walls. The dimensions of the enclosed area were specified in a configuration file, for all the experiments herein, the arena was 50 by 50 square units. This space was subdivided into a square grid, each cell of which had a side length equal in magnitude to the diameter of a moving object. Each object was then placed at a vertex of the grid, which was specified semi-randomly. The semi- prefix comes from a desire to optimise the running time of the placement code. Several attempts were made to place the object on an unoccupied random vertex, but for high collider densities, many vertices would already be occupied, leading to many random positions needing to be tried and discarded. To alleviate the problem, once a certain number of random placement attempts had failed, the code would perform a methodical sweep through available points, and place the moving object appropriately. The example generation algorithm is made explicit below.

```
generate_dataset(){
    for each row, column in grid
        set grid[row][column] as unoccupied

    until the specified number of colliders has been placed do
        count := 0
        do
            generate a random position (x,y)
            count := count + 1
        while grid[x][y] is occupied and count < 10

        if grid[x][y] is occupied then
        for every row in the grid
                for every column in the grid
                    if grid[row][column] is unoccupied then
                        free_x := row
                        free_y := column
                        goto place_it

place_it:
        if free_x and free_y have not been set
            return an error
        grid[free_x][free_y] := occupied
        m := a new moving object
        set the centre of m at the point free_x, free_y
        set the x velocity of m to a random value between -50.0 and 50.0
        units per second
        set the y velocity of m to a random value between -50.0 and 50.0
        units per second
        add the collider to the set of colliders

    add four walls to the set of colliders
    return the set of colliders
}
```

3.3 Naive Collision Detection

The situations produced by the algorithm in the previous section need to be classified if they are to form examples for learning. In addition, situations used to test the learned hypotheses need to be evaluated, so the accuracy of the hypothesis can be judged. These two functions should be performed by the same piece of code, as it is the output of that code that is being learned.

The pseudocode for the example classification routine is shown below. Its purpose is to examine two objects and decide if they will collide or not. [1]

```
detect_collision( impacter ){
    dvx := my_x_speed - _x_speed_of_impacter
    dvy := my_y_speed - _y_speed_of_impacter
    dx := my_x_coord  - _x_coord_of_impacter
    dy := my_y_coord  - _y_coord_of_impacter
    sigma := (my_radius + _radius_of_impacter)*(my_radius + _radius_of_impacter)

    A = dvx*dvx + dvy*dvy
    B = 2*(dx*dvx + dy*dvy)
    C = dx*dx + dy*dy - sigma

    if ( B*B < 4*A*C )
        return false

    if ( A == 0.0)
        return no collision

    root = Math.sqrt( B*B - 4*A*C )
    sol1 = (-1*B + root ) / (2*A)
    sol2 = (-1*B - root ) / (2*A)

    if ( sol1 <= 0 && sol2 <= 0 )
        return no collision

    return collision happens
}
```

3.4 Learned Collision Detection

A learned hypothesis for collision detection should have as low an overhead as possible, in order that comparisons between it and the naive method are as fair as possible. While it would be possible to also evaluate the accuracy in this mode, optimising for speed requires all debug output to be removed, and some indirection to be used. The decision was therefore taken to generate two forms of the hypothesis, one optimised

[1] The astute reader will have noticed that this code will not correspond to real-world physics under certain rare boundry conditions. This is irrelevant since the thesis is evaluating performance in learning the function defined in this code, not the ability to learn accurate physics.

for speed, and the other easy to read. The latter is the simple tree walking algorithm shown below:

```
evaluate_hypothesis( root, situation ){
    if root is a leaf node then
        Return the classification of root

    if the test of root passes situation then
       Return evaluate_hypothesis( pass child of root, situation )
    else
        Return evaluate_hypothesis( fail child of root, situation )
}
```

This recursive tree walking is obviously not ideal from a speed point of view. While an iterative algorthim could be used, the multiple function invocations would still represent an overhead. For this reason, the hypothesis was compiled into a boolean expression using Java syntax, and this expression was wrapped in the appropriate class declarations and compiled. The only overhead that this method has over the naive algorthm is a single extra procedure call.

Compilation is performed by the routines shown below. The first, `compile_tree_to_java`, takes a hypothesis in the form of a tree, and uses the routine `compile_tree_to_string` to convert this tree into a logical expression. This expression is wrapped in just enough class declaration code to make it legal Java, and then compiled into a binary. An instance of the class representing the binary code is then returned, and this is used as a handle to invoke the collision detection expression.

The process of converting a tree into a string is handled by `compile_tree_to_string`, which makes use of `find_collision_path` to find structures in the tree known as collision paths. A collision path is a path from the root to a leaf with a classifiation of `collision will occur`. Any of these paths, if taken, is sufficient for a collision to be predicted, and at least one of them is necessary. Thus, collision paths are joined by `or` statements. Within a truth path, each attribute tested must take on a single value, either true or false. Therefore, the attributes within a path are interleaved with `and` statements.

```
compile_tree_to_java( root, default ){
    string_form := compile_tree_to_string( root );

    if string_form is empty then
        string_form := default

    Wrap string_form in a class declaration
    Write the declaration to a file
    Compile the file
    Return a new instance of the object representing the compiled code
}

compile_tree_to_string( root ){
        path_endpoint := find_collision_path( root )
```

```
        if path_endpoint is null then
            return "false"

        hypothesis := ""
        while path_endpoint is not null do
            path := new array, equal in size to the depth of
                path_endpoint in the tree
            current_node := path_endpoint
            current_node_index := last valid index of the array path

            while current_node has a valid parent do
                path[current_node_index] := the parent of current_node
                current_node := the parent of current_node
                current_node_index := current_node_index - 1

            if there are any elements of path then
                string_form := ""
                for every element in path except the last do
                    string_form := string_form +
                        the test of the node at this element +
                        logical_and

                string_form := string_form +
                     the test of the last node in path

                string_form := "(" + string_form + ")"
                if this is not the first thing added to hypothesis
                    hypothesis := hypothesis + logical_or
                hypothesis := hypothesis + string_form
            path_endpoint := find_collision_path( root )

    return hypothesis
}


find_collision_path( node ){
    if node is a leaf and node hasn't been marked as used then
        mark node as used
        return node

    if node is a leaf or node has been used before
        if ( n.has_type || n.parsed )
            return null

    tmp := find_collision_path( first_child_of_node )
    if ( tmp is not null )
        return tmp

    return find_collision_path( second_child_of_node )
}
```

3.5 Learning Algorithms

Central to this thesis are the algorithms used for learning decision trees. There are two such algorithms: one which uses pure information gain to guide learning, and one which modulates information gain with speed of evaluation. Both make use of a common framework, which is presented below:

```
Learn( Attributes, Examples ){
    if there are no useful attributes remaining
        guess a classification C
        create a node N with classification C
        return N

    A := select_best_attribute( Attributes, Examples )
    create a node N which tests A
    pass_examples := all elements of Examples which cause A to
        evaluate to True
    fail_examples := all elements of Examples which cause A to
        evaluate to False
    N->pass_child = Learn( Attributes - A, pass_examples )
    N->fail_child = Learn( Attributes - A, fail_examples )
    return N
}
```

There are three factors in the accuracy attained by the algorithm. Firstly, there is the manner in which the best attribute is determined - that is, the implementation of `select_best_attribute`. This is the manner in which the various algorithms differ, and is covered in depth in later chapters. The second factor is the attributes themselves, because what they measure determines what can be learned. All learners discussed later draw on the same set attributes, outlined in a table in Appendix B.

