Project Number: DXF-SR61

Learning in Goal Oriented Autonomous Systems

A Major Qualifying Project Report:

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

By

_____

David Casavant

_____

Stuart Floyd

_____

Abraao Lourenco

Date: 3 March 2006

Approved:

_____

Professor David Finkel, Advisor

Abstract

This report, prepared for the Computer Science Laboratory at SRI International, covers the development of a Learning component for PAGODA, a modular architecture for goal directed agents that explore an environment without need for constant commands from an operator. Two learning algorithms are presented and our algorithm implementations analyzed. The report highlights the changes made to PAGODA and the development challenges encountered. Additionally, several potential improvements to the PAGODA system are given.

Acknowledgements

We would like to thank Andy Poggio, Dr. Carolyn Talcott, and Dr. Grit Denker for their guidance and support throughout the project. This project would not have been possible without their help.

We would also like to thank our project advisor at Worcester Polytechnic Institute, Prof. David Finkel, for his valuable advice and feedback, as well as for finding this great project for us.

Finally, we would like to thank everyone at SRI International and its Computer Science Lab for their support and for making our visit a memorable experience.

Table of Contents

Table of Figures

Executive Summary

PAGODA (Policy And Goal Based Distributed Autonomy) is an architecture for designing partially autonomous systems. It consists of one or more software agents that can sense and affect their environment and collaborate with one another to achieve their goals. A single PAGODA node consists of several components that work together to help the node achieve its goals.

In this project we extended the PAGODA architecture to include a Learner component. We implemented two learning algorithms and compared their performance by creating testing scenarios, running learning algorithms, and analyzing the results. We developed the Learner component so that a different Learner implementing a new learning algorithm could be substituted in easily, thus maintaining the modular design of PAGODA.

One of the two algorithms that we wrote is called a hill climbing algorithm. A hill climbing algorithm maintains the current optimal parameters of the environment in memory. The algorithm then tries to generate a new set of parameters randomly. If the new parameters yield better results, it keeps them. If the new parameters lead to worse results than the original, the algorithm tries another random set of parameters. This repeats continuously, allowing the algorithm to climb to an optimal scenario.

We also implemented an inductive logic programming (ILP) learning algorithm. This algorithm learns by analyzing the background knowledge, which consists of the relationship between parameters and effects under normal conditions, generating examples (i.e. new parameters) from the background knowledge, and recording the results (i.e. the effects). The results are analyzed and new parameters are generated

according to hypotheses relating parameters to effects, thus providing adaptation where the relationship between parameters and effects changes due to adverse or unexpected conditions.

In comparing the two algorithms we found that in general the hill climbing algorithm is able to find a better set of parameters when tweaked correctly. The ILP algorithm performed better than having no Learner under unexpected and adverse conditions, but it did not perform better than hill climbing algorithm.

There are many opportunities for extension and improvement of the PAGODA system that are just beyond our scope for this project. The work we have completed is only a small piece of a larger project. One example of improving the system is in the hardware simulation. In the PAGODA system, the parameters and sensors of the hardware abstraction layer are binary. One possible extension would be to have parameters and sensors with more values, or even a continuous range of values. Another possible extension would be to implement a way for the system to choose the learning algorithm that best fits the scenario on the fly. We implemented two algorithms that produce novel parameter settings in situations where the hardware doesn't work properly or there are adverse environmental factors. More algorithms could be plugged into PAGODA to determine how effective they are.

Section 1: Introduction

Learning is fundamental to intelligent life because it allows living beings to adapt to their environment. Giving a computer system the ability to learn allows it to adapt to new situations. Learning can improve efficiency and increase the chances of success when encountering even the most complex problems. The ability to remember previous experiences with similar situations is especially important to learning as it offers the opportunity to adapt based on previous actions.

Our project was to add a Learner component to SRI's PAGODA (Probabilistic And Goal Oriented Distributed Autonomy) system. The PAGODA system is composed of one or more agents, or nodes, which work together towards a mutual goal. A PAGODA node knows how to sense and affect the environment through the information made available in the device model. The device model determines which of the parameters the node can set as well as which parameters the node can read. By using sensors, the node is able to create a model of the environment. The node then uses this environmental model to determine the best way to achieve its goal(s). Each PAGODA node has several components: Headquarters, Reasoner, Monitor, Coordinator, Knowledge Base, and a Hardware Abstraction Layer.

The Learner is responsible for helping build better models as well as for developing strategies to create a more complete model of the environment through use of the decided upon learning algorithms. These models allow the agent to determine the best strategy for fulfilling its goals. There are several categories of algorithms within machine learning, each with its own strengths and weaknesses. We considered several types algorithms and chose to implement hill climbing and inductive logic programming. We then evaluated the performance of these algorithms by comparing the outputs with the expected results and directly with one another.

In the next section we present an overview of SRI International as well as more details on PAGODA and on Maude, the programming language used to implement PAGODA. In the third

section we present the hill climbing and the inductive logic programming and the details of their design and implementation. The results and conclusion, along with suggestions for future work follow at the end.

Section 2: Background

2.1 SRI

The Stanford Research Institute was created in 1946 by a small group of business executives in conjunction with Stanford University as a center to support economic development in the region following World War II [History]. Its mission was that of being "committed to discovery and to the application of science and technology for knowledge, commerce, prosperity, and peace" [History]. After purchasing all the land it occupied from Stanford University, the Stanford Research Institute became a separate entity called SRI International [History]. SRI International has grown to an organization with over 1,400 employees worldwide leading research in a diverse variety of technologies, from logic to polymers to imaging. [History]

SRI has spun off over 20 companies since its founding. In 2004, for example, SRI formed Artificial Muscle Incorporated to develop Electroactive Polymer Artificial Muscles. [Inventions] Another company called PacketHop Communication Systems developed Turemesh to work with autonomous mobile mesh networks. SRI has influenced a wide variety of innovations that are used in our everyday lives. Some notable examples of influence are the first magnetic ink reading system, the first computer mouse, the liquid crystal display, and speech recognition technology. SRI International has completed more than fifty thousand projects worldwide and continues to be a leading innovator in the world. [Inventions].

SRI has several R&D divisions, organized into each of its core competencies: Engineering and Systems Division, Policy Division, Information and Computing Sciences Division, Biosciences Division, and Physical Sciences Division [RD]. The Computer Science Laboratory is part of the Information and Computing Sciences Division. This division has three labs: Artificial Intelligence Center, Speech Technology and Research (STAR) Laboratory, Computer Science Laboratory [ICSD]. The Computer Science Laboratory studies "the logical foundations of scalable systems, that are beyond

the scope of traditional testing or simulation, and builds and applies efficient high-level tools for rigorous mechanical analysis" [CSL]. These systems included traditional hardware and software as well as biological systems [CSL].

2.2 Maude

2.2.1 Overview

Maude is a flexible, high level programming language and environment for development of programs using both equational and rewriting logic [TalcottTut]. According to Theodore McCombs "anything you can describe with human language, you can express in Maude", including natural numbers, biological systems, and even Maude itself [Primer].

A Maude system specification consists of an equational part describing system states and their properties in an abstract data type [TalcottTut], and rules specifying transitions between system states [Manual]. Abstract data types are specified in Maude using functional modules, which are a collection of data types and subtypes, operators, and equations; functional modules cannot contain rules [TalcottTut]. In mathematical terms, modules define algebras. An algebra is "a set of sets and operations on them", and modules consist of a set of sorts and operations on those sorts [Primer]. In the module SAMPLE-BOOL for example, we have the boolean values **true** and **false** and the operator **and** defining a boolean algebra.

```
fmod SAMPLE-BOOL is
       sort Bool .

       op true : -> Bool .
       op false : -> Bool .

       op _and_ : Bool Bool -> Bool .

       var myVar : Bool .
       eq true and myVar = myVar .
       eq false and myVar = false .
endfm
```

Figure 1: A sample functional module in Maude

2.2.2 Sorts

A sort is a category for values, similar to data types in other programming languages [Primer]. A sort can be any single value, or even a structure of multiple values. Specific groups of values belonging to the same sort are denoted by subsorts [Primer]. For example, "non-zero natural numbers" is subsort of natural numbers. Values of a certain sort do not share any inherent structure. Sorts are declared using the keyword **sort** and subsorts are declared using the keyword **subsort**. Multiple sorts are declared using the keywords **sorts** and multiple subsorts are declared using the keyword **subsorts**. It's important to note that before a subsort can be declared with the **subsort** keyword, they must be declared as sorts using the **sort** keyword.

As an example consider the Maude code section below in Figure 2, which declares the relationship between various quadrilateral shapes. Quadrilaterals are Polygons, so **Quadrilateral** is a subsort of **Polygon**. Similarly, **Parallelogram** (i.e. a quadrilateral where opposite sides are parallel) is

a subsort of **Quadrilateral** and **Rhombus** (i.e. a quadrilateral where all sides are equal) and **Rectangle** are both subsorts of **Parallelogram**. Since a **Square** is a special case of a both a **Rhombus** and a **Rectangle**, it is also a subsort of both. Note that although **Trapezoid** is at the same level as **Parallelogram** in the hierarchy, it is declared on separate line. This is because if it was declared alongside **Parallelogram**, it **Rhombus** and **Rectangle** would be treated as subsorts of **Trapezoid**, which is undesirable. These relationships are illustrated in Figure 2 below.

```
sorts Quadrilateral Polygon Trapezoid Parallelogram Square Rhombus Rectangle .
subsorts Square < Rhombus Rectangle < Parallelogram < Quadrilateral < Polygon .
subsort Trapezoid < Quadrilateral < Polygon .
```

Figure 2: Sorts and subsorts

This code creates the followship relationship between the sorts:

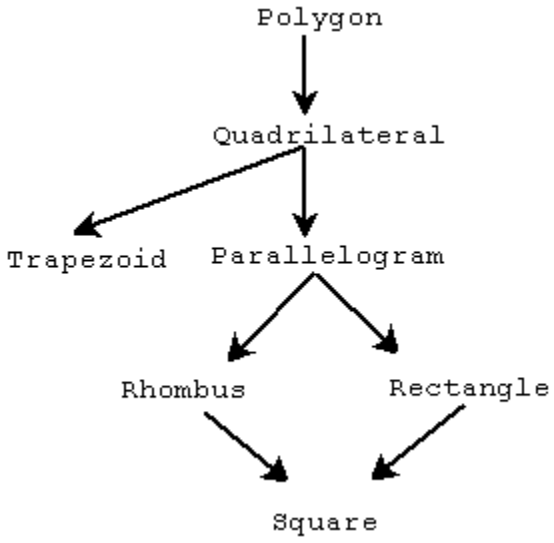


Figure 3: Relations between sorts and subsorts.

Converting between sorts is simply a matter of declaring that an operator takes in the source sort and produces the target sort; no manipulation of the data is required. The structure of a value is

determined by the equation that produces it, and not by its sort. For example, in the code below a Trapezoid is defined by the size of the two parallel sides and the size of its two equal sides, and a square is defined by the size of its sides. We can convert between those by using an operator that declares a Trapezoid to be a Square. This is clearly absurd. It is up to the programmer to maintain the structure of the data where necessary. The next section describes operators in more detail.

```
op Trapezoid : Nat Nat Nat -> Trapezoid .
op Square : Nat -> Square .
op convert : Trapezoid -> Square .
```

Figure 4: Conversions using operators.

2.2.3 Operators

Operators in Maude are declared using the **op** keyword with an operator identifier (i.e. letters, numbers, and many symbols are allowed) and optional placeholders (Maude uses underscores as placeholders) for arguments [TalcottTut]. These placeholders can be placed anywhere with regards to the operator identifier, allowing the programmer to use postfix, infix, or prefix notation at their discretion; this is called mixfix notation [TalcottTut].

In Figure 5 below, the operator is defined in prefix form in the first two lines. These first two lines show two details about operators. Parenthesis must be used around the operator if parentheses are already being used as part of the operator syntax. Operators can be overloaded by declaring them to take different sorts. In the third line the operator is declared in infix form. It is important to note that even though the operator in line three uses the plus sign, it is not overloading the operators defined in the first two lines, as the argument placing is different.

```
op (+(_,_)) : Nat Nat -> Nat .
op (+(_,_)) : NzNat NzNat -> NzNat .
op _+_ : Nat Nat -> Nat .
```

Figure 5: Operators and syntax.

There is also a special type of syntax called "empty syntax" where the operator identifier is omitted; in that case the arguments of the operator are separated by spaces [Manual]. In the example below we declare an operator with empty syntax that is used to concatenate two item lists into a single ordered list. Since an Item is a subsort of an ItemList, it can be supplied as an argument to the operator.

```
sort Item ItemList .
subsort Item < ItemList.
op __ : ItemList ItemList -> ItemList .
```

Figure 6: Operators and syntax with subsorts.

A simple use of operators is to declare **constants**. Constants in Maude are operators that take no input and produce a specific sort. The code below declares the constant **none** that can be used to represent an empty list [Manual]. Constants can be overloaded just like any other operators. Since this type of operator does not take input, in order to overload it we must declare it to return a different sort. Thus, if we had a sort called ItemSet, we could create an operator that represents an empty ItemSet with the same identifier **none**.

```
op none : -> ItemList .
op none : -> ItemSet .
```

Figure 7: Empty lists and sorts.

Additionally, binary operators can be declared to be either associative or commutative by using equational attributes [TalcottTut]. Going back to our ItemList and ItemSet examples, we need to make use of operator attributes to differentiate between the two, as well as to specify that the identifier **none** represents an empty set or an empty list as the case may be.

Let the concatenation operator have empty syntax (i.e. items are concatenated by being placed next to each other). In both a set and a list, x (y z) is the same as (x y) z, so the operator is associative in both cases; this indicated using the **assoc** operator attribute.

A list is ordered while a set isn't. In order to represent this difference we use the **comm** operator attribute in the concatenation operator that takes ItemSets. This means that **x y** is the same as **y x**.

Binary operators can also specify an identity term with the attribute **id** followed by the identifier of the identity operator. We specify the **none** operator as the identity operator for both the ItemSet and ItemList concatenation operator, since any set or list concatenate with an empty set or list is itself.

```
sort Item ItemList ItemSet .
subsort Item < ItemList ItemSet .

op __ : ItemList ItemList -> ItemList [ctor assoc id: none] .
op __ : ItemSet ItemSet -> ItemSet [ctor assoc comm id: none] .

op none : -> ItemList .
op none : -> ItemSet .
```

Figure 8: Lists and sets.

Finally, there is also a special type of operator called a **constructor** that specifies the basic terms of an algebra (or abstract data type) [TalcottTut]. A constructor is specified using the operator attribute **ctor**.

2.2.4 Variables

Variables in Maude are constrained in value to their declared sort [Manual]. They can be declared by using the keyword **var** or on the fly when accompanied by their sort. The scope of variables declared using **var** is the whole module and they have the effect of replacing occurrences of the variable with the variable name followed by its kind, in effect serving as a shortcut for the second method.

If the second method is used then its sort must follow every occurrence of the variable name.

```
vars A B C : Nat .
op ex1(A, B, C) = A + B + C .
op ex2(D:Nat, E:Nat) = D:Nat + E:Nat .
```

Figure 9: Variables.

2.2.5 Equations

Equations in Maude follow many of the same rules as mathematical equations and are used to specify how Maude expressions should be simplified [Primer]. Each equation corresponds to one operator, and an operator may have multiple equations associated with it. Constructors are the only expressions that Maude does not simplify [Manual]. The goal of a set of equations in a module is to provide rules for describing expressions using the constructors of an algebra—this is called the canonical form [Primer].