Lastly, the number and nature of the examples determines how accurately a concept can be pinned down. This is one of the limitations of the algorithm - it is not designed to be accurate when given only one class of example. Such algorithms do exist, but the framework used herein assumes a examples from both classes are provided. If they are not, then it will simply class everything as the class which it has seen, because in its universe, that is the only class there is. Additionally, the algorithm can only work in situations which are qualitatively similar to ones it has been trained on. To clarify, if two situations are qualitatively similar, then they cause the attributes to take on the same values. If too few examples have been seen, then it is likely that the whole gamut of configurations has not been seen by the learner, leading to inaccurate predictions when shown a test set.

3.6 Evaluation

As discussed earlier, the domain of the learning algorithms is collision detection, within a simulation of moving objects. The basic unit of such a simulation is a `step`. A step consists of detecting the earliest collision, moving all objects forward in time

by the appropriate amount, and then changing the velocity vectors of the colliding objects. The basic code is outlined below:

```
step() {
    for every object_one in the test set
        for every object_two in the test set
            if object_one and object_two are not the same then
                prediction := detect_collision( object_one, object_two )
                if prediction is positive then
                    time := calculate_intercept_time(object_one, object_two)
                    if time is less than min_time or
                min_time has not been set
                then
                        min_time := time
                        collider_one := object_one
                        collider_two := object two

    if no collisions take place then
        return

    if min_time is not right now then
        for every object in the test set
           move object forward in time by min_time

    change_vectors( agent_one, agent_two )
}
```

To be used in judging accuracy, the outline above needs to be augmented by code to record the desired outcome of a check, and to compare this to the actual outcome. This gives rise to two flavours of the `step` template, one which records accuracy and one which checks accuracy. In addition, versions of the routine are needed to record and compare speed. In total, there are four flavours of routine implemented:

- `step_hybrid_accuracy`,
- `step_pure_accuracy`,
- `step_hybrid_speed` and
- `step_pure_speed`.

The `hybrid` and `pure` in the names refers to the detection method used by the routine. Hybrid routines use methods which have been learned, whilst pure routines solve the quadratic equation presented earlier.

3.6.1 Assessing Accuracy

3.6.1.1 Measurement

The important aspects of the accuracy measuring adjustments to the `step` template are shown below. The excerpts are not complete, the missing code is identical to that in the template.

```
step_pure_accuracy( step_number ){
    create pure_predictions[step_number] as a new array of booleans
    index := 0
    for every object_one in the test set
        for every object_two in the test set
            if object_one and object_two are not the same then
                prediction :=
            detect_collision( object_one, object_two )
                pure_predictions[step_number][index] := prediction
                index := index + 1
                if prediction is positive then
< follow template >

step_hybrid_accuracy( int step_number ){
    create hybrid_predictions[step_number] as a new array of booleans
    index := 0
    for every object_one in the test set
        for every object_two in the test set
            if object_one and object_two are not the same then
                prediction := detect_collision_hybrid( object_one, object_two )
                hybrid_predictions[step_number][index] := prediction
                index := index + 1
                if prediction is positive then
< follow template >
```

At each step, two boolean arrays are created - `pure_predictions` and `hybrid_predictions`. The result of each check made is stored in the appropriate array. The number of mispredictions can then be calculated by counting the number of places at which the arrays differ.

3.6.1.2 Error

The accuracy assessment above gives a score on how many test situations were predicted correctly. The central question, then, is how representative performance on the single set of test data is of performance on classification tasks. This is measured by using the statistical notion of confidence intervals, as described in chapter five of Mitchell's book, [Mitchell 1997]. In brief, the true error T is calculated from the sample error S as

$$T(z) \in \left[S \pm \zeta(z) \sqrt{\frac{S(1-S)}{n}} \right]$$

where n is the size of the test set, and z denotes the required confidence interval. This is the required probability that the true error rate will fall within the range indicated. ζ is a scaling constant which depends on z. All error results in this thesis are presented as 99% confidence intervals.

A second issue is exactly what the error score represents. There are two types of error score, termed *perfect* or *fast* error, and *safe* error. The former requires exact accuracy, collisions may not be predicted as misses, and misses may not be predicted

as collisions. The latter allows misses to be predicted as collisions, as the subsequent check of collision time will reveal the error.

3.6.2 Assessing Speed

3.6.2.1 Measurement

To measure speed, all the bookkeeping code in the above section is stripped out. This leaves two routines which are essentially the same as the `template`, except that the hybrid routine includes code that handles collisions with walls differently - this is for a variety of reasons, mainly because the implementation doesn't allow the assigning of a single coordinate to a wall. Therefore the attributes used can not be evaluated in that situation. The pseudocode for this modified routine is as follows:

```
step_hybrid_speed(){
    for every object_one in the test set
        for every object_two in the test set
            if object_one and object_two are not the same then
                if neither object_one nor object_two are walls then
                    prediction := detect_collision_hybrid(object_one, object_two)
                else
                    prediction := detect_collision( object_one, object_two )
                if prediction is positive then
< follow template >
```

Timing measurements are taken with the following code template, where `step_routine` is a placeholder for either `step_hybrid_speed` or `step_pure_speed`.

```
    Pristine := deep_copy( test_data )
    start_timer()
    for a number of runs
        for a number of iterations
            step_routine()
        test_data := deep_copy( Pristine )
    stop_timer()
```

As mentioned before, the step routine involves the detection and handling of a single collision. To form an entire simulation involves linking together a number of steps. That is the purpose of the inner loop in the above pseudocode. The purpose of the outer loop is to minimise the effects of error on the result. Each simulation is performed several times, and the total time elapsed is measured. This total is then divided by the number of steps performed to yield an average step time. Obviously for this average to be relevent, each simulation needs to start with the same data, which is the effect of the `Pristine` variable and the `deep_copy` operation. The variation within an iteration provides the hybrid hypotheses with a variety of situations in which to perform.

3.6.2.2 Error

There are two sources of error encountered when measuring time. First of all, there is the resolution of the timer to take into account. The timing data presented later

were gathered under a Debian Linux system, using the high resolution profiling timer available under some Java runtimes with the package `sun.misc.perf`. This was empirically discovered to have a resolution of 3×10^{-6} seconds.

The second source of error comes from the fact that multiple invocations of the same routine on the same data do not necessarily take the same amount of time to run. This could be the result of a number of factors, ranging from a sensitivity to the context of the caches to other processes running under a multitasking system. To minimise the latter effects, the timing data were gathered by booting Debian Linux in single user mode. This cut down the number of active processes (for example, there is no X server running) - the complete list of what was running is given in Appendix C.

3.7 Phases and Repeats

Two concepts which will be used later and need to be introduced are phases and repeats. A phase is a single simulation, it consists of the following steps.

```
perform_simulation(){
    train_learner
    measure_accuracy
    measure_speed
}
```

The details of the `train_learner` step are the focus of later chapters of this thesis, while the other two steps have been explained above. Each phase learns using the mistakes of previous phases. To guard again statistical flukes, each series of phases is repeated several times, enabling an average to be taken. Each repeat is called a `run`. The overall structure of the code is:

```
run( pupil, title ){
    for a number of repeats, do
        for a number of phases, do
            perform_simulation()
}
```

Speed Results

This chapter presents results comparing the speed of evaluation of hypotheses generated by the two learning methods. These results show that the new method produces a decision tree which is always faster than the tree produced by the standard method. For medium to large problem sizes, the trees from the new method are also faster than using the untutored method - the standard method produces trees which never outperform naive methods.

4.1 Methodology

All results for this chapter were gathered by using five phase learning, with five independent trials for each problem size. The graphs below represent an arithmetic mean of all the measurements which were predicted with a hypothesis learned from both positive and negative examples. The error bars show the standard error in this mean, calculated as:

$$\text{error} = \frac{\text{standard deviation}}{\sqrt{\text{number of measurements}}}$$

All the graphs plot various quantities against collider density. This quantity is related to the problem size (number of moving objects) as follows:

$$\text{collider density} = \frac{\text{number of moving colliders}}{\text{size of arena}}$$

4.2 Baseline

The baseline speed is the speed at which all collision checks are done without the benefit of learning. Figure 4.1 displays how the baseline speed varies with the collider density. The figure plots the time taken to perform a simulation step, in milliseconds, against the collider density. Two lines are shown, representing the speedup when this baseline was used to analyse the entropic learner, and when it was used to measure the rate learner. These two lines agree to within standard error, which is expected.

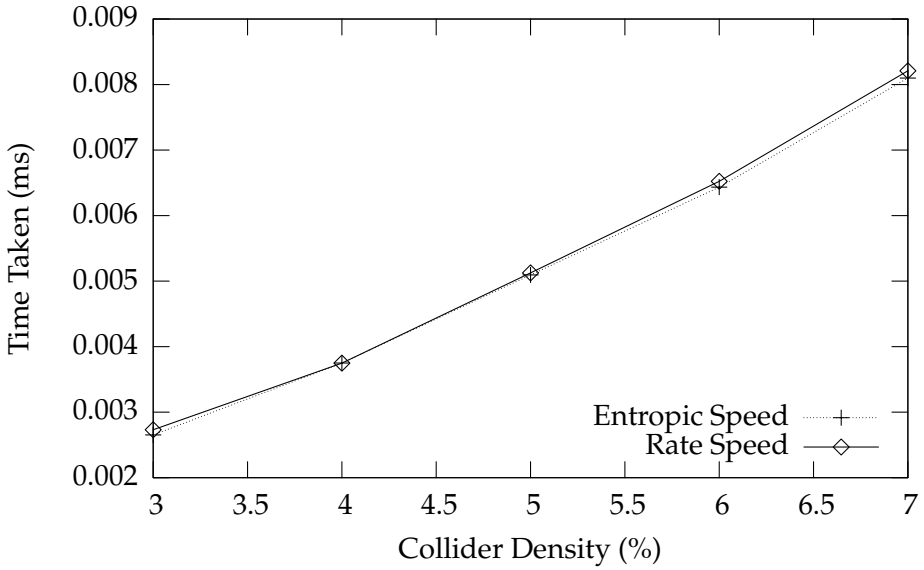

Figure 4.1: Baseline speed with standard optimisations

4.3 Entropic Learning

The speed of the traditional information gain algorithm is shown in Figure 4.2. The horizontal axis of the graph shows the number of moving objects for which collision detection must be performed, and the vertical axis shows the ratio of the speed of the learned predictions to that of the baseline. The graph shows that at all points measured, the performance of this algorithm was worse than that of the untutored system.

4.4 Rate Learning

The speed results from using the new method are shown in Figure 4.3. As before, the horizontal axis of the graph shows the number of moving objects involved, and the vertical axis shows the ratio of the speed of the learner divided by the speed of the non-predictive equation solving algorithm.

In this case, the performance of the learned hypotheses is poorer than the untutored method for small problems. As the problem size increases, however, the learning method comes into its own, and achieves performances of up to 15% over the baseline.

4.5 Comparison

For ease of comparison, the two plots are joined in figure 4.4. The first thing to note about the graphs is that the rate-based method is always faster than the traditional entropic method. This shows that considering the speed of the variables does produce