Equations in Maude are declared using the keyword **eq** followed a term that matches an op declaration, followed by the equal sign, followed by a term representing the body of the equation, and optional statement attributes [Manual]. The equation below, for example, specifies how to add a number to 0.

```
eq N + 0 = N .
```

Figure 10: Simple equations in Maude.

Maude also supports conditional equations through the use of the keyword **ceq**. Equational conditions can be regular comparisons using = , or matching equation using := . In matching equations the variable on the left hand side becomes instantiated to the value of the term on the right hand side.

```
ceq N + 0 = N if N == ... .
```

Figure 11: Conditional equations in Maude.

2.2.6 Rules

Rules are used to specify transitions between states. If the left hand side of the rule is matched against the current system state, the state is changed to what is specified on the right hand side of the rule. Similarly to equations, a conditional statement can also be added to the end of rules to make them conditional.

Maude works by pattern matching. That is, in order to determine which rule to apply, Maude matches the system state against the left hand side of the rule. If there are one or more matches, Maude applies a strategy when selecting which rule to execute. This strategy is specifiable by the user; e.g. the user can specify that the first match be executed, or that a match be selected fairly so that the same match isn't picked over and over again. Furthermore, execution in Maude is non-linear. While the current implementation of Maude is deterministic, it should not be assumed that statements are executed in the same order. A good analogy is to assume that every rule has its own thread that tries to execute at all times (constrained of course by matches).

2.2.7 Debugging in Maude

Tracing is the most basic of the debugging tools. It is invoked using the command **set trace on .** at the Maude prompt [Manual]. When activated, the trace prints out all rewrites performed by Maude while reducing an expression. For example: in Figure 12, tracing is used to see what rewrites Maude performs while simplifying a mathematical equation. Maude performs two rewrites while simplifying the expression, which are shown in Figure 12. For each rewrite, the output tells you where the starting expression, what components are being evaluated, and the resulting expression. The tracing feature in Maude is an easy debugging solution, but is inadequate for more complex programs due to the sheer output size. The PAGODA system can run for over an hour with the trace activated simply due to the significant increase in the amount of output required. The tracing feature has an option for only displaying specific rewrites; this feature is explained in more detail in the Maude manual [Manual].

```
Maude> set trace on .
Maude> red 4 + 5 * 3 .
reduce in CONVERSION : 4 + 3 * 5 .
*********** equation
(built-in equation for symbol _*_)
3 * 5
--->
15
*********** equation
(built-in equation for symbol _+_)
4 + 15
--->
19
rewrites: 2 in 10ms cpu (113ms real) (200 rewrites/second)
result NzNat: 19
```

Figure 12: Using Maude's tracing tool.

Another debugging tool in Maude will color the terms in a given output. This does not give any additional textual information, but it allows the programmer to differentiate types by looking at the text. The color of the text is dependent on the type and reduction level of the statement [Manual page 258]. The coloring command is activated by typing **set print color on .** (again, note the period at the end of the command). Colors are assigned based on the level of strangeness. Strangeness refers to terms that contain non-constructors [Manual]. "The idea is that red and magenta indicate the initial locus of a bug, while blue and cyan indicate secondary damage. Green denotes reduction pending and cannot appear in the final result." [Maude Manual]. By activating the coloring tool, tracing a bug in the program can become a much easier task. Like any system, however, the coloring does have its

limitations. Often times in complicated program runs, the coloring may just flag everything, thereby burdening the debugging process and providing little useful information.

The debug prompt provides a customizable debugging interface. It can be activate by using a Ctrl + C interrupt, hitting a breakpoint, or prefixing a command with the keyword debug [Manual]. Breakpoints can be added, removed, activated or deactivated from the Maude and the debug prompt. Breakpoints are created using the **break select** command [Manual pg 259]. Once the breakpoints are set, they can be activated by typing **set break on .** at either the Maude or the debug prompt. When the program is run, the execution will halt at the specified breakpoint(s). The program may then be controlled using the **where**, **step**, **resume** and **abort** commands [Manual].

Another helpful debugging technique is the selective usage of **rewrite** and **frewrite**. Rewrite is a top down evaluation strategy. The execution order of rewrite is somewhat predictable, as it will try to execute commands following the top down left to right approach of a search tree for the answer. This strategy can have drawbacks that result in problems such as an infinite loop [Manual pg86]. The frewrite, or fair rewrite command uses a "position-fair" strategy. This approach is designed to prevent starvation of a single rule when there is another rule that can still be rewritten [Manual section 15.2]. If there are commands that have the potential of looping because of multiple, infinite paths that the program could become stuck in, applying a fair rewrite will ensure that all potential solution paths are taken with an equal probability.

2.3 PAGODA Architecture

2.3.1 Initial Version

The initial version of PAGODA (Policy And GOal-based Distributed Autonomy) was a flexible and modular architecture for creating a partially autonomous agent capable of addressing a wide variety of problems in a virtual environment. This architecture was composed of six components, as shown in

figure 1. The PAGODA architecture was inspired by autonomous space systems [talcott], such as the Mission Data System architecture developed by the Jet Propulsion Laboratory at NASA [dvorak-ieee00] and its precursors [muscettola-etal-ai98].

The current SRI implementation of the PAGODA architecture supports an agent (also called a node) that works to achieve a given goal [email]. This node is able to sense its environment and in turn affect the environment according to its goals and constraining policies [email]. The node has six components: a knowledge base, a reasoner, a monitor, a hardware abstraction layer, headquarters, and a coordinator [talcott].

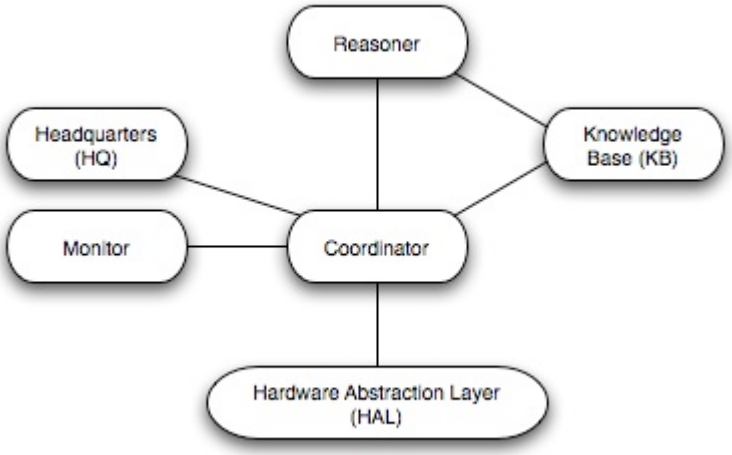


Figure 1: PAGODA node. [Talcott05]

The headquarters component communicates with an outside director that instructs the agent to modify its behavior by supplying new goals and policies. The hardware abstraction layer interacts with the environment in which the node is operating. This component changes the environment with its affectors and it collects data about the environment with its sensors. The coordinator controls the order and frequency of interactions between the headquarters and the hardware abstraction layer [Talcott05].

The knowledge base stores information about the goals that the node is trying to achieve, the

policies constraining its actions, and information about the current state of the node as well as about what is known about the environment. The headquarters component updates this information when the node receives new goals [Talcott05].

The reasoner and the monitor supervise the operation of the node. The reasoner assesses how changes in goals or unexpected parameter settings effect the operation of the node through use of initial and previously collected data from the knowledge base. The monitor keeps track of the operation of all other components in the node and notifies the reasoner when any unexpected conditions arise [Talcott05].

2.3.2 Functional Overview

Each component in PAGODA is initialized using an operator that produces a value of the Object sort. These objects are then concatenated together to form the overall state of the system. The overall value produced has the sort 'SState'. All rules in PAGODA are matched against this value.

When a component is initialized it contains an instance ID and a class ID. This combination uniquely identifies the component. The instance ID is the same for components in the same node, and the class ID is the same for components of the same type. Each component also has a set of attributes as well as incoming and outgoing message queues.

The components in a PAGODA node communicate by placing messages in their outgoing queue and reading messages from the incoming queue. Once a message is placed in the outgoing queue, the Switch is responsible for routing the message to the appropriate queue in another component. The Switch contains rules that govern the transport of messages across message queues among the different components. Message destinations and sources are identified by instance IDs and class IDs. Once the Switch matches the destination of an outgoing message to the appropriate destination, it removes the message from the outgoing queue in the source node and places it in the incoming queue in the target node.

Once every component is initialized, the Headquarters component tells the clocks to start ticking. Once the clock is running, it informs every interested component of each tick. When the headquarters component begins to receive ticks from the clock, it tells the Coordinator when to start a phase. A phase is a state of desired effects that the PAGODA system passes through. Each phase contains a collection of goals for the system to achieve. One of the sample phases in our implementation of PAGODA is called parking lot. This phase has certain attributes associated with it called effects that are important to the system in that particular phase. In the parking lot phase for example, these desired effects include high bandwidth and low error rate. This preference of effects does not mean that a certain effect that is not listed is not desired, but it allows the phase to have a priority. In other phases such as dispersed movement, the effects include minimal resources. This could be because in the dispersed movement phase, unlike the parking lot phase, resources would be greatly limited. In the implementation of PAGODA that we utilized, there were three phases that occurred sequentially: parking lot, dispersed movement and enemy engagement. These phases are completely arbitrary and are only used to keep certain desired effects. The coordinator specifies the interactions with the HAL and the monitor. The monitor keeps track of sensor status with regards to the goals and the HAL interacts with the virtual environment through its sensors and knobs.

Knobs are used by the PAGODA system to simulate interactions with the environment. As they are currently implemented, each knob in the system has a binary value of Hi or Low. When a knob is flipped, or toggled, the hardware abstraction layer simulates that change. The results are then read back into the system in the context of sensors. The sensors are, like to the knobs, binary. The knobs and sensors are used together by the PAGODA system to detect the simulated environment as well as interact with it.

MadRad is the radio simulator currently implemented in PAGODA. It is the hardware simulation with which the hardware abstraction layer interacts. MadRad has three modes of operation, any of which can be concurrently active at a time: normal, unusual, and faulty. In normal mode the

knobs have the intended effects and there is nothing anomalous about the environment. The normal mode is always active and other modes can be activated simultaneously. In unusual mode knobs don't have the intended effects (e.g. because of interference). In faulty mode, some sensors are stuck at a setting and knobs don't have any effect at all.

2.3.3 Data Structures

PAGODA

The PAGODA system is represented in Maude as an unordered set of Objects, where each Object is a component of the PAGODA system. The Object data structure has the following form:

```
[ o ( Cid, Inst) : Cid | Attrs | <- InQueue ; >- OutQueue ] .
```

Figure 13: PAGODA system object.

Cid: This is component id. Every component of this type has the same Cid. This is useful in a scenario where PAGODA has multiple node, each with its own set of components. In such a scenario, for example, all coordinators would have the same Cid.

Inst: Instance, or node id. All components inside a node share the same instance id. Again, this is only useful in a scenario where PAGODA has multiple nodes.

o (Cid, Inst): The combination of the component id and instance id for an object forms that object's object id, or Oid. Every object has a unique Oid.

Attrs: This is an unordered set of key-value pairs representing data stored in the Object. This data can be a component state (e.g. the component is in waiting mode) or a list of information (e.g. current knob settings).

InQueue: This is an ordered list representing the messages this object has received from other objects.

OutQueue: This is an ordered list representing the messages this object wants to send to other objects.

2.3.4 In Depth Component Overview

Clock

The Clock component in PAGODA represents a system clock. This clock is decoupled from real-time and represents the node's perception of time. When initialized the Clock is given a list of the components in the node that are interested in receiving clock ticks. Every time the clock ticks, a tick message is sent to each of those components.

The clock in Maude has two modes of operation: lockstep and free running. In lockstep mode the Clock waits for acknowledgement for each tick message that it sends out before sending another tick message. Thus some ticks may last longer than others when measured in real time. In free running mode the clock sends ticks messages to interested components as fast as Maude allows. This is a better simulation of a real clock as the real time between ticks is more equivalent.

Coordinator

The Coordinator is responsible for handling communication between different components. Unlike the Headquarters and the Knowledge Base, and similarly to the Clock and the Switch, the Coordinator doesn't hold any data, and its functionality is instead to coordinate communication between components using its rules.

The Coordinator has one rule for each message it handles. The Coordinator handles starting the appropriate phase, communication between the Knowledge Base and the Learner, communication

between HAL and the Learner, and communication between the Monitor and the Learner.

Headquarters

The Headquarters component is responsible for receiving orders from an operator in form of hard coded goals and for managing transitions between different phases. The Headquarters stores the number of nodes in the PAGODA network, the minimum time at which each phase should start, and the actual time at which the phase started. The Headquarters component is also responsible for controlling the Clock component.

The first task of the Headquarters component is to start the Clock component. Once started, the Clock component starts sending ticks to the interested components. One of these components is the Headquarters component itself. If a tick message from the Clock indicates that the time is greater than or equal to the minimum start time of a phase (or as is the case frequently, phases), the Headquarters component instructs the Coordinator to start that phase. The Headquarters then goes into waiting mode until the Coordinator sends a message stating that the phase is complete, at which time it returns to running mode. When in waiting mode, the Headquarters ignores tick messages from the Clock.

Knowledge Base

The knowledge base stores the following pieces of information: the current phase, the current situation, the effects for each phase, the knobs associated with each effect, the sensors associated with each effect, the default knob settings, and the initial sensor readings. Each one of these pieces of data is stored an attribute (i.e. key value pair), and together they are stored as an unordered set.

Other components such as the Learner query the Knowledge Base for information by sending it messages. For each message or query there is a rule in the Knowledge Base that matches it. Once a query is received in the Knowledge Base, some operators (depending on the query) are used to extract the information from the unordered set that is the Knowledge Base's store of information. The

extracted information is included in an outgoing message to the component that sent the original query. This process is repeated for each piece of information that can be queried in the Knowledge Base.

MadRad

MadRad is the hardware simulation used in PAGODA. It is a virtual radio that shares some characteristics with real radios. It can operate in different modes as well as in more than one mode at the same time. Its namesake comes from the fact that seven out of eight modes provide anomalous results [PAGODAdoc].

MadRad is initialized with the current mode, the current duration for that mode, the initial values of the knobs, and the probability for each mode. When the duration for the current mode expires, a new mode is picked based on the probability for each mode using a random number generator [PAGODAdoc].

MadRad has three mode classes: normal, unusual, and faulty. Normal mode produces the expected results when the radio works properly and there are no adverse environmental conditions. This mode is always active. The unusual mode class models adverse conditions in the environment. For example, there may be a lot of interference due to a lot of nodes. The current unusual modes are dense foliage, high concentration of nodes, noisy environment, and rapid movement [PAGODAdoc]. Unlike in normal mode, setting a knob to maximize an effect will make some sensors relevant to that effect read Con, and setting the knob to minimize the effect might make some sensors read Pro [PAGODAdoc]. Lastly, faulty mode classes represent the case where there is a permanent hardware or software fault in the radio. This faulty mode is simulated by having some number of sensors stuck at a value independent of the knob settings. Current faulty modes are bad battery, bad receiver, and bad transmitter [PAGODAdoc].

Switch

The Switch is the simplest of the components as well as the most important. The function of the Switch is to match a message in a component's outgoing queue with the incoming queue of the destination component, and to move the message from the outgoing queue to the incoming queue. This is what enables communication between different components. Furthermore, the Switch also maintains a history of the messages that it sees. This is frequently used for debugging.