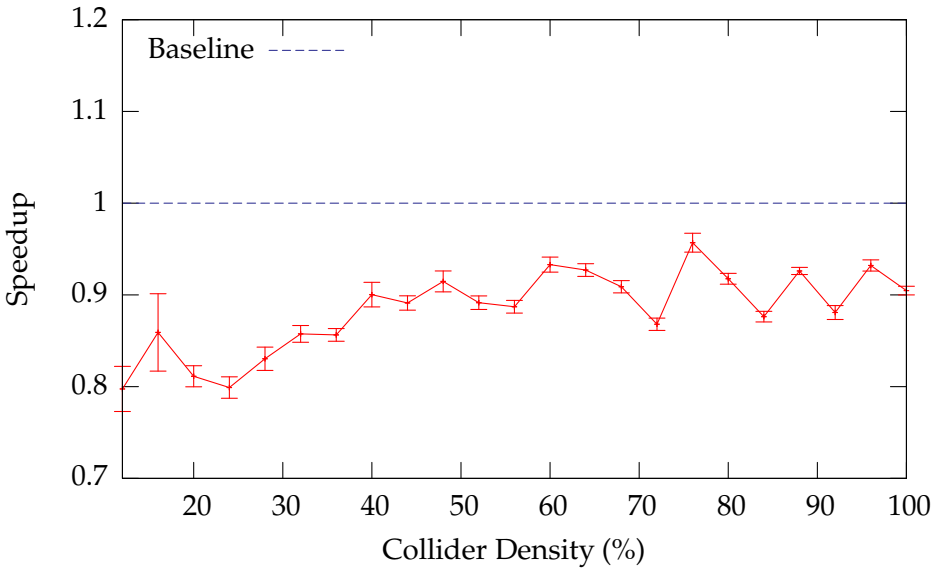

Figure 4.2: Entropic Learner, speedup with safe prediction

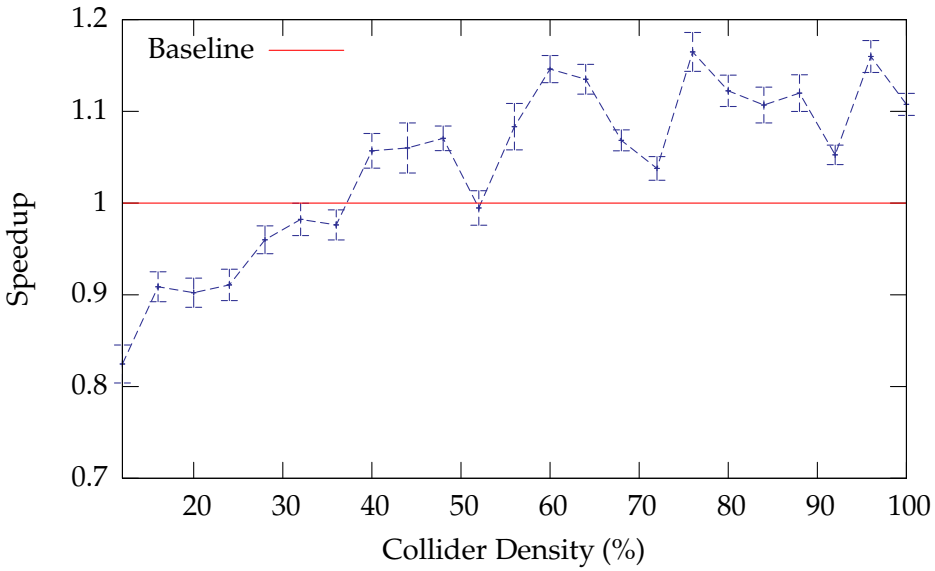

Figure 4.3: Rate Speeds

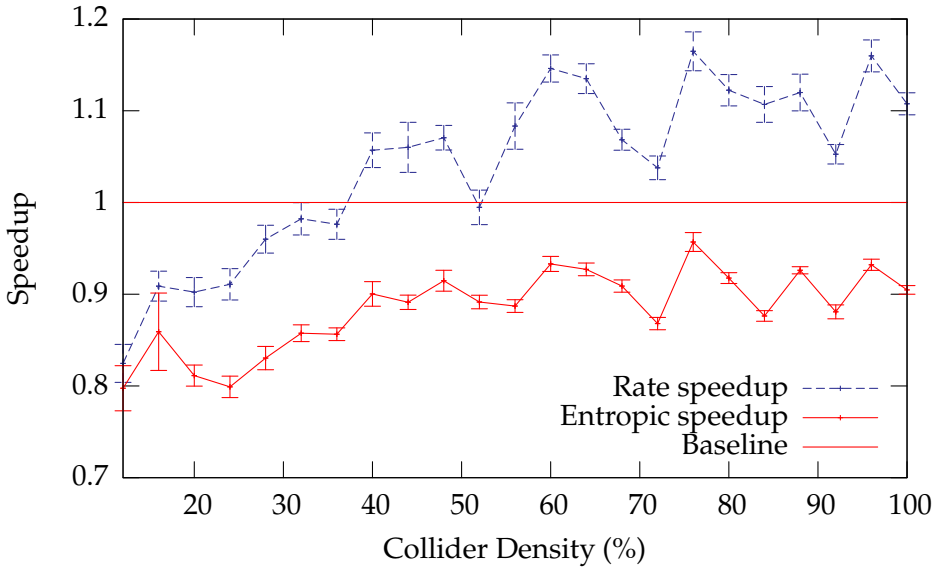

Figure 4.4: Baseline Speeds, normal optimisation

a measurable difference in evaluation time. The magnitude of this difference is shown in Figure 4.5. [2] The improvement ranges from just over 3% up to just over 24%.

The second aspect of the graphs is that they are really quite similar. This is highlighted by Table 4.1, which classifies the points on each graph as either local minima, local maxima, or neither. It can be seen that there are only six points where the two graphs disagree - that is, 24% of the plotted points. Moreover, all of these points are at or below the problem size of 14 moving objects.

This allows a couple of inferences to be drawn. Firstly, it suggests that both algorithms are sensitive to the problem size, and they are generally react in the same way to the same problem size. This similarity is only qualitative, though. Figure 4.6 shows how the speed changes as the problem size increases. The vertical axis of this graph measures the ratio of the speed at successive collider densities. This graph shows that the entropic learner is very sensitive to changes in the problem size while that size is small, but it is much less perturbed once the problem size increases past six. The new learner, on the other hand, has a much lower maximum ratio, but displays its maximum ratio fairly regularly and more towards the larger problem sizes. Interestingly, the two appear to provide stability in different areas.

Why this sensitivity should be present is unknown, but there are two possibilities. Firstly, it could be an artifact of the Java runtime system. The speed measuring routine is invoked roughly one thousand times per run, which would certainly make it a good candidate for dynamic optimising compilation. To test whether such optimising compilation affected the timing results, tests were run in which the dynamic compilation was disabled. In addition, garbage collection was disabled. The results are shown in

[2] Errors are present in the graph, but are on the order of 1×10^{-3} or smaller, and hence are not visible on the graph's scale.

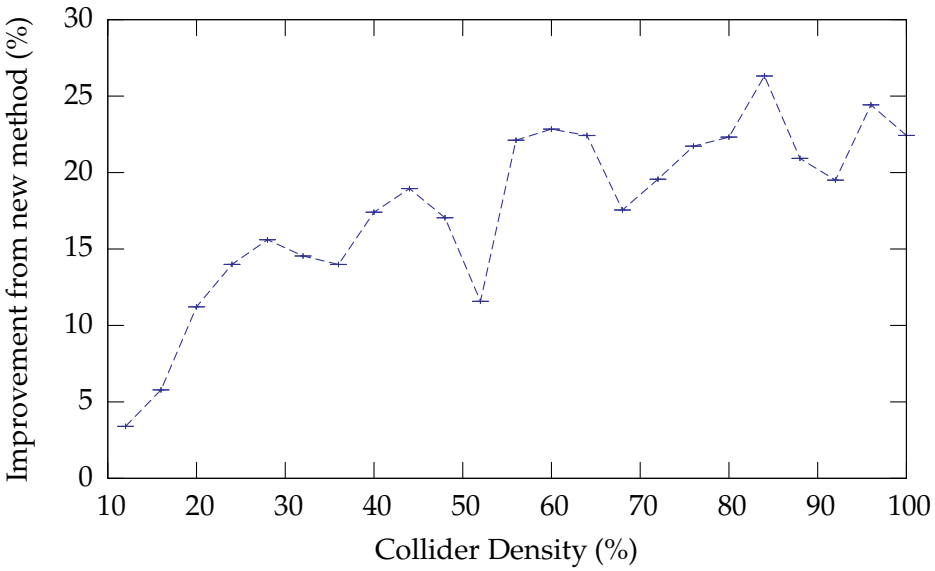

Figure 4.5: The magnitude of the improvement from the new method.

Table 4.1: Local Extrema Comparison of Figure 4.4

N	Entropic	Rate	N	Entropic	Rate
3	Min	Min	15	Max	Max
4	Max	Max	16	-	-
5	-	Min	17	-	-
6	Min	-	18	Min	Min
7	-	-	19	Max	Max
8	Max	Max	20	-	-
9	Min	Min	21	Min	Min
10	Max	-	22	Max	Max
11	Min	-	23	Min	Min
12	Max	Max	24	Max	Max
13	-	Min	25	Min	Min
14	Min	-	-	-	-

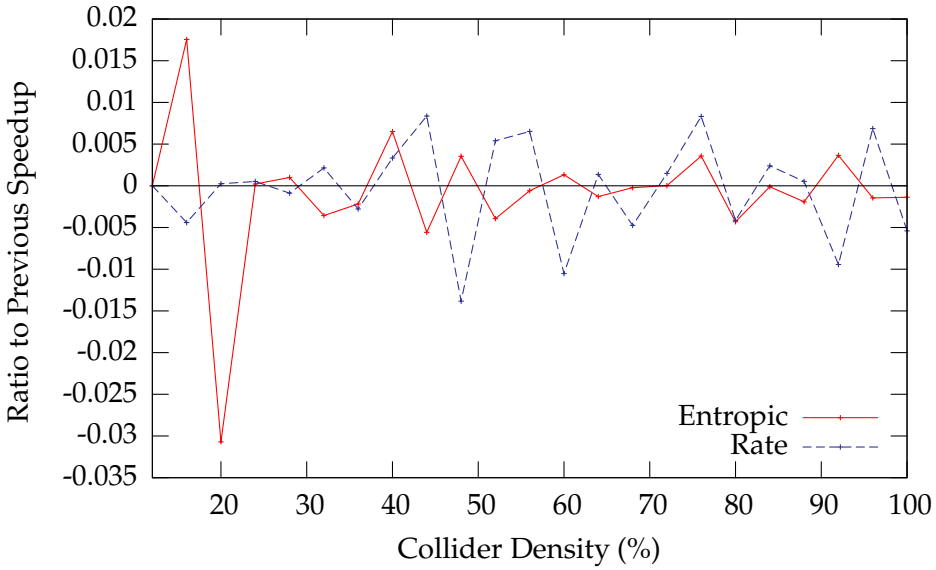

Figure 4.6: Ratio of speedups of successive problem sizes

Figure 4.7. The sensitivity to problem size remains.

The difference between the two learners is the hypotheses they generate. The rate-based learner generally induces the hypothesis shown in Figure 4.8, while the entropic learner induces the one shown in Figure 4.9. The structure of these hypotheses shows the reason for the difference in speed - although there are more nodes in Figure 4.9, each one involves less calculation. However, the presence of more nodes implies longer equations to calculate, as well as more branches to predict. These may be a factor, although without knowing the details of how the JVM used performs interpreted mode execution, it is not possible to be sure.

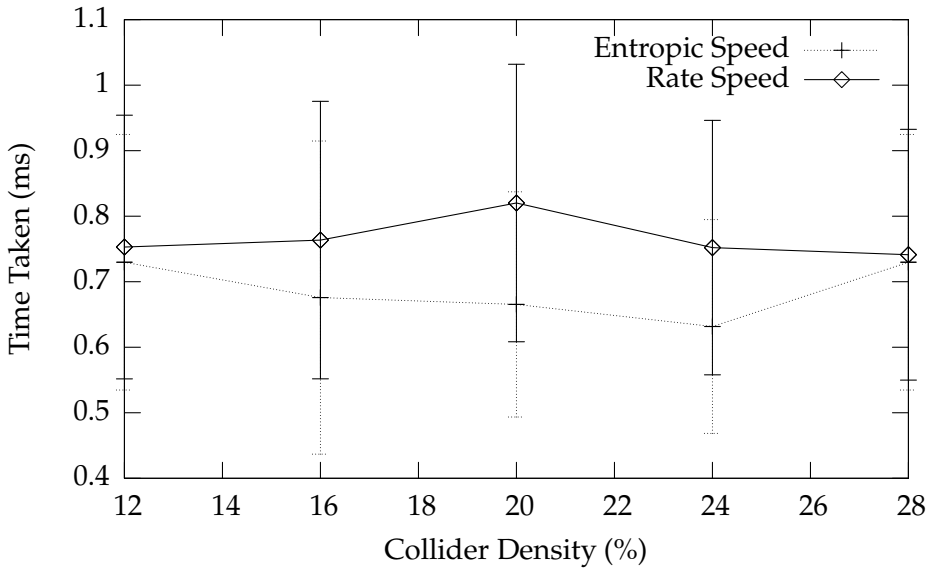

Figure 4.7: Speedup with no optimisation options

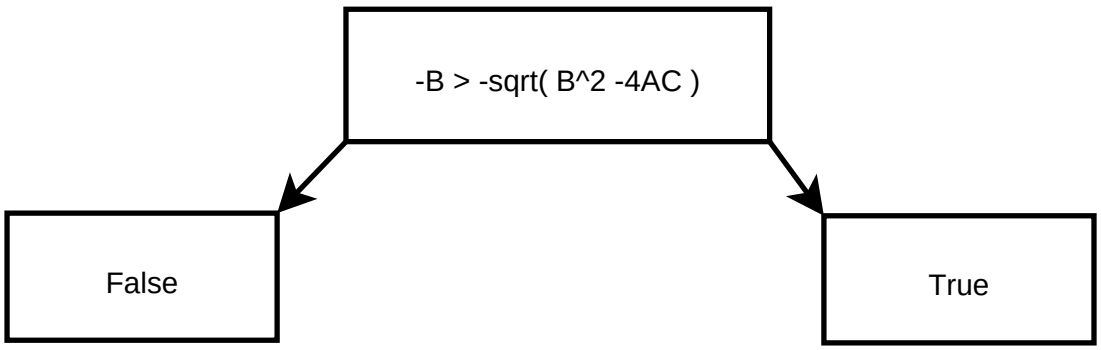

Figure 4.8: Typical Entropic Hypothesis

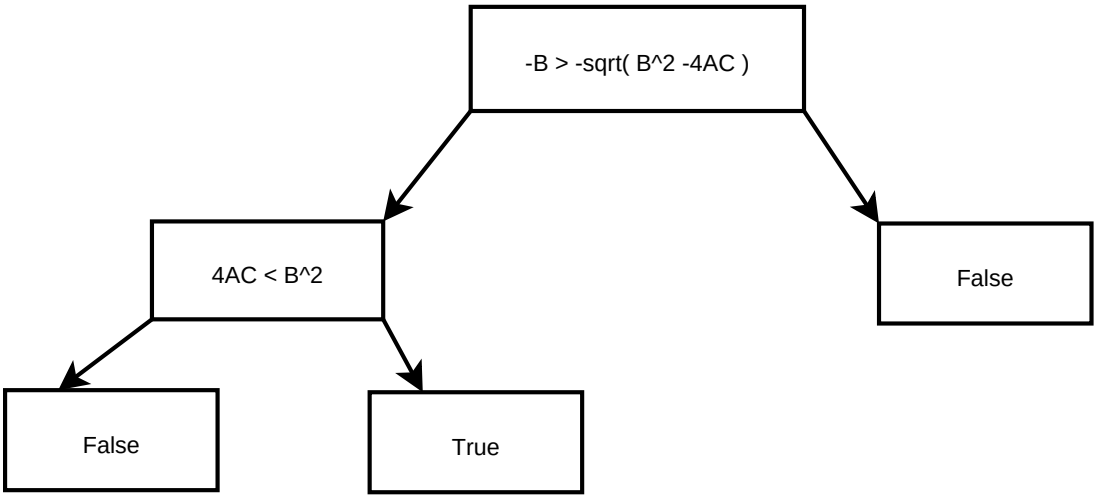

Figure 4.9: Typical Rate Hypothesis

Accuracy Results

Chapter 4 showed that the speed-aware learner is significantly faster than its entropic cousin. This chapter presents results to show that this gain is not paid for by any decline in accuracy.

5.1 Methodology

All results for this chapter were gathered by using five phase learning, with five independent trials for each problem size. The graphs below deal with error rates and confidence intervals, as explained in section 3.6.1.2. The error rate for a given collider density is the arithmetic mean of the all runs at that density which result from hypotheses generated by learners which have both positive and negative examples available to them, and the error bars in the plots denote 99% confidence intervals.