Section 3: Learning Component

3.1 Algorithms

3.1.1 Hill Climbing

A hill climbing algorithm is a local search that aims to find the optimal solution through use of a heuristic function. The first thing the algorithm does is to pick an initial location and find the heuristic evaluation for that point. The algorithm then chooses a second point and finds the heuristic for that point. If the heuristic of the first point is better than the second, it keeps the first point and compares it against a new point; otherwise it drops the first point and keeps the second. In the end, the algorithm will eventually reach the best possible solution.

The goal of our hill climbing algorithm is to find novel knob settings that produce desired sensor readings. The local search approach of hill climbing is ideal for search space with a limited number of parameters that yield easily measurable results. Thus we have developed a hill climbing algorithm that randomly chooses several knob settings to change and tests the sensor readings that they generate against the original sensor reading produced by the old knob settings. There are a number of steps to this as are outlined in the following algorithm:

1. Once the Learner receives the message to start a phase, it obtains the goal effects associated with that phase from the Knowledge Base. The Learner then obtains the knob settings associated with these effects. The knob settings expected to satisfy the most sensors are used to initialize the hardware for the phase.
2. To initialize its state, the Learner queries the knowledge base for the *current knob settings*, which it stores as such, and queries the Reasoner for the *current sensor readings*, which it stores as such. If the current sensor readings satisfy all goals (i.e. all sensors are positive) then mark the respective *learning done* flag as true.

3. This algorithm relies on the learner receiving clock ticks. The clock ticks trigger a number of different actions depending on whether the learner is in-phase, the state of the *learning done* flag, and its state. If the learner is not in phase or the state is not ready then a reply is sent and ticks are ignored. If the *learning done* flag is true then the algorithm proceeds to step 7. If the state is ready and the *learning done* flag is false then a new hill climbing cycle, a new learning cycle, is started as is outlined in step 4.
4. This step represents a single learning cycle. The learner generates a set of *test knob settings* by randomly flipping the values of some of the items in the *current knob settings*. The *test knob settings* are sent to the hardware and the *test sensor readings* are retrieved. The *current sensor readings* and the *test sensor readings* are compared using the heuristic function, hereafter known as fitness function, as outlined in step 6. If the former receives a higher fitness score then the test knob settings are discarded, otherwise if the latter has a higher fitness score then the *test knob settings* are stored as the *current knob settings*.
5. If at any time the new *current sensor readings* satisfy all of the goals, the *learning done* flag is set to true, see step 7.
6. The fitness function takes in a set of sensor readings that are organized by the goals to which they are relevant and returns a value indicating the closeness of the sensor readings to satisfying all of the goals, higher being better. This function is implemented in such a way that it is easily modifiable should there be a desire to apply weights differently. Currently each goal has the potential to contribute 100 points to the fitness function. Of these 100 points only the percent of positive sensors verses the total number of sensors is actually added to the fitness function, thereby making the goal of the hill-climbing algorithm to maximize the number of goals that are close to having all of their sensors positive.
7. If the *learning done* flag is set and the learner gets a clock tick it will either ignore the clock tick or, every 15^{th} tick, it will get the current sensors to ensure that the situation has not changed and

all of its goals are still met. If the learner finds that the sensors have changed, it will set the learning done flag to false and the *current sensor readings* to the readings just obtained from the hardware. The learner will then start step 4 again.

Simulated Annealing

One of the disadvantages of a hill climbing approach is that it's possible to be stuck in a local maximum within an environment that contains several local maxima. Simulated annealing can be used to overcome this problem. Unlike hill climbing which uses changes of the same size, simulated annealing starts by making large changes in the parameters, and gradually reducing the rate of change over the course of the experiment.

We can modify the hill climbing algorithm to allow it to support simulated annealing by modifying the parameters deciding the frequency and magnitude of changes in the knob settings. These modifications were made in the following order:

1. We define a parameter *InitialPercent* at the top of the Learner code module that allows it to specify the starting value for the chance of a knob being flipped.
2. Another parameter C*hangeEachCycle* is added to the Learner, specifying the amount that the percentage should be decreased by each learning cycle.
3. An attribute set, *RandomProbability*, is added to the Learner's state to keep track of the current chance of a knob flip. At the start of a new phase *RandomProbability* is reset to *InitialPercent*.
4. During a phase (i.e. when knobs settings are being generated from the current knob settings), *RandomProbability* is used as the probability that any one knob will be flipped.
5. At the beginning of every learning cycle, after the new set of knobs has been calculated,

RandomProbability is decreased by *ChangeEachCycle*.

3.1.2 Inductive Logic Programming

Again, the goal of this learning algorithm is to find novel knob settings that produce the greatest number of desired effects. If this is not possible, the algorithm also identifies whether one or more of the sensors are faulty. Ultimately, the algorithm will return the knob settings that produce the greatest number of desired sensor readings. In order to determine which knob settings produce the greatest number of desired sensor readings, the Learner component uses the following algorithm:

1. Once the Learner receives the message to start a phase, it obtains the goal effects associated with that phase from the Knowledge Base. The Learner then obtains the knob settings associated with these effects.
2. Knob settings for different effects can conflict or overlap. To address this issue the Learner uses a scoring system to weigh the importance of each knob setting. Each knob setting is given one point for each association with an effect. Knob settings are in the format (PktSize ~ Hi), (PktSize ~ Lo) or (TransFreq ~ Hi) where the first part is the attribute name such as PktSize and the second part is the value such as Lo or Hi. For example, if (PktSize ~ Hi) is associated with two goal effects, it is assigned two points and if (PktSize ~ Lo) is associated with three goal effects, it is assigned three points.
3. Once the Learner component has a list of knob settings and their respective scores, it creates a group of knob settings containing the highest scored setting for each knob. The knob settings in this group are removed from the pool of knob settings that was generated in step 2.
4. The Learner tries this group of knob settings and records the sensor readings.
5. The group of knob settings is assigned a score based on how many sensors read Pro. The knob settings along with the score and the respective sensor readings are stored in the Learner.
6. If this is the first group of knob settings tried, it stored as the current best group of knob

settings. Otherwise, it replaces the current best group of knob settings if its score is higher.

7. The next highest scored knob setting in the pool from step 2 is replaced in the current group of knob settings. Steps 4, 5, and 6 are repeated.
8. Step 7 is repeated until the pool of knob settings is exhausted. At this point there should be several groups of knob settings along with their respective score and associated sensor readings. The Learner analyzes the groups of knob settings and sensors readings as follows:
    a. For the current best knob settings, find the sensors that read Con.
    b. Find the highest scored knob settings that produce a Pro reading for any of those sensors.
    c. Find the difference in knob settings from steps a and b and generate combinations from these differences.
    d. Try each of these combinations in turn. Store their score and sensor readings in the Learner.
9. If the Learner is unable to produce ideal knob settings in step 6 after an exhaustive search, it produces the best approximation it found and tries to determine whether there faulty sensors:
    a. For each sensor, find out any knob settings produced conflicting readings for that sensor.
    b. If the answer to step a is no and all knob settings produced Con readings for that sensor, mark that sensor as faulty.

3.2 Design

The Reasoner component was included in the initial version of PAGODA. The Reasoner fetched the knob settings that will optimize each of the goals from the Knowledge Base and chooses the knob settings that appear in the most goals in order to determine the optimal knob settings that will best encompass all of the goals. Thus, the Reasoner did not try different knob setting when it

encountered unexpected results or as the relationship between the knob settings and the sensors changed.

3.2.1 Component Interaction

We designed the Learner component to overcome the problem of static decisions by allowing it to use a learning algorithm that attempts to produces optimal knob settings by experimenting and learning from the results. The new Reasoner acts as a proxy between the Learner component and the rest of the PAGODA system in order to allow for a different Learner component to be used, implementing a different algorithm, to be used with minimal changes. The Knowledge Base stores basic knowledge for the Learner, consisting of knob settings that produce the desired effects for the goals under normal, or ideal, situation meaning no hardware malfunctions or adverse environmental factors. The Learner requests these knob settings from the Knowledge Base and uses them for its background knowledge, as described in the Learning Algorithm Section.

3.2.2 Alternative Design

An alternative implementation of the system would involve merging the Reasoner and Learner. This concept of putting the learning algorithm inside the Reasoner itself would be the most efficient at runtime, due to less rewrites being necessary for each communication with the learning algorithm, as well having one less block of code required for the system. The negative result of this move is that it would also reduce the ease of swapping learning algorithms in and out. Instead, the entire Reasoner-Learner component would need to be exchanged. Merging the Learner and the Reasoner may prove to be the best option once the system and components interactions are set, but in developmental phase, it is best to keep them separated. Our decision to keep them separated will allow the Learning component to be modified an updated more easily as well as adding the flexibility to add features to the Reasoner.

3.3 Implementation

3.3.1 PAGODA Architecture

We implemented two learning algorithms that interface with the other PAGODA modules to help the node better meet its goals. Though each of the algorithms operates differently, they both interface with PAGODA from within the Learner component. The Learner communicates with all of the other modules, such as the Coordinator and the Headquarters, through the Reasoner. The exception to this is communication directed at the Knowledge Base.

3.3.2 Hill Climbing

Hill climbing was the first algorithm that we implemented [RHeuristic1]. The hill climbing algorithm attempts to maximize the number of goals met as defined by the fitness function by randomly changing its parameters and testing for improvement. If the newly generated set of parameters has a lower fitness value than the previous set of parameters, the new set of parameters is discarded in favor of the previous set. Similarly, if the new set of parameters is better than the previous set of parameters, the new set of parameters is used.

In PAGODA, the parameters that the hill climbing algorithm is trying to optimize are the knob settings in the radio hardware. The first task for the algorithm is to generate new sets of knobs for comparison with the current knobs. This task is accomplished with the operator *randChangeKnobs*, which randomly changes a set of knobs according to some percent, thus generating a new set of knobs. The operator takes as input the current knob settings and the percent chance that a knob will be flipped. For every knob in the current knob settings, a random value from one to one hundred is generated. If the random value for the knob is less than the percent chance of a knob flip, then the knob is flipped.

After generating a new set of knobs the algorithm needs to determine if this new set is better than the previous set. Knob settings are tested by sending them to the hardware and then reading the

resulting sensors. Once sensors are read, they need to be compared with the goals to determine how well the goals are now met. The fitness function was designed for this purpose. It generates a numeric value indicative of the degree to which the sensor readings satisfy the current PAGODA goals. The fitness function is used to calculate a fitness scores for the sensors from both the initial knob settings and the new knob settings. Of these two, the fitness score with the higher value is set, or kept, as the current set of knob settings.

The generation of a new knob setting, the calculation of its fitness, and the comparison of the generated knob settings with the current knob settings constitute a cycle of the hill climbing algorithm. A cycle is started when a clock tick is received. However, clocks ticks are discarded when in the middle of a cycle, or whenever the Learner is otherwise occupied. At the start of the cycle a new set of knob is generated using *randChangeKnobs*. This new set knobs is sent to the Coordinator, which asks the HAL to change the knobs. The Learner then sends a request to the Coordinator to query the HAL for the current sensor readings. Once the sensors are returned, the fitness score is used to determine if the new set of knobs meets one or more goals. If the fitness score is higher, the new knob set is used; otherwise, the current set of knobs is kept. By performing several iterations of this cycle, it is possible to find a set of knobs that meets the most of the goals.

3.3.3 Inductive Logic Programming

In inductive logic programming there are three main components: background knowledge, positive and negative examples, and hypotheses [ILP94]. The goal in inductive logic programming is to create hypotheses that cover the background knowledge and the positive examples, but not the negative examples [ILP94].

In our implementation, the background knowledge consists of the expected relationships between knobs and effects, and between effects and sensors under normal conditions. This information is stored in the Knowledge Base in the following manner.

- The *PhaseEffects* attribute in the knowledge base contains the effects (or goals) associated with each phase. For example, *EnemyEngagement* is associated with *ImprovedConnectivity*, *MinimalResources*, *Security* and *DiffServ* (prioritizing certain nodes).
- The *EffectKnobs* attribute in the Knowledge Base contains the knob settings that are associated with an effect under normal conditions. For example, to obtain improved connectivity we would set transmission power to high, transmission frequency to low, routing updates to high, caching to high, and directional transmission to low.
- The *EffectSensors* attribute in the Knowledge Base contains the sensors that are used to detect certain effects. For example, to detect improved connectivity we need to check signal strength, interference, multipath, connection loss, and environmental conditions.

When the Learner receives a start phase message from the Coordinator, it queries the Knowledge Base for the phase effects. Once it receives the phase effects, it queries the Knowledge Base for all knob settings associated with those effects. The Knowledge Base responds by concatenating the list of the knob settings for each effect together. That is, if two effects require the same knob setting, then this knob setting will be present twice in the response from the Knowledge Base. Likewise, if two effects have conflicting settings for the same knob, both settings will be present in the response that the Learner receives. An example response would be:

```
(TransPwr ~ Hi) ; (TransPwr ~ Hi) ; (TransPwr ~ Lo) ; (PktSize ~ Hi)
```

Figure 14: A response of phase effects.

The Learner implements a voting system to summarize this information. Each knob setting is given one vote for each time it shows up in the response. For the above sample response, this voting system would produce:

```
((TransPwr ~ Hi) ~ 2) ; ((TransPwr ~ Lo) ~ 1) ; ((PktSize ~ Hi) ~ 1)
```

Figure 15: Voting on phase effects.

Equipped with this summary of the background knowledge (represented by the attribute *KnobSettings*), the Learner is able to generate its first group of knob settings. For each knob represented in the summary, the Learner picks the knob setting with the highest score, removes it from the summary, groups them together, and sends it to the Coordinator to be tested in the hardware. For the summary above, the Learner would send the following group of knob settings:

```
(TransPwr ~ Hi) ; (PktSize ~ Hi)
```

Figure 16: Knob setting reply from the Learner.

The summary then would only consists of (TransPwr ~ Lo), since it was the only setting that was not used.

Once the knobs are set in the hardware, the Learner requests sensor readings from the HAL. These sensor readings are provided by the HAL similarly to how the Knowledge Base provides knob settings: the sensor name along with the sensor reading (either Pro, Con, or Unk). Pro is the positive reading, and Con is the negative reading. Unk is neutral according to the design, but the algorithm treats it similarly to Con. This is because in the current implementation of PAGODA there are no goals associated with Con readings for sensors; i.e. all effects are associated with Pro readings for sensors.

Given the sensor readings, the Learner counts the number of Pro sensors, and records the sensors readings and the score, along with the knob settings used to produce those results in a data structure called a KSA (Knob Sensor Association). This KSA is stored in the Learner's internal database, represented by its *ExpResults* attribute. Since this is the first result, it is also stored as the best result in the attribute *BestResult*.

Before the Learner can analyze its database of results, it must first assure that it has a sufficient number of results to analyze. This minimum number of results is represented in the Learner by the attribute *MinResults*. At this point, there may be knob settings remaining in the summary that was created when the Learner first obtain the background knowledge. The Learner uses the knob settings contained in the summary to produce more results:

- Select a knob setting from BestResult.
- If there is another setting for the same knob in the summary, generate a new group of knob settings consisting of the knob settings from BestResult, with the setting from summary substituted into it.
- If there isn't another setting in the summary, pick another knob setting.

Once a new group of knob settings is produced, the same procedure of trying them, reading the sensors, recording the results as a KSA in ExpResults, and setting BestResult to the appropriate value is repeated.

Once the summary is empty, the Learner switches to the analysis stage regardless of whether or not the minimum number of results has been reached. Of course, the Learner also switches to this stage if the minimum number of results is reached before the summary is exhausted.