5.2 Entropic Learning

Figure 5.1 shows how the predictive accuracy of the purely information-based learning algorithm varies with the size of the input data. The problem size is plotted on the horizontal axis, and the vertical axis shows the mean error rate of hypotheses learned from both positive and negative examples. The error bars denote 99% confidence intervals. The plot shows both safe and perfect accuracy results.

As expected, the perfect accuracy requirement generates an error rate which is at least equal to the rate generated when only safe accuracy is required. Both lines trend towards zero in general, with only one data point breaking the trend - that at an input size of six. By an input size of ten, both methods make no errors, even under perfect accuracy requirements.

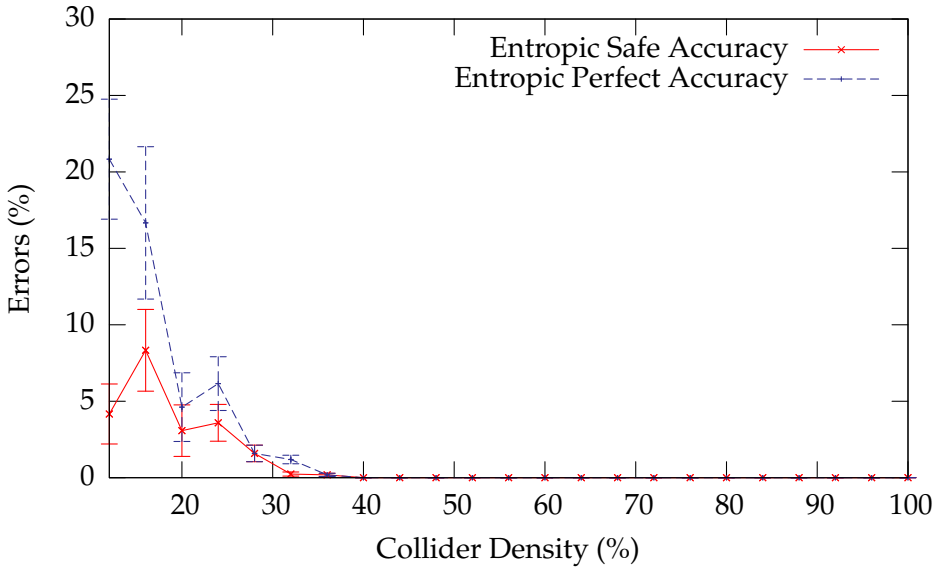

Figure 5.1: Comparison of Entropic Accuracies

5.3 Rate Learning

The diagram in figure 5.2 shows how the accuracy of the rate based learning method varies with input size. Again as expected, the perfect accuracy is no more accurate than the safe accuracy - in some places, it is significantly less accurate. Once the problem size reaches ten, both perfect and safe accuracies reach and remain at one hundred percent.

5.4 Comparison

Figures 5.3 and 5.4 plot the results of the two learners side by side for comparison. The following conclusions can be drawn from these plots. Firstly, in both cases, the new learning system is no worse off than the standard learner in terms of accuracy. According to these graphs, it is actually better, in some cases significantly so. However, this is an artifact of the way the learning algorithm, as described in section 3.5, handles ties. If two or more nodes are an equally good choice at a certain point, the algorithm selects the first one it sees. In the results reported here, the nodes are presented without regard for their possible utility. If the most useful nodes are presented first, then the two learners are equal in performance. This indicates that the new learning method is more robust than the standard technique, when the correct order to present the nodes