The goal of the Learner in the analysis stage is to find the flaws (i.e. Con sensors) in the best result it has found so far, and then figure out how to fix them (i.e. get a Pro reading). As mentioned, a result consists of knob settings, sensor readings, and a score. To find the best result, the Learner first finds the best complement to the best result so far. A complement to a result A is another result B that has Pro sensors where result A has Con sensors. The best complement is a result C that has the greatest number of Pro sensors.

Once the best complement to the best result is found, the Learner analyzes the knob settings of the best result and its complement to determine which knobs to change:

- First the Learner finds the Pro sensors in the complement for which the best result had a Con reading.
- The Learner queries the Knowledge Base for all knob settings associated with these sensors. The Knowledge Base finds the effects associated with the sensors, and then finds the knobs associated with the effects.
- Similarly to the very first knob settings generated, the Learner weighs each knob setting according to how many times it appears.
- The top three (this is an adjustable parameter) knob settings are substituted into the group of knob settings that represents the best result.
- The generated group of knob settings is tried in the hardware, the results recorded, and the analysis is repeated.

The analysis stage of the Learner stops if more than a certain number of results have been reached, in which case the Learner produces the best result it has found so far, or if the ideal (i.e. all sensors read Pro) result is found.

3.4 Analysis

3.4.1 Hill Climbing

The hill climbing algorithm has been demonstrated to work by observing that the fitness function increases as the agent progresses through the phases. We ascertained that there was still a need to test several approaches to the hill climbing algorithm itself. There are two constants within the code of the hill climbing algorithm. One variable adjusts the initial probability of each knob being flipped when creating a new random set of knobs from the original knobs, and the other controls the rate at which simulated annealing occurs; zero being simulated annealing off. We conducted numerous

runs to test which combination of variables were optimal given normal conditions as well as the unusual conditions of knobs and/or sensors not working. The number of learning cycles it takes to reach the known optimal set of goals for each set of conditions is presented in the results section. This algorithm is also used for comparison with the Inductive Logic Programming algorithm as is outlined in the next subsection below

3.4.2 Inductive Logic Programming

There are several levels of analysis that we preformed on the inductive logic programming learning algorithm. Again, the first step in the analysis of the algorithm was the verification that it adjusts the knobs to produce better sensor readings as the algorithm progresses. Additionally, we looked to see if the normal case sensors were better with the learning algorithms than with the original way of calculating knobs. We also, through the use of settings in the Madrad component to produce unusual and faulty sensors, demonstrated the algorithms ability to overcome unexpected interactions between the knobs and the sensors. Unusual sensor readings were produced to coincide with realistic scenarios such as dense foliage, a high concentration of nodes, or rapid movement. Each possible scenario was run on top of the normal mode and has its own variation on the expected sensor readings. It was important in this test as well as others to ensure that the sample run was long enough to allow the algorithm a chance to adapt to the current scenario.

Analysis with the advanced functionality of Madrad was used as a way to confirm that the algorithm works and show the algorithm's success in different situations, but a baseline for comparison was needed. One way of determining the performance of the inductive logic programming algorithm was to compare its performance with that of the hill climbing algorithm and the base PAGODA system without a Learner in a controlled simulation using Madrad.

A large part of the learning component analysis was collecting data from the run. Information such as the length of the run, the number of rewrites, the goals met, and the percent of sensors that read

'Pro' during the run was collected over multiple test runs. There were sample runs with the inductive logic programming learning algorithm in place, with several variations on the hill-climbing algorithm in place, and with just the base PAGODA system without any modifications. After numerous test runs were preformed under each of these conditions, the resulting data was analyzed. We were able to discover from the data which of the algorithms preformed the best under the different simulated conditions.

Section 4: Results

4.1 Hill Climbing

4.1.1 Data Collection

To analyze the hill climbing algorithm we ran a series of tests and observed the results. The graphs and plots included in Appendix A are a comprehensive collection of the data obtained in over five hundred runs of PAGODA, using a Learner component that implements the hill climbing learning algorithm with simulated annealing.

Each collected data point represents a unique run. The data points collected over the runs reflect changes to two variables: the initial probability of a knob being changed, and the amount by which this probability decreases in every learning cycle (it's also called the annealing rate). Recall from the design section that simulated annealing is a process where the probability of a knob being flipped decreases every cycle by an amount equal to the annealing rate.

The first variable, the initial probability, is not shown as an axis in the graph in Figure 1. Instead, several graphs are grouped by their initial probability. The second variable, the annealing rate, is shown as the horizontal axis. The vertical axis represents the number of learning cycles until the best known combination of sensors was reached. A learning cycle is the process of randomly generating a set of knobs, sending the new set of knobs to the hardware, evaluating this new set of knobs by checking the effect the new knobs have on the hardware, and determining whether to keep the new set of knobs.

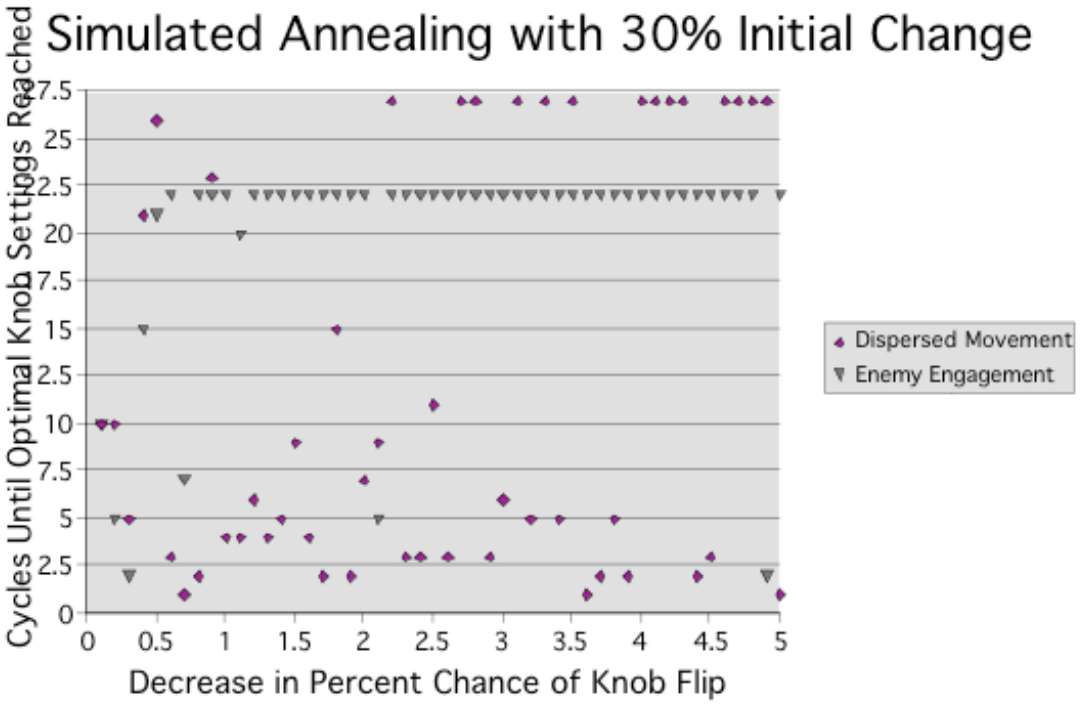


Figure 17: Simulated annealing with 30% initial change.

The generated graphs contain data sets generated for the Enemy Engagement and Dispersed Movement phases. The Parking Lot phase is not shown because it has very simple goals and the optimal sensor readings appear immediately at the start of the algorithm. This means that the learning algorithm had nothing to do and the data is simply a flat line along the horizontal axis.

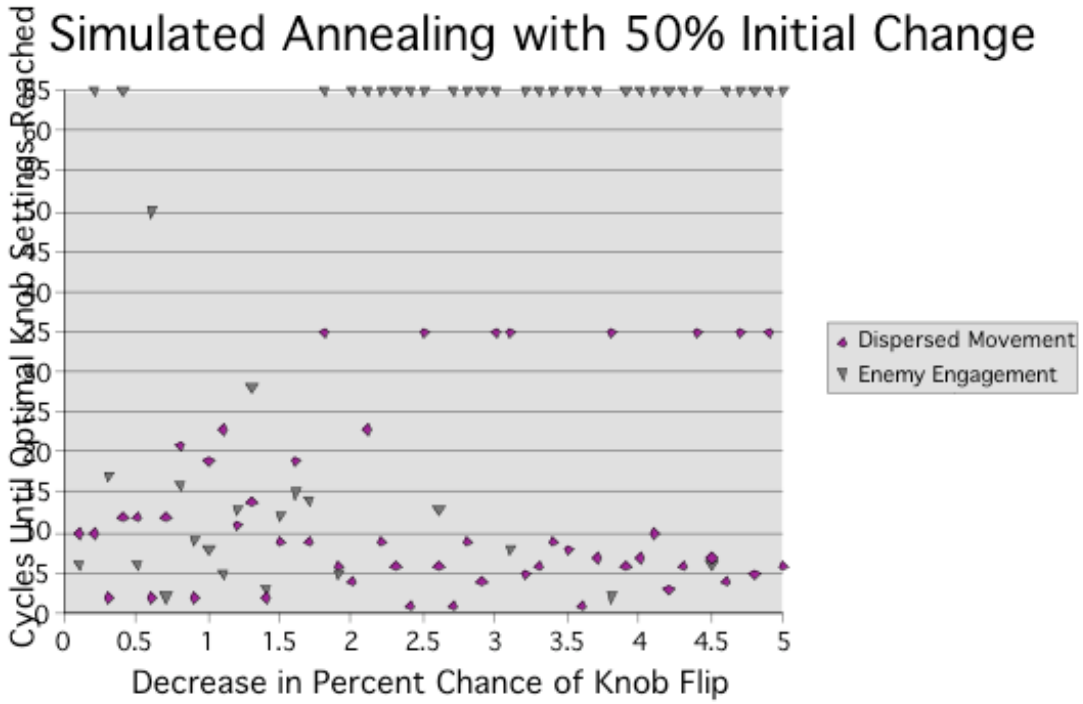


Figure 18: Simulated annealing with 50% initial change.

4.1.2 Smoothing the Data

The use of a random number generator as part of the hill climbing algorithm generated very noisy data, as can be seen in Figure 18. This noise made the graphs extremely difficult to analyze for clear patterns. In an effort to find patterns in the data, we decided to smooth the data by averaging some of the numbers together. Smoothing the data is a common technique to reduce noise and extract more pronounced patterns in the data. We smoothed our analysis data by averaging each data point with the two values that were sequentially greater then the data point and the two values that were sequentially less then the data point in the data set.

Smoothing the data creates a new problem when the best known combination of sensors is never found during a run. Effectively the value of best combination of sensors is unknown, and therefore assumed to be infinity. This causes the surrounding values of other runs to be skewed towards infinity (i.e. remember that each run is a data point). To address this problem we assigned runs

with unknown/infinite values as one plus the maximum number of cycles until the best group of sensors was found. For example, consider a dataset containing runs with values of 4, 10, 13, and 15, where the value represents how many learning cycles it took to find the best result for that run. These runs with unknown/infinite values would therefore assigned 16. This allows for the smoothed data to take into account the values where the best set of sensors were not found without skewing several answers where the best sensor readings were found due to one where the best sensor readings were not.

4.1.3 Data Analysis

Figure 19 is a smoothed graph with an initial probability of fifty percent of flipping knobs. The axes are as described earlier with the horizontal axis showing the decrease in the probability of a knob flip, or the annealing rate, and vertical axis represents the number of learning cycles until the best known combination of sensor readings was reached. The Enemy Engagement phase is shown in gray and Dispersed Movement is shown in purple.

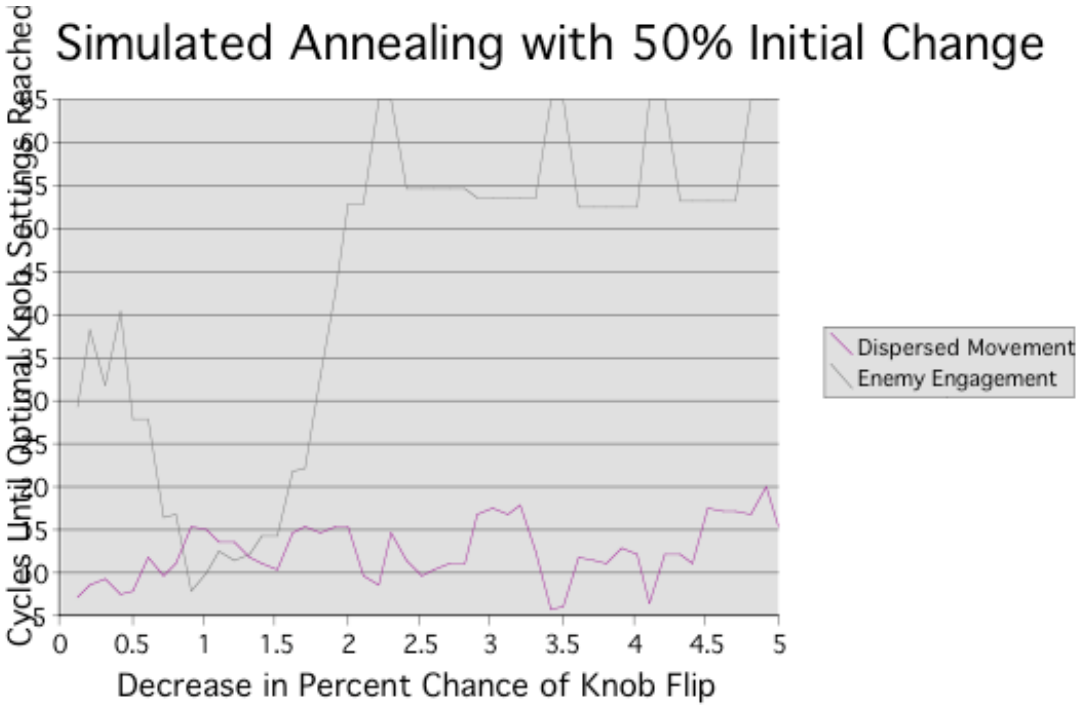


Figure 19: Simulated annealing with 50% initial change.

In order to determine the performance of a specific annealing rate in Figure 19, we trace the point up vertically to the desired phase line. Consider an annealing rate of 1.5. Tracing this point up the vertical axes to the gray line for Enemy Engagement shows that with an annealing rate of 1.5 it took 15 learning cycles to find the best set of sensor readings. This means that for the hill climbing algorithm, with an initial probability of the knobs being flipped set to 50 percent and with an annealing rate of 1.5, it took 15 learning cycles till the best sent of sensor readings were found.

Let's use Figure 19 to analyze the performance of the algorithm under the Enemy Engagement phase. When the annealing rate drops between around 0.5 and 1.75 the algorithm finds the best group of sensor readings within around 25 cycles. Over the rest of the graph the solution is found after a much longer time, or not at all. This indicates that the probability with which the knobs are flipped decreases significantly in a very short time, far too quickly for any learning to occur. Consider that with an annealing rate of 1.75 it takes approximately 30 cycles until a solution is found. By that point

the probability of a knob being flipped is 50 - (30 * 1.75), or 2.5 percent. As the annealing rate increases along the horizontal axis it becomes harder for the optimal set of knobs to be found due to the probability of a knob being flipped decreases below a probability of 2.5 ever more quickly. On the other hand, with annealing rates lower than 0.5, it takes around 35 learning cycles to find the best set of knobs settings. This indicates that it takes quite a few learning cycles before the percentage drops low enough for the best set of sensor readings to be found. This graph clearly shows that simulated annealing produces the best results for Enemy Engagement when the annealing rate is between 0.5 and 1.75.

Figure 20 is another smoothed graph, this time with an initial probability of 30 percent. Let's use this graph to analyze the performance of the algorithm in the Dispersed Movement phase. Notice that with the annealing rates between around 0.75 and 2.0 the algorithm finds the best group of sensor readings within around 10 learning cycles. Unlike Enemy Engagement in Figure 19, there are other areas in Figure 20 where the algorithm is likely to find the best sensor readings for the Dispersed Movement phase. This shows that the Dispersed Movement phase has different set of parameters that optimize it from those that optimize the Enemy Engagement phase.

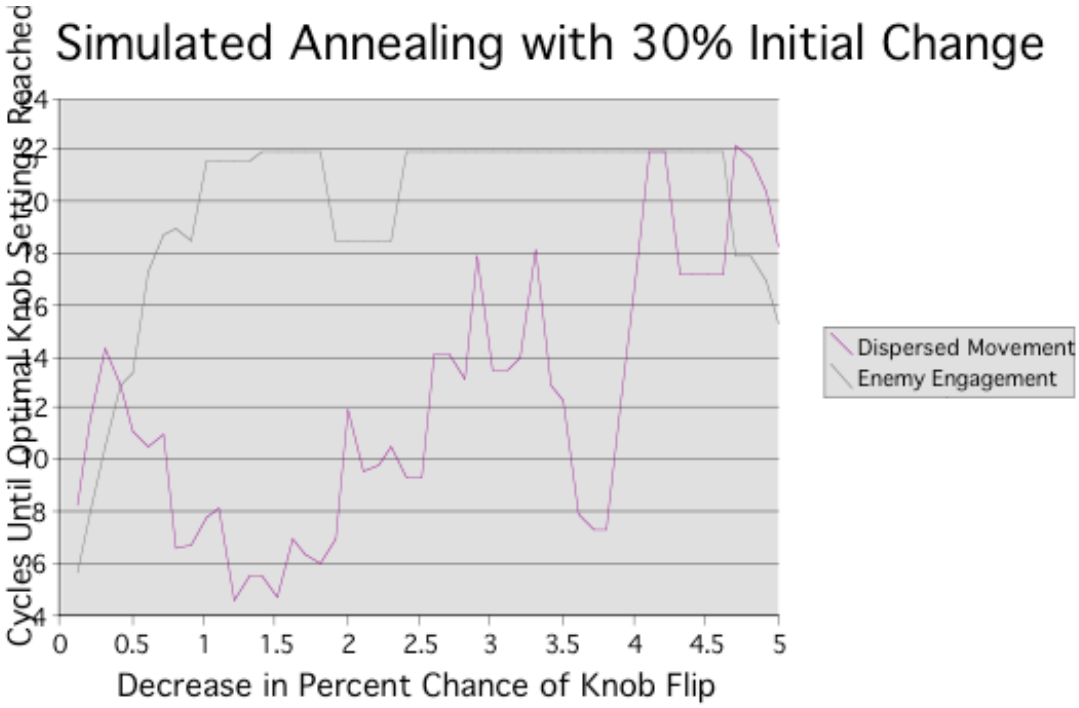


Figure 20: Simulated annealing with 30% initial change.

From the graphs in Figure 19 and Figure 20 we see that there are different combinations of initial probabilities of a knob being flipped and ranges of annealing rates that generate the best sensor readings more quickly for the Dispersed Movement and Enemy Engagement phases. For the Enemy Engagement phase, using an initial probability of 50 percent and an annealing rate of between around 0.5 and 1.75 we get the best sensor readings within 25 learning cycles. For the Dispersed Movement phase, using an initial probability of 50 percent and an annealing rate of between 0.5 and 1.75 will eventually find the best sensor readings but can be tweaked to find the set of sensors more quickly with some optimization. One way to optimize the Dispersed Movement phase is to use an initial probability of 30 with annealing rates between around 0.75 and 2.0, which will get the best sensor readings within 10 learning cycles. For the Enemy Engagement phase, using an initial probability of 30 with annealing rates between around 0.75 and 2.0 will most likely not find the best sensor readings. This shows how important it is to optimize the initial probability as well as the annealing rate in order to make the

algorithm as efficient as possible at finding the best sensor readings.

4.2 Inductive Logic Programming

In order to test the Inductive Logic Programming algorithm we decided to compare its performance to the performance of PAGODA without a Learner (i.e. Normal), and the performance of the hill climbing algorithm under the Normal, Unusual, and Faulty modes of operation of the hardware as described in the PAGODA background chapter, under the MadRad section. Each algorithm was evaluated based on the number of sensors that read Pro.

The table below shows these results. Along the top are the different phases of operation in the PAGODA implementation, as well as the maximum number of sensors read in that phase. Along the left are the different modes of operation of the hardware, along with rows for the hill climbing algorithm, the ILP algorithm, and PAGODA without a Learner. Along the top the maximum sensors are for each phase are listed, and under each phase the number of goals met by each algorithm under each mode is listed. A question mark in any field indicates a result could not be produced for that setting.

Both algorithms and basic PAGODA without a Learner (i.e. “Base”) performed well in Parking Lot under Normal and Unusual. This is because this phase has fewer goals than the other phases.

There were a higher number of question marks, or results that could not be produced, under the Dispersed Movement column than under Parking Lot and Enemy Engagement comumns. This might indicate implementation problems with Dispersed Movement because the algorithm could not complete the phase.

Table 1 - Algorithm comparison

Mode	Learning Component	Parking Lot	Dispersed Movement	Enemy Engagement
Maximum Sensors		3	8	9
Normal	Base	3	?	5
	Hill Climbing	3	7	8
	Inductive Logic	3	3	6
Unusual	Base	3	?	5
	Hill Climbing	3	7	7
	Inductive Logic	3	3	6
Faulty	Base	1	?	3
	Hill Climbing	3	6	7
	Inductive Logic	?	?	?

As can be seen from the table, the hill climbing algorithm outperforms ILP and PAGODA without a Learner (Base). This can be seen as a higher number of 'Pro' sensors under the hill climbing and ILP rows in the table. ILP outperforms Base slightly when the hardware is in Normal and Unusual mode, but not in Faulty mode, as ILP does not yet support that mode due to time constraints on the project. Hill climbing performs slightly worse in Faulty mode than in Normal mode and Unusual mode. This is to be expected, as some sensors will never read 'Pro' in Faulty mode.

Section 5: Future Work

The PAGODA system is still in development, and thus there is room for extension and improvement. These improvements include: implementing another learning algorithm such as Bayesian learning or support vector machines; tweaking parameters of the inductive logic programming algorithm, or adding multiple node support to PAGODA.

The most straightforward and perhaps desirable extension is the addition of a Learner that implements a different machine learning algorithm. One that would be of great benefit to the system would be a Bayesian learning algorithm [Mitchel97]. Implementing a Bayesian learning algorithm would most likely require work with the Maude metalevel. The metalevel is a Maude module that allows reflection so that a program can create new rules. By adding the rules to the program itself at runtime, the system would be able to adapt very natively to the environment.

The inductive logic programming algorithm has several policies and parameters that can be tweaked to increase its successfulness. One way to tweak the algorithm would be to experiment with different ways of weighing the knobs, and adding weights to sensors. Similarly to knobs, in some cases it may be worthwhile to sacrifice one or two trivial sensors to gain a 'Pro' reading on a single more important sensor. Another area of the algorithm that is open to improvement is the generation of the initial knowledge from trial runs. Currently the algorithm selects knobs settings that overlap the most and use those to generate new knob settings. An alternate approach would be to experiment with knob settings randomly, or to generate the initial experiments using the hill climbing algorithm. Finally, the most important area that is open for improvement is the analysis of results. The algorithm tries to produce novel knob settings to achieve goals by inferring the relationship between knob settings and sensor readings without storing the information it produces in the process. This information could be stored as hypothesis in the Learner and used to enhance the production of new knob settings.

A PAGODA network's effectiveness can be increased by allowing the Learner components in

other PAGODA nodes within the network to share their knowledge with one another. In a scenario where all nodes are trying to achieve the same goal the benefit would be very large, especially if there are many agents working together on the problem.

Another improvement to the PAGODA system would be to allow knobs with a range of discrete or continuous values. Binary knobs are easier to program and understand, but they do not always reflect real world scenarios. It is not very likely for every piece of hardware in the field to have only a high power or low power setting for a transmitter, or a transmission frequency knob that only can access two frequencies. By adding the ability to set a knob to a number, say between 0 and 100, the program gains flexibility during run time as well as making the simulation more realistic.

There is also an opportunity for developing an editor for Maude. Currently there is not any standard integrated development environment for Maude, or even a basic editor that supports its syntax highlighting. For this project we have been using Eclipse since it provides an open source cross platform plain text editor. Another great benefit of Eclipse is its CVS features, which we used for group editing and maintaining a versioning system. Given that it is possible to interface Maude with Java [MQP], it is worth investigating the possibility of developing a Maude plug-in for Eclipse, or even an editor that performs syntax highlighting.

Section 6: Conclusions

Our primary design challenge was to implement two machine learning algorithms that could be used in a Learner component within the PAGODA system. Previously the Reasoner component in PAGODA made decisions based on relationships between knobs and effects, and between sensors and effects under normal conditions. By implementing learning algorithms to learn and adapt we have enabled PAGODA to improvise under unexpected or adverse conditions.

We decided to make the Learner components modular so that they could be interchanged depending on the situation. The first Learner component was based on a hill climbing algorithm. This design randomly changed a group of knob settings in successive iterations, trying to optimize the results. The second algorithm we implemented was based on inductive logic programming. This algorithm tries knob settings based on all the results it has already seen.

To test the algorithms we compared them to each other. We expected the inductive logic programming algorithm to surpass the hill climbing algorithm under unexpected and adverse conditions. Likewise, we expected both learning algorithms to far exceed the base PAGODA system with no Learner component. The hill climbing algorithm performed better than basic PAGODA, and unexpectedly, better than the ILP algorithm, both under normal mode and under unusual and faulty modes. The ILP algorithm did slightly better than the basic PAGODA architecture, which did not have a learning component.

References

[CSL] SRI Computer Science Laboratory. SRI International. 13 Dec. 2005 <http://www.csl.sri.com/>.

[desJardins92] Desjardins, Marie E. PAGODA: A Model for Autonomous Learning in Probabilistic Domains. Diss. Univ. of California, Berkeley, 1992. 21 Oct.-Nov. 2005.

[dvorak-ieee00] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks.
Software Architecture Themes In JPL's Mission Data System.
In IEEE Aerospace Conference, USA, 2000.

[email] Talcott, Carolyn. "Project Description." E-mail to David Finkel. 30 Oct.-Nov. 2005.

[History] History of SRI International. SRI International. 13 Dec. 2005 <http://www.sri.com/about/history>.

[ICSD] Information and Computing Sciences Division. SRI International. 13 Dec. 2005 <http://www.sri.com/icsd/>.

[ILP94] Lavrac, Nada, and Saso Dzeroski. Inductive Logic Programming: Techniques and Applications. New York: Ellis Horwood, 1994. 6 Dec. 2005 <http://www-ai.ijs.si/SasoDzeroski/ILPBook/>.

[Inventions] Timeline of SRI International Innovations: 1946 - 1960. SRI International. 13 Dec. 2005 <http://www.sri.com/about/timeline/timeline1.html>.

[Manual] Clavel, Manuel, Francisco Duran, Steven Eker, Patrick Lincoln, Narciso Marti-Oliet, Jose Meseguer, and Carolyn Talcott. "Maude 2.2 Manual and Examples." The Maude System. Dec. 2005. SRI International. 10 Jan. 2006 <http://maude.cs.uiuc.edu/maude2-manual/>.

[Mars] "Space Communications with Mars." Astronomie Amateur. 5 Dec. 2005 <http://www.astrosurf.org/lombry/qsl-mars-communication3.htm>.

[Manual] Clavel, Manuel, Francisco Duran, Steven Eker, Patrick Lincoln, Narciso Marti-Oliet, Jose Meseguer, and Carolyn Talcott. "Maude 2.2 Manual and Examples." The Maude System. Dec. 2005. SRI International. 10 Jan. 2006 <http://maude.cs.uiuc.edu/maude2-manual/>.

[Mitchell] Mitchell, Tom M. Machine Learning. 1997, McGraw-Hill Companies, Inc. New York, New York

[MB99] Muggleton, Stephen, and Michael Bain. Analogical Prediction. International Workshop on Inductive Logic Programming, 1999, University of York. 6 Dec. 2005 <http://www.cs.bris.ac.uk/~ILPnet2/Tools/Reports/Abstracts/ilp99-muggleton-bain.html>.

[MQP] Visualizing Goals in Autonomous Space Systems. 16 Dec. 2004. Worcester Polytechnic Institute. 24 Nov. 2005.

[muscettola-etal-ai98] N. Muscetolla, P. Pandurang, B. Pell, and B. Williams. Remote Agent: To

Boldly Go Where No AI System Has Gone Before. Artificial Intelligence, 103(1-2):5-48, 1998.

[Muggleton] S.Muggleton and F.Marginean. “Logic-based machine learning”. In J.Minker, editor, Logic-Based Artificial Intelligence, pages 315-330. Kluwer, 2000.

[PAGODAdoc] “PAGODA Documentation”. SRI International. 13 Dec. 2005 <http://pagoda.csl.sri.com/PAGODA.zip>.

[Primer] McCombs, Theodore. “Maude 2.0 Primer.” Aug. 2003. SRI International. 5 Dec. 2005 <http://maude.cs.uiuc.edu/primer/maude-primer.pdf>.

[Quinlan95] Quinlan, J. R., and R. M. Cameron-Jones. “Induction of Logic Programs: FOIL and Related Systems.” New Generation Computing 13 (1995): 287-312. 01 December 2005 <http://www.cs.nmsu.edu/~ipivkina/KLAP/quinlan95induction.ps>.

[RD] R&D Divisions. SRI International. 13 Dec. 2005 <http://www.sri.com/rd/>.

[RHeuristic1] Norvog, Peter and Russell, Stuart; Artificial Intelligence A Modern Approach Second Ed, Pearson Education Inc., NJ, 2003

[RHeuristic2] "Heuristic." Wikipedia. 5 Dec. 2005 <http://en.wikipedia.org/wiki/Heuristic>.

[Shannon] Shannon, C.E. (1950). Programming a computer for playing chess. Philosophical Magazine, 41(4):256-275

[Talcott05] Talcott, Carolyn “PAGODA” (October 26, 2005)

[TalcottTut] Talcott, Carolyn. "Pathway Logic Tutorial." 7 Dec. 2005 <http://www.csl.sri.com/~clt/ABC/>.