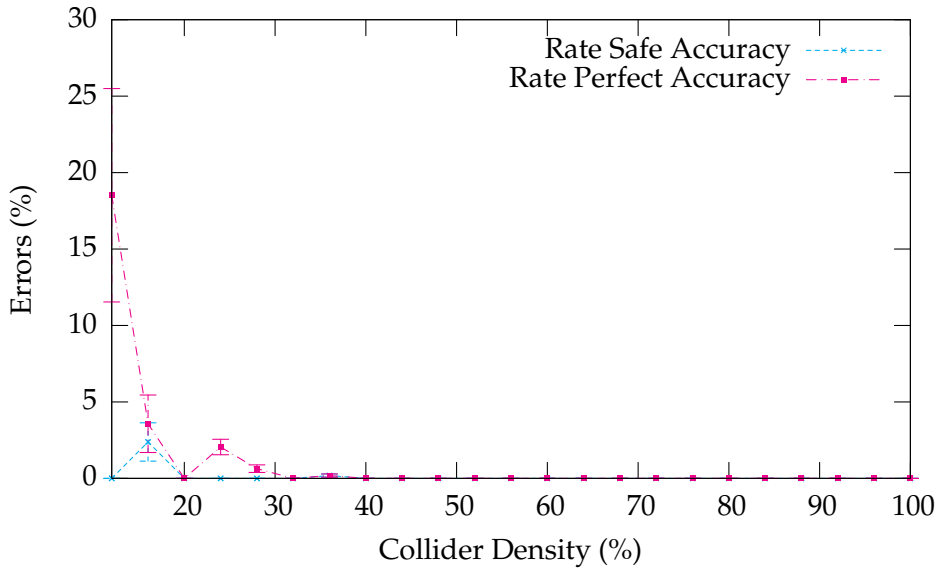

Figure 5.2: Comparison of Rate Accuracies

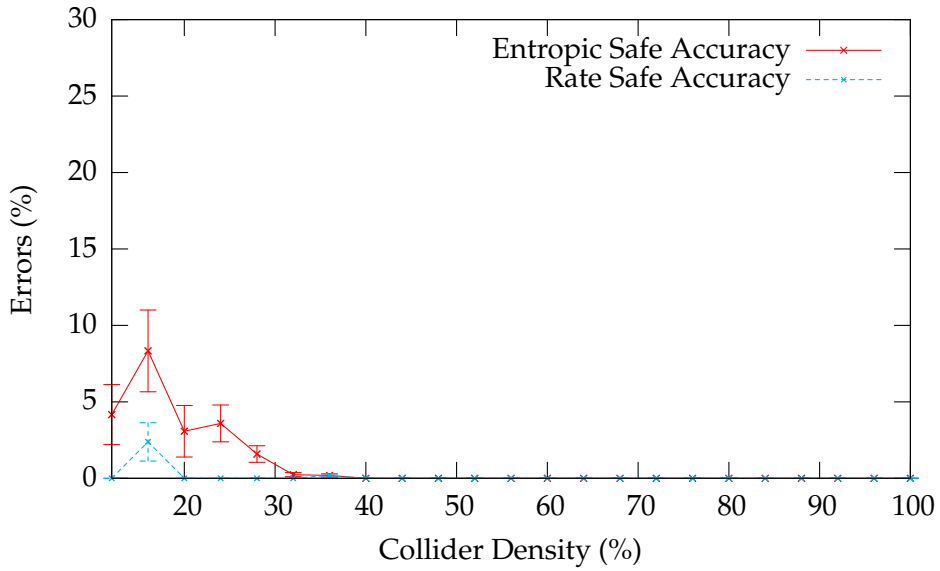

Figure 5.3: Comparison of Safe Accuracies

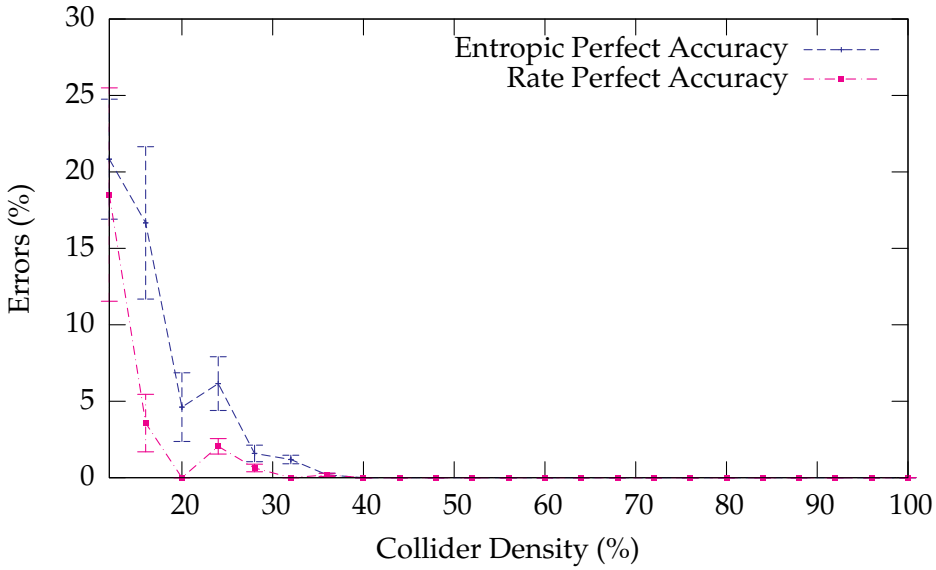

Figure 5.4: Comparison of Perfect Accuracies

in is unknown. However, this situation only holds for small problem sizes where ties are likely to occur.

The second observation involves the behaviour of both graphs - the points of low and high accuracy appear to be largely the same for both systems. This suggests that the variations in accuracy at low problem numbers are related to a factor common to the two algorithms - perhaps the relative density of positive and negative examples available to learn from.

It is instructive to compare the places where the general downward trend of the plots is broken. First, consider the problem size six. The standard method records an increase of 33% in the perfect error rate, while the safe error rate increases by 17%. This shows that roughly half the mispredictions are false positives, and the rest are false negatives. The rate learner, on the other hand, records an increase in the perfect error rate, but none in the safe error rate. This shows that the two hypotheses fail in different ways. In this case the new method will produce errors which affect only the speed of the simulation, not its integrity. This is not the case with the standard method.

Weighting

The rate-based learner discussed in chapters 4 and 5 employed a simple formula for scoring attributes - it took the raw information value and divided it by the raw speed. This section investigates the utility of weighting the scores - granting more weight to one of the two components. Results are presented in terms of how such weighting affects accuracy and speedup.

6.1 Algorithm

The algorithm in the time-aware learner was changed to calculate the score of an attribute α as:

$$Score(\alpha) = \frac{(1-\omega) * \text{information\_gain}}{\omega * \text{speed}}$$

where $\omega \in (0, 1)$ is a weighting.

6.2 Effect on Accuracy

Figure 6.1 shows how the extreme values of 90% and 10% weighting towards speed affects the accuracy attained. At low densities, there is quite a large gap between the two plots, and furthermore, the optimal weighting changes with every increase in problem size. As the density increases through 32%, though, both weightings attain perfect accuracy.

6.3 Effect on Speed

Figure 6.2 shows the effect of extreme weightings on the speed of the learner. It is interesting to note that giving more weight to the speed of an attribute doesn't necessarily result in a faster execution. This is because if there is enough information

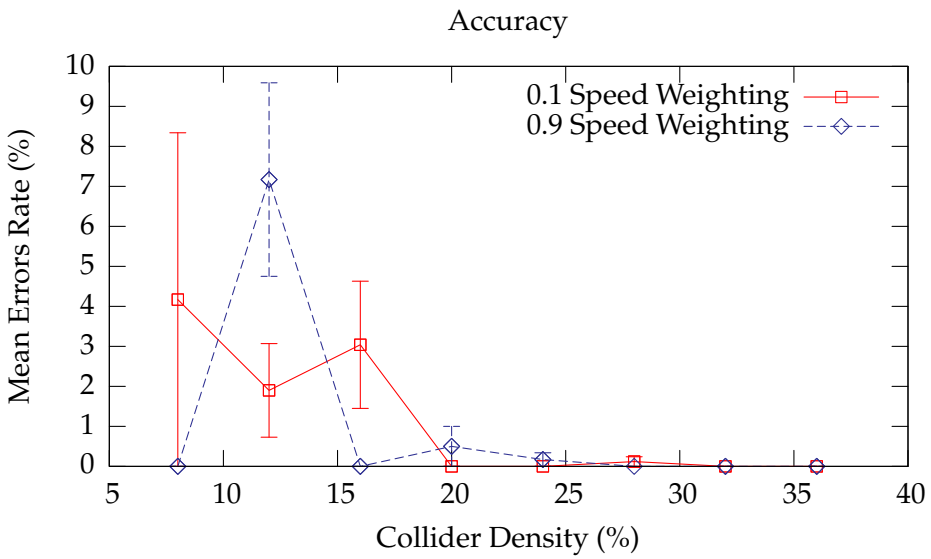

Figure 6.1: Mean errors vs Density for 0.9 and 0.1 speed weightings

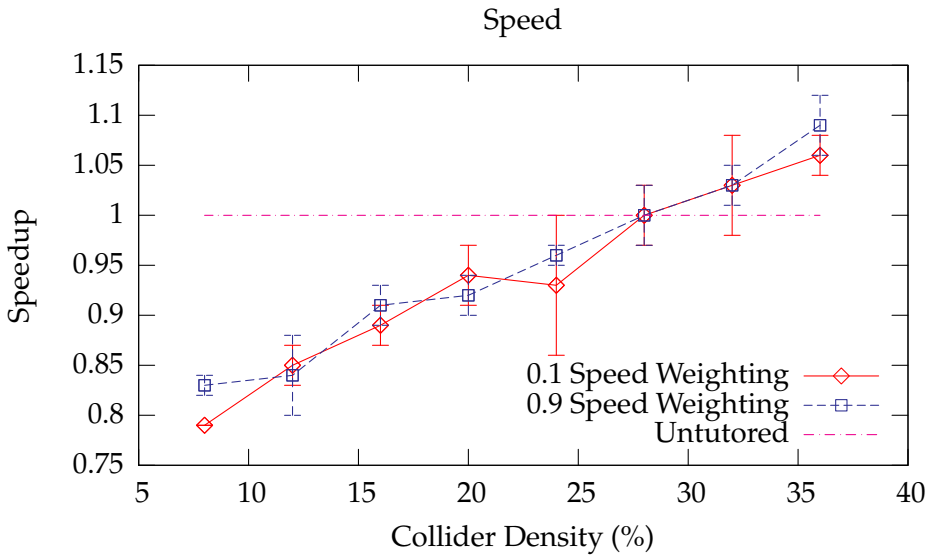

Figure 6.2: Mean errors vs Density for 0.9 and 0.1 speed weightings

available, allowing the information gain to have more influence reduces mispredictions, and as the measurements are on safe speedup, every false positive incurs an overhead. In such situations, granting more influence to speed than to weight allows the hypotheses to generate the wrong answer faster. Once the problem size reaches a density of 25%, though, granting a larger weight to speed results in a speedup which is no worse, and often better than, granting a larger weight to information gain. This is because from that point on there is enough information available to drop the error rate to zero.