Appendix A: Hill Climbing Data

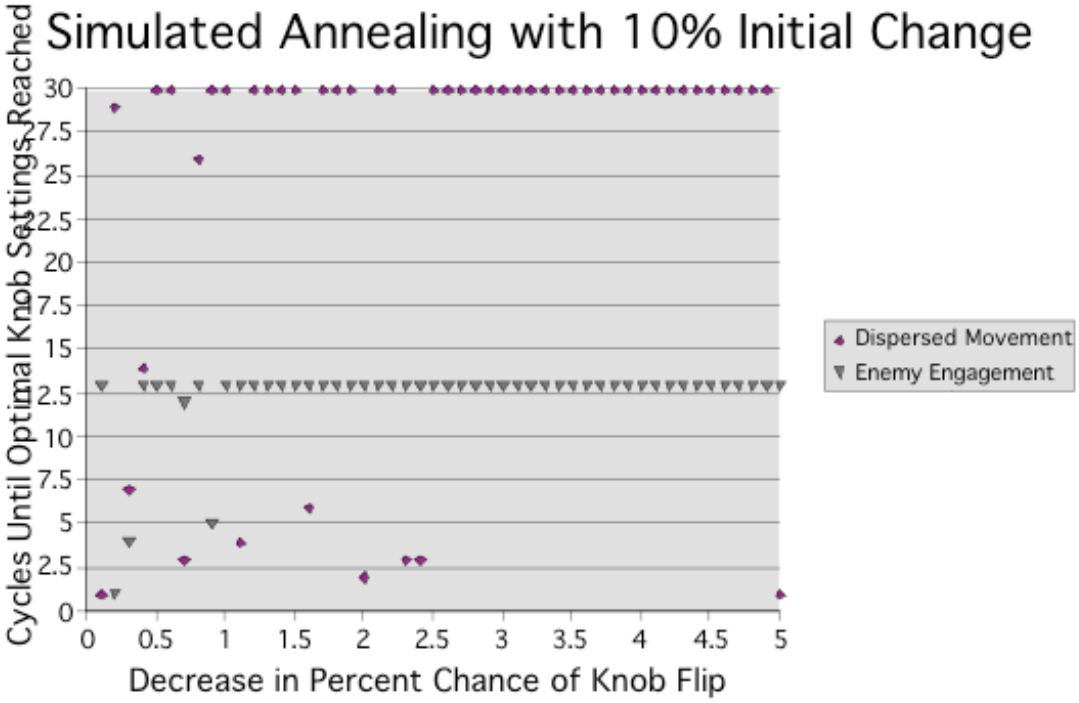


Figure 21

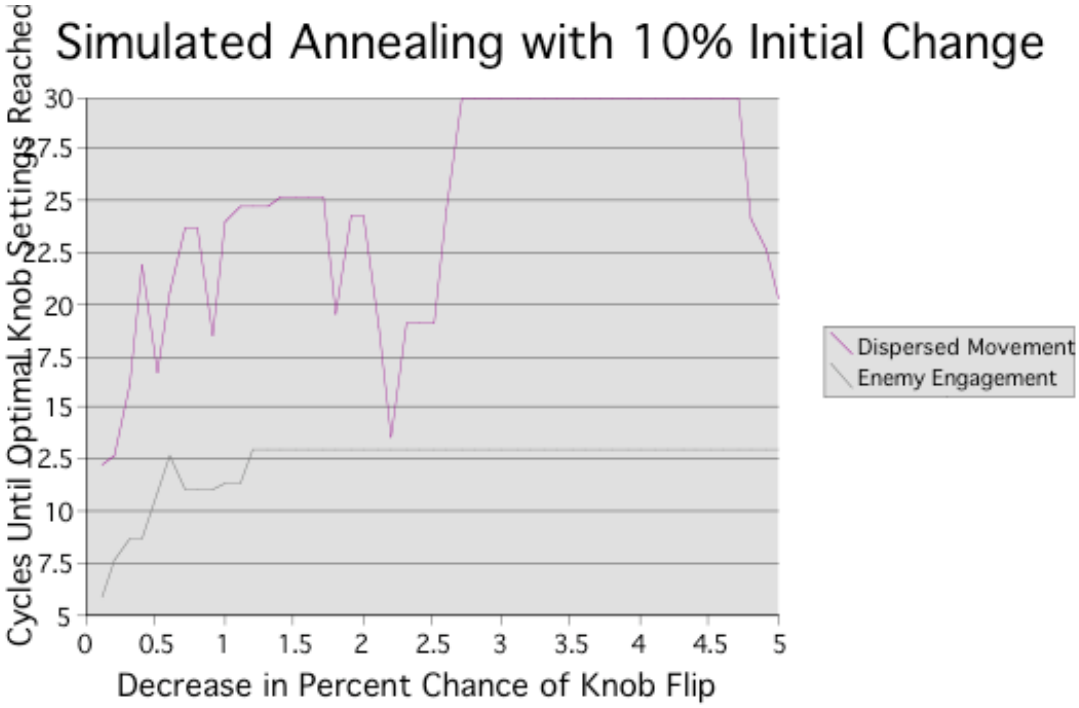


Figure 22

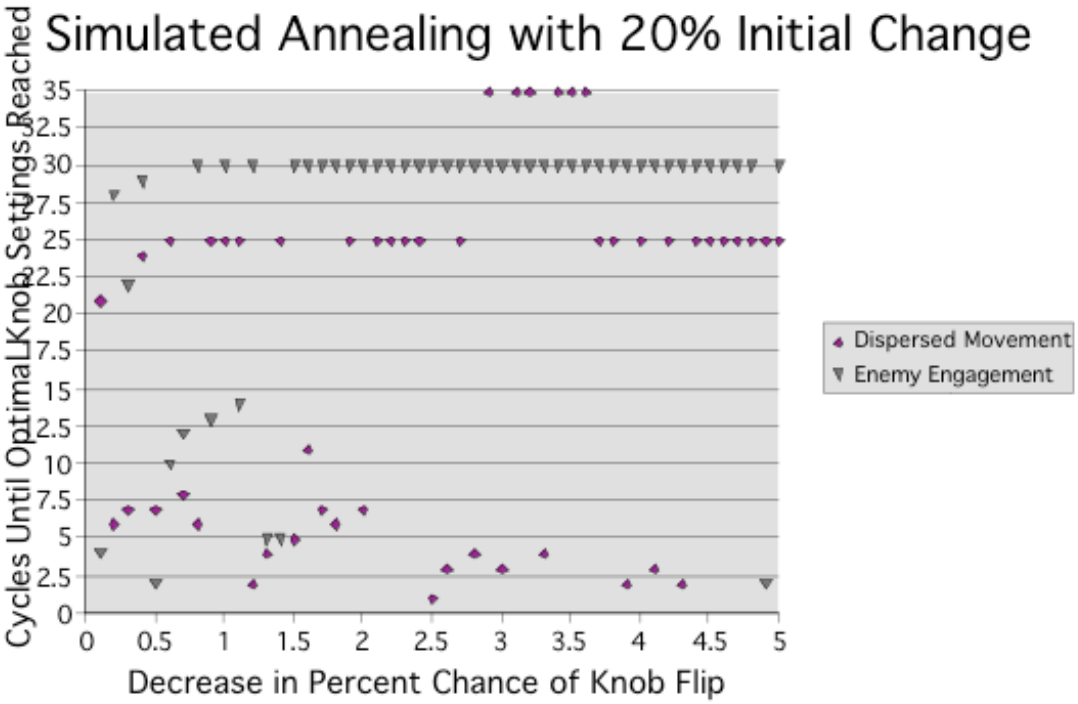


Figure 23

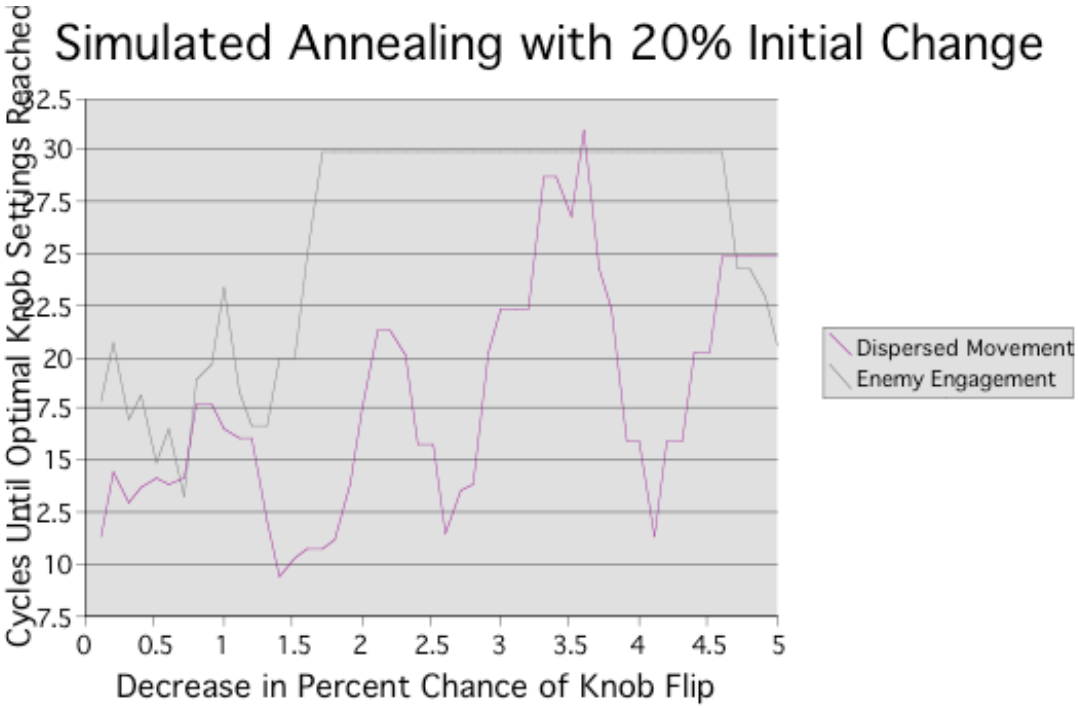


Figure 24

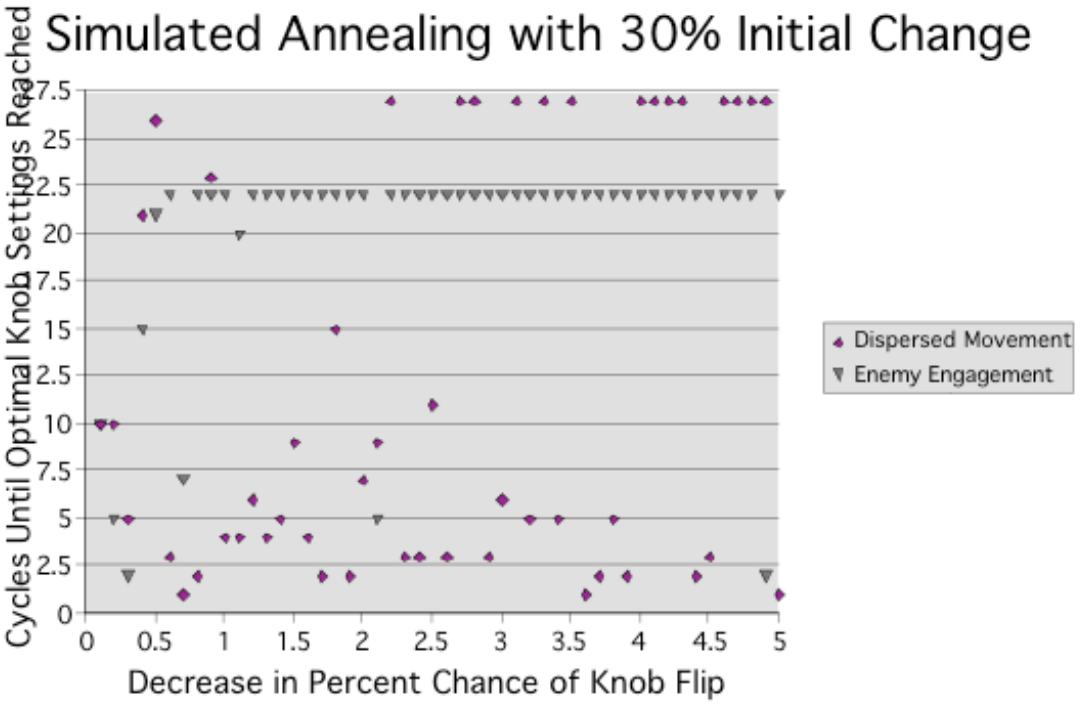


Figure 25

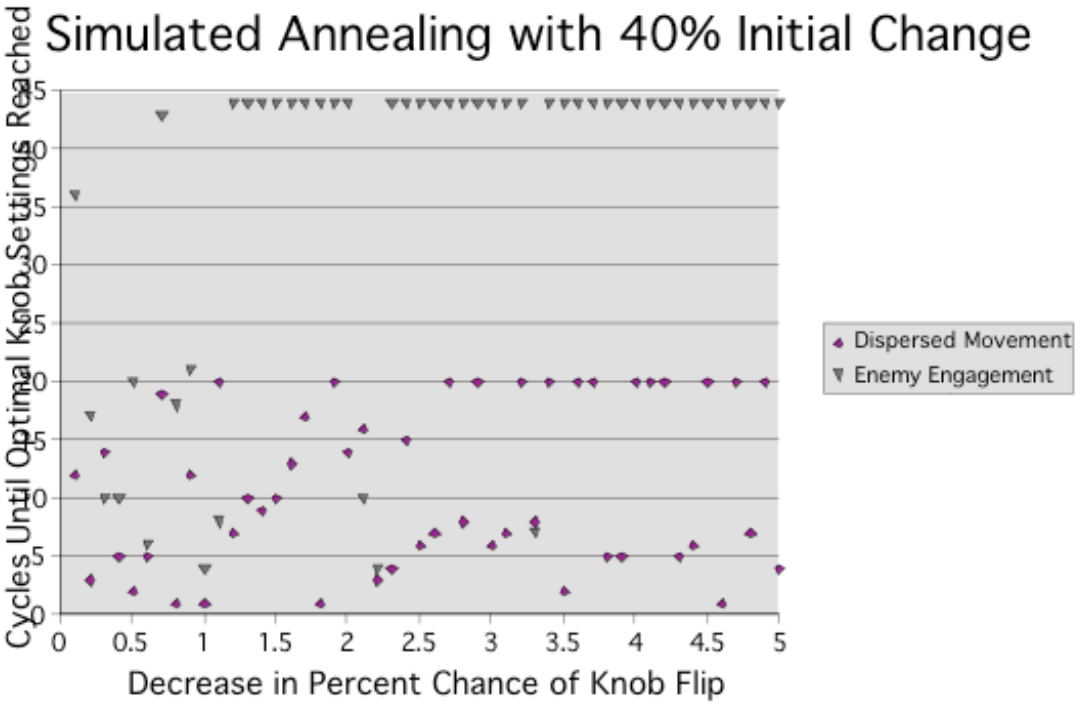


Figure 26

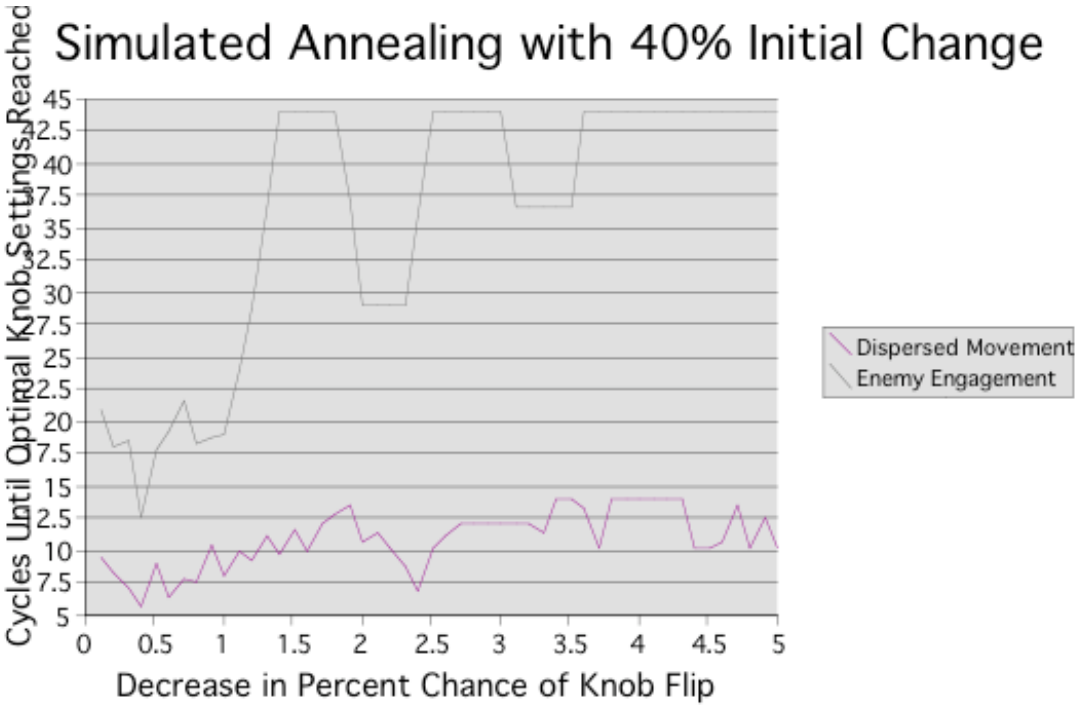


Figure 27

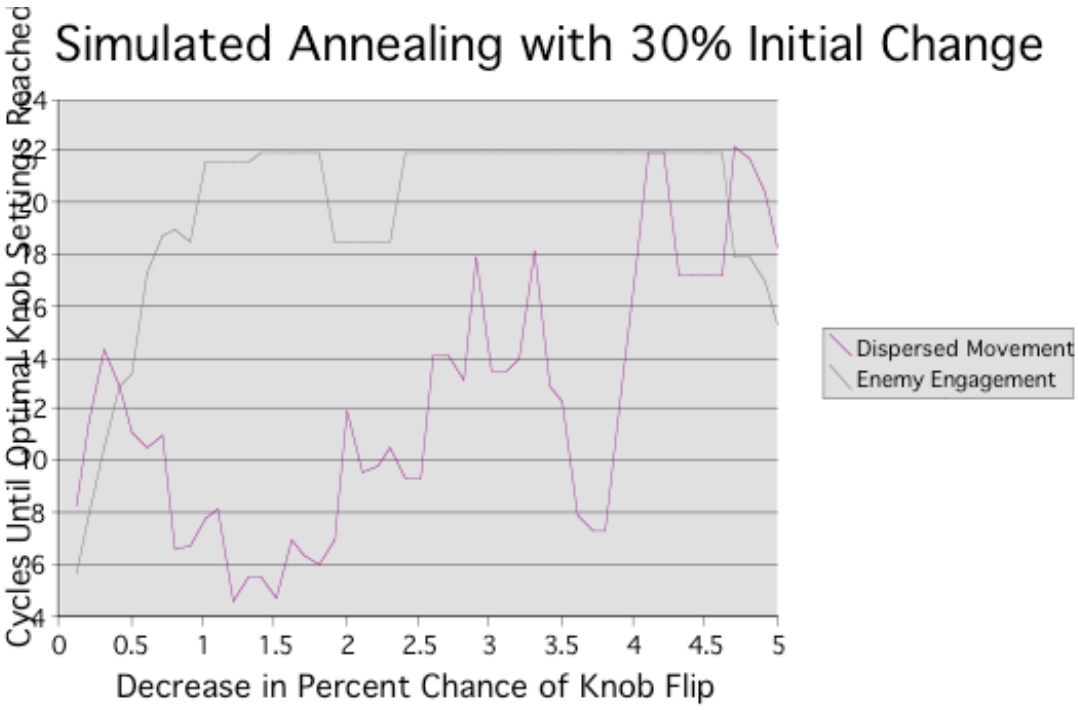


Figure 28

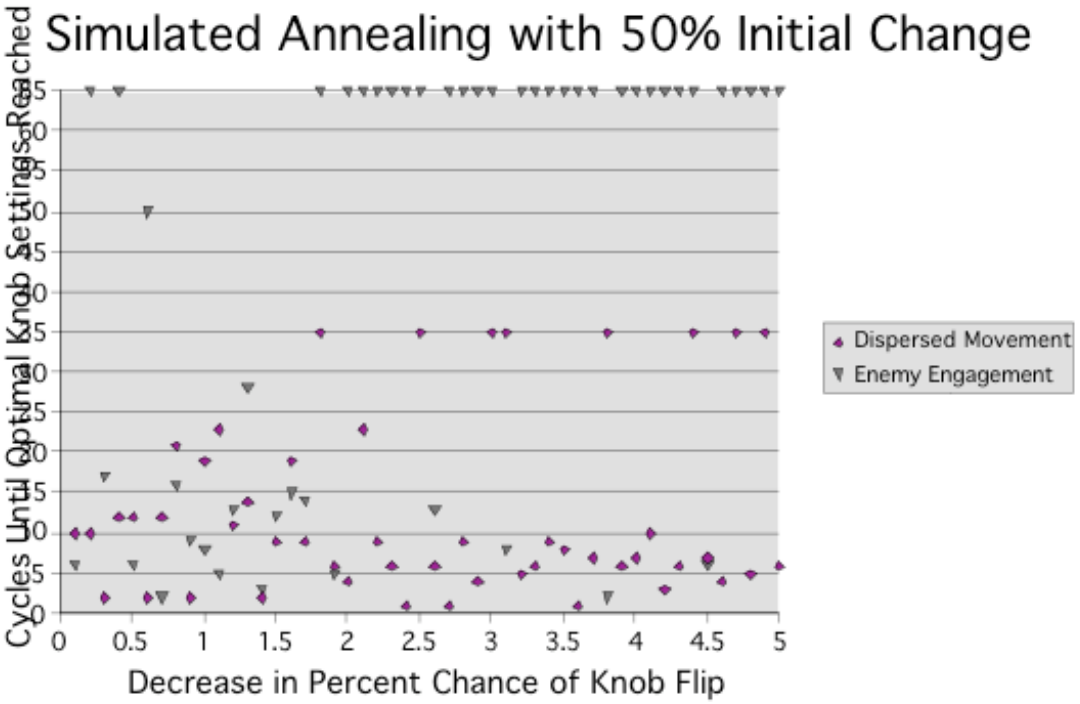


Figure 29

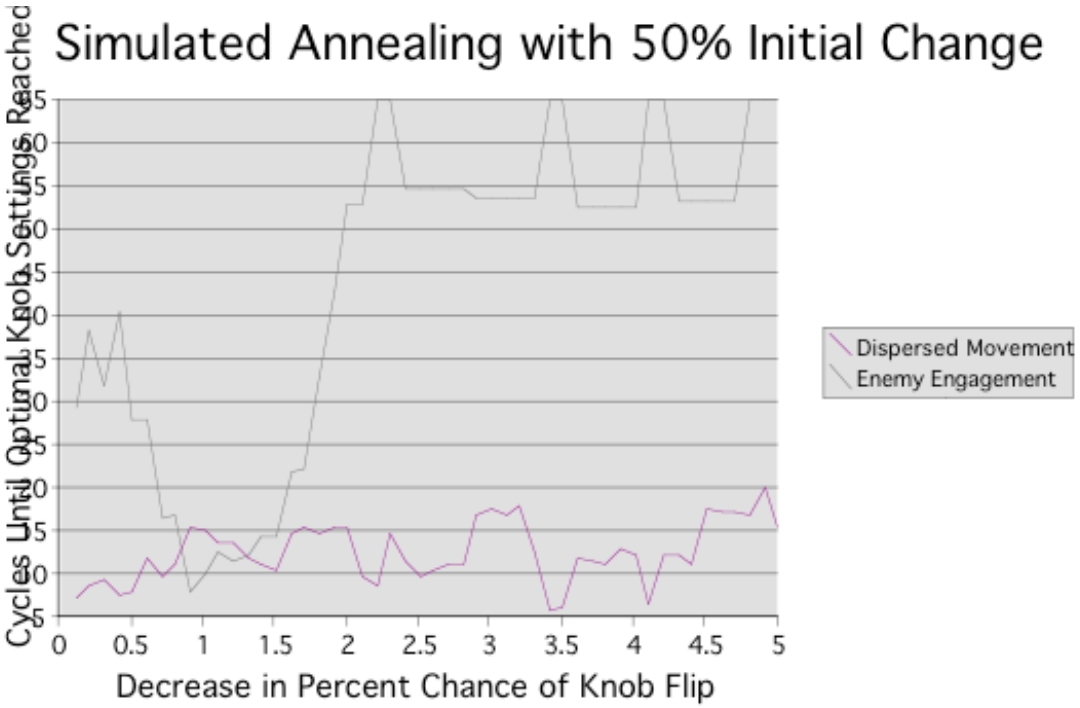


Figure 30

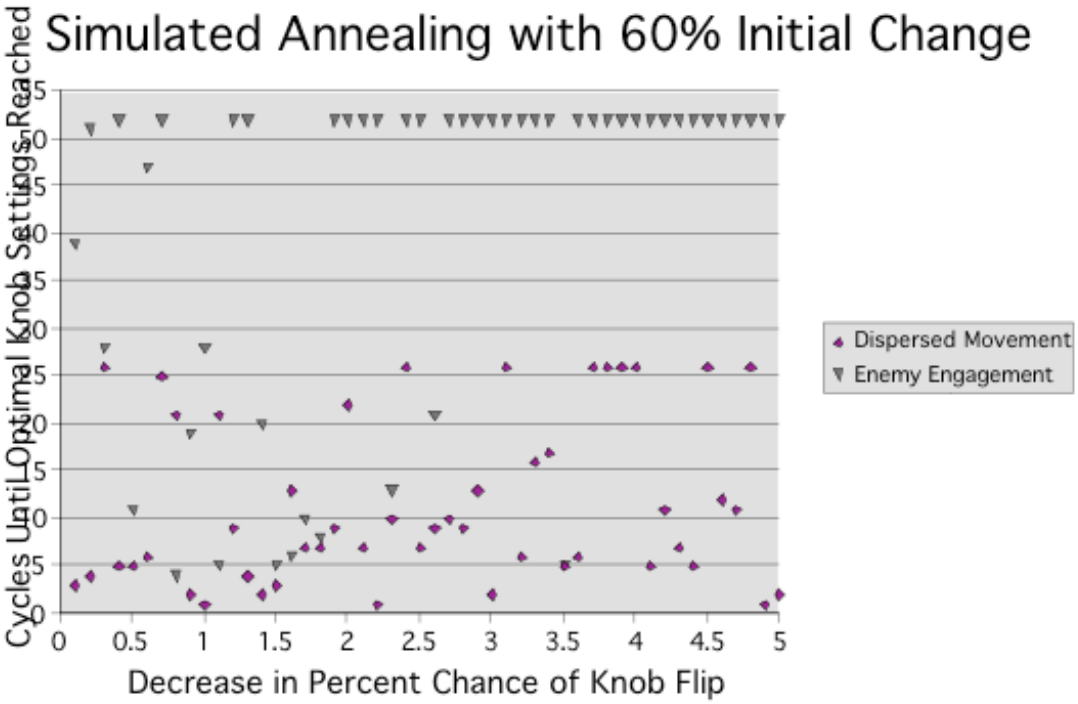


Figure 31

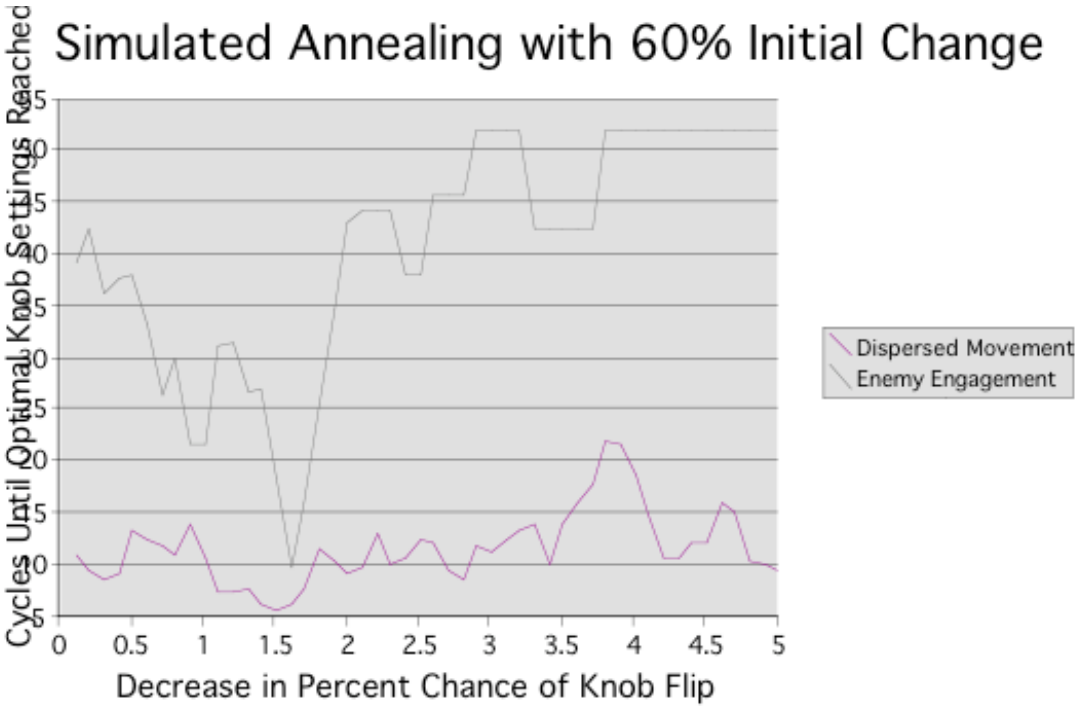


Figure 32

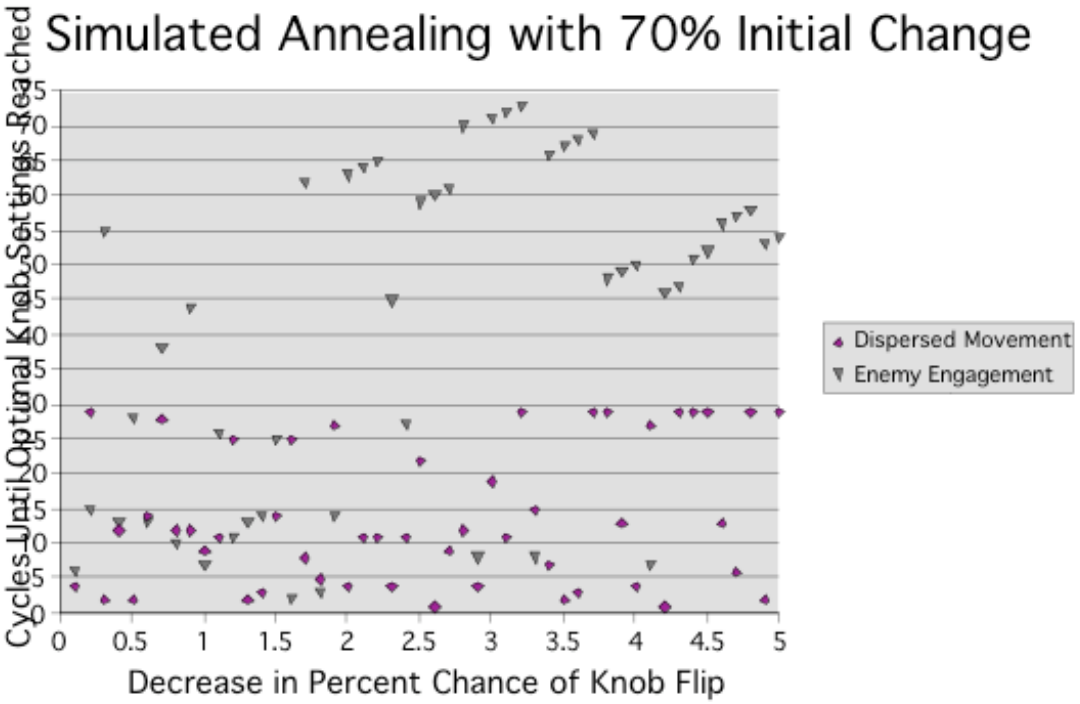