Conclusion

The new method of attribute selection was shown to achieve a significant speedup over the benchmark method. Furthermore, this speedup comes at no cost to accuracy. It was also shown that extreme weightings in the new algorithm can enhance or degrade either metric, depending on the problem size.

Future work in this area could evaluate the new method against different domains. These domains could possess a greater variety of attribute speeds, or could require larger decision trees. Perhaps a general theory of applicability could be formulated.

Additionally, the new method could be extended to handle multiple attribute trees. It could also incorporate techniques for pruning and boosting, two methods which enhance accuracy in standard algorithms.

Thirdly, the method could be assessed against methods other than pure information gain, such as those presented earlier, *c.f* χ^2, gain-ratio, the G-statistic and GINI.

Appendix A

Pseudocode Syntax and Semantics

At various points through this thesis, code snippets are included to aid comprehension. They have been translated from their original Java syntax into a pseudocode, which should aid comprehension while hiding irrelevant implementation details. Some details of the pseudocode used are given below.

Assignment is denoted by the Eiffle-style :=,

Data Types are generally ignored,

Statements are written one per line, and semi-colons are not used,

Subroutines have signatures of the form

```
name ( argument 1, argument 2, ..., argument n )
```

Subroutine invocation is denoted by parentheses after a name, even for argument-less routines which are written `foo()`,

Loops and if-else statements are delimited by indentation,

Variables are case insensitive and not declared, anything to the left of an assignment operator is a variable.

Available Attributes

Table B below shows the attributes available to the learning algorithms. They are inequalites based around combinations of three quantities A, B and C, defined by:

$$\begin{aligned} A &= \delta\delta x^2 + \delta\delta y^2 \\ B &= 2(\delta\delta x \delta x + \delta\delta y \delta y) \\ C &= \delta x \delta x + \delta y \delta y - \sigma r^2 \end{aligned}$$

Where δx and δy denote the difference in horizontal and vertical position, repectively, of the two potentially colliding objects; $\delta\delta x$ and $\delta\delta y$ denote the difference in their horizontal and vertical speeds; δr denotes the x and y speeds; and σr denotes the sum of the radii of the potential colliders.

These are the quantities used to solve the collision equation exactly, and so should provide a space of variables rich enough to learn the concept. A central question in this thesis, however, is whether there are attributes which are faster to evaluate and still accurate enough.

0	$+ve_{\surd}$			$-ve_{\surd}$			4AC			-B			B^2			2A			0		
	$<$	$=$	$>$	$<$	$=$	$>$	$<$	$=$	$>$	$<$	$=$	$>$	$<$	$=$	$>$	$<$	$=$	$>$	$<$	$=$	$>$
$+ve_{\surd}$	-			-			X	X	X	X	X	X	X	X	X	X	X	X	-	X	-
$-ve_{\surd}$	-			-			X	X	X	X	X	X	X	X	X	X	X	X	-	X	-
4AC	-			-			-			X	X	X	X	X	X	X	X	X	X	X	X
-B	-			-			-			-			X	X	X	X	X	X	-	-	X
B^2	-			-			-			-			-			X	X	X	X	X	X
2A	-			-			-			-			-			-			-	X	-

Table B.1: Available Attributes

Environment

This appendix details the environment in which speed measurements were taken. The underlying hardware was the same for accuracy measurements, but the software environment was different. However, as the accuracy is a function of the algorithm and not the platform, software details are only provided about the speed measuring environment.

C.1 Hardware Platform

The experiments were run on a Dell laptop with a 3 GHz Intel Pentium 4 processor and 1 GB of RAM. Outputs of `cat /proc/cpuinfo` and `cat /proc/meminfo` are shown below.

```
CPU info:
processor : 0
vendor_id : GenuineIntel
cpu family : 15
model : 3
model name : Intel(R) Pentium(R) 4 CPU 3.00GHz
stepping : 4
cpu MHz : 2993.788
cache size : 1024 KB
fdiv_bug : no
hlt_bug : no
f00f_bug : no
coma_bug : no
fpu : yes
fpu_exception : yes
cpuid level : 5
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep
  mtrr pge mca cmov pat pse36 clflush dts acpi mmx
  fxsr sse sse2 ss ht tm pbe pni monitor ds_cpl cid
bogomips : 5931.00
```

```
Mem info:
MemTotal:       906736 kB
MemFree:        889552 kB
Buffers:          4828 kB
Cached:           5016 kB
SwapCached:          0 kB
Active:           8136 kB
Inactive:         2044 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:       906736 kB
LowFree:        889552 kB
SwapTotal:           0 kB
SwapFree:            0 kB
Dirty:              24 kB
Writeback:           0 kB
Mapped:           1588 kB
Slab:             3752 kB
Committed_AS:      520 kB
PageTables:         24 kB
VmallocTotal:   122800 kB
VmallocUsed:       620 kB
VmallocChunk:   121892 kB
```

C.2 Software Platform

C.2.1 Speed Measurements

Speed measurements were conducted while running Debian Linux, 2.6 kernel under single user mode. The processes active while the experiments were performed were obtained by use of the command `ps -A`, whose output was:

```
  PID TTY          TIME CMD
    1 ?        00:00:00 bash
    2 ?        00:00:00 ksoftirqd/0
    3 ?        00:00:00 events/0
    4 ?        00:00:00 khelper
    5 ?        00:00:00 kacpid
   40 ?        00:00:00 kblockd/0
   50 ?        00:00:00 pdflush
   51 ?        00:00:00 pdflush
   53 ?        00:00:00 aio/0
   52 ?        00:00:00 kswapd0
  195 ?        00:00:00 kseriod
  305 ?        00:00:00 kjournald
  323 ?        00:00:00 ps
```

The compiler was Blackdown's javac, version 1.4.2-02, and the VM was Blackdown's build 1.4.2-02 of the HotSpot Server VM.

Bibliography

ACZEL, J. AND DAROCZY, Z. 1975. *On Measures of Information and their Characterisation*. Academic Press. (p. 6)

HUNT, E. B., MARIN, J., AND STONE, P. J. 1966. *Experiments in Induction*. Academic Press. (p. 7)

MINGERS, J. 1989. An empirical comparison of selection measures for decision-tree induction. *Machine Learning 3*, 4, 319–342. (p. 8)

MITCHELL, T. M. 1997. *Machine Learning*. The McGraw Hill Companies, Inc. (pp. 5, 16)

QUINLAN, J. R. 1983. Learning efficient classification procedures and their application to chess end games. In R. S. MICHELSKI, J. G. CARBONELL, AND T. M. MITCHELL Eds., *Machine Learning, an Artificial Intelligence Approach*, Volume 2. Tioga Publishing Company. (p. 5)

QUINLAN, J. R. 1987. Induction of decision trees. *Machine Learning 1*, 81–106. (p. 7)

RUSSEL, S. AND NORVIG, P. 2003. *Artificial Intelligence: A Modern Approach*. Pearson Education Ltd. (p. 7)

TIPLER, P. A. 1999. *Physics for Scientists and Engineers*, Volume 1. W. H. Freeman and Company. (p. 3)