Figure 33

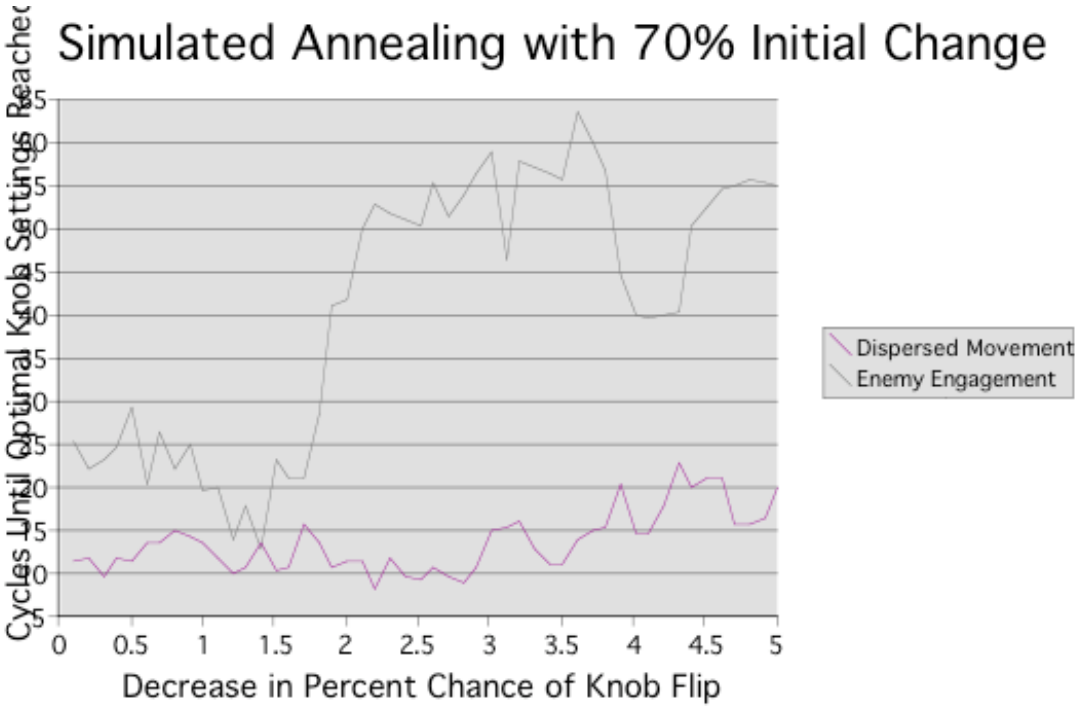


Figure 34

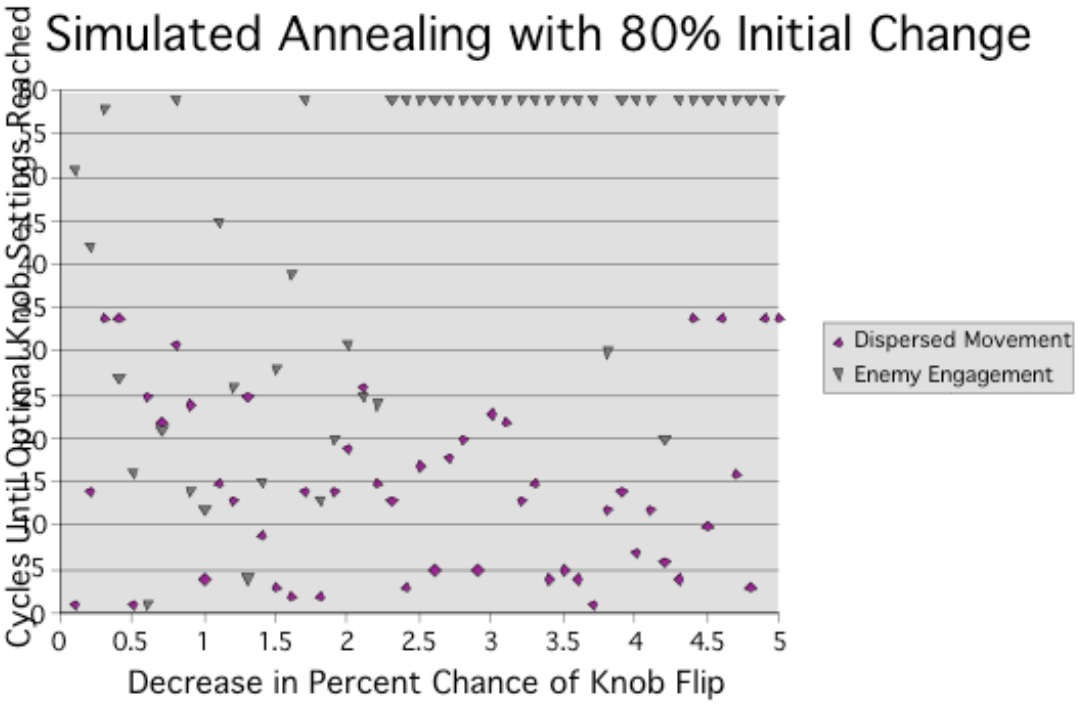


Figure 35

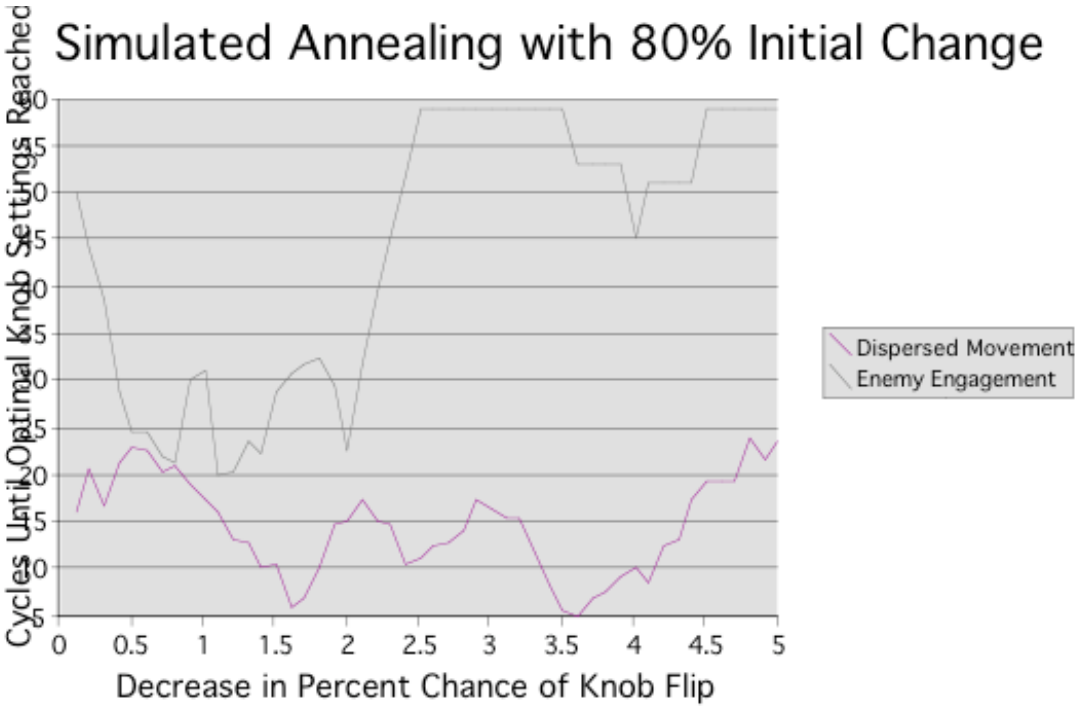


Figure 36

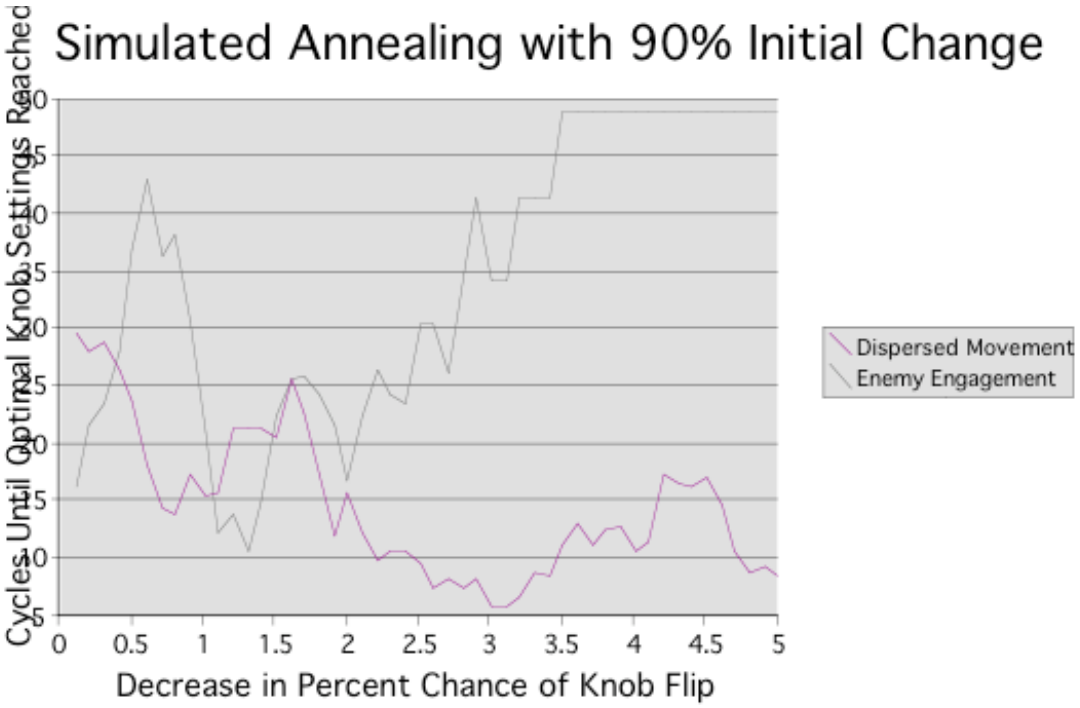


Figure 37

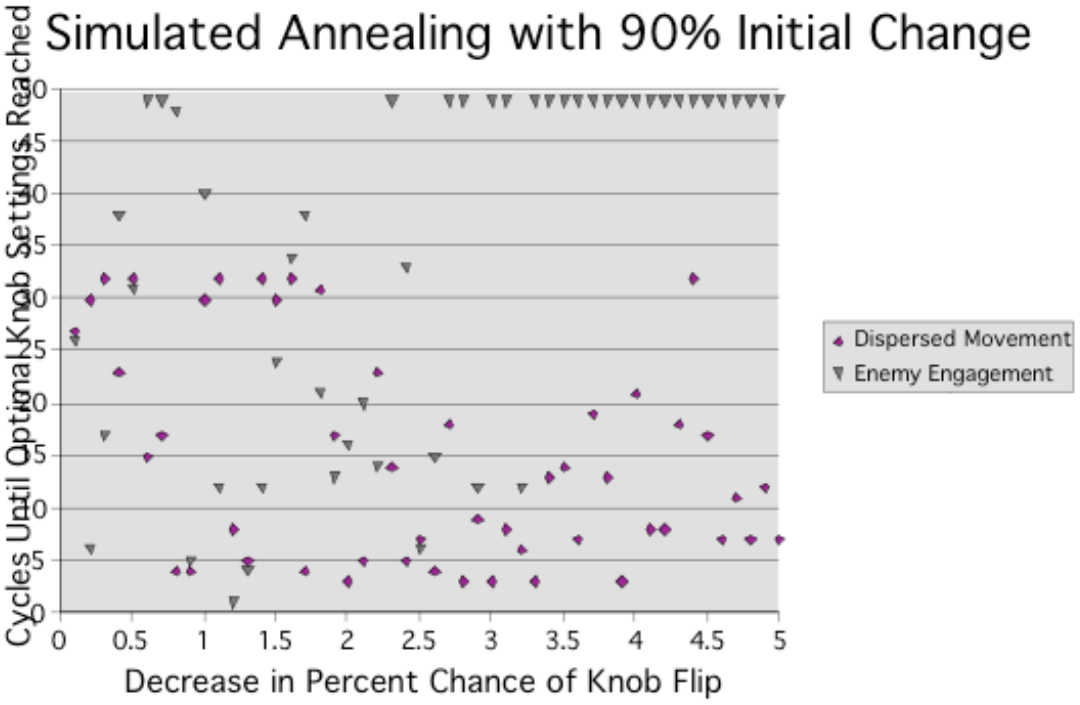


Figure 38

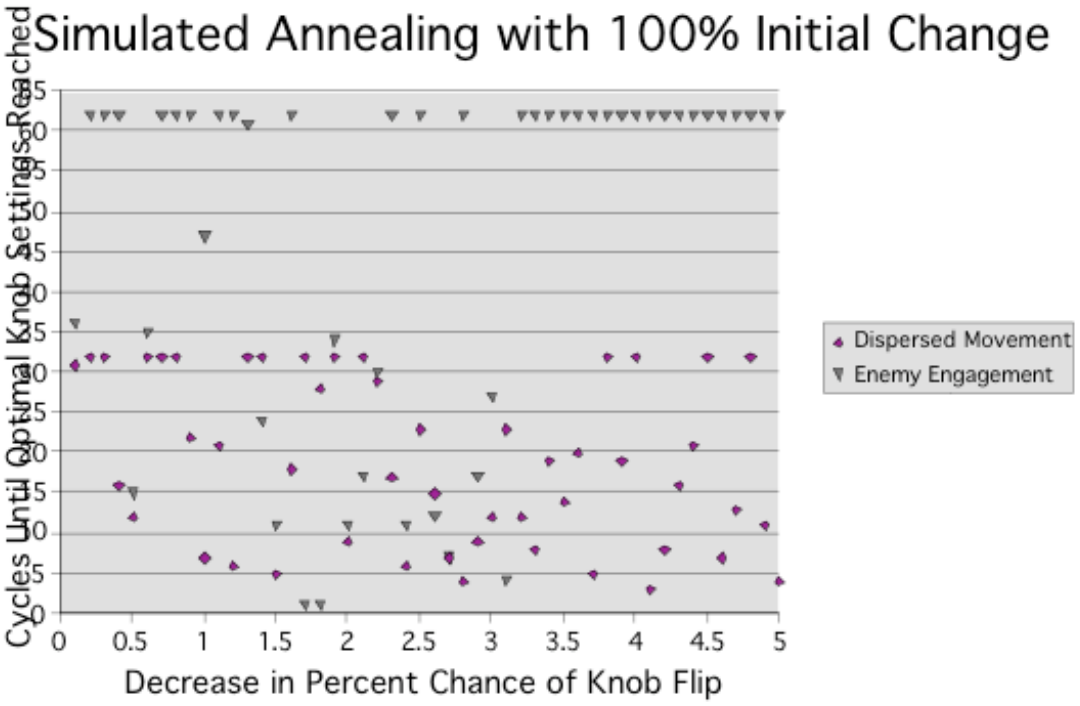


Figure 39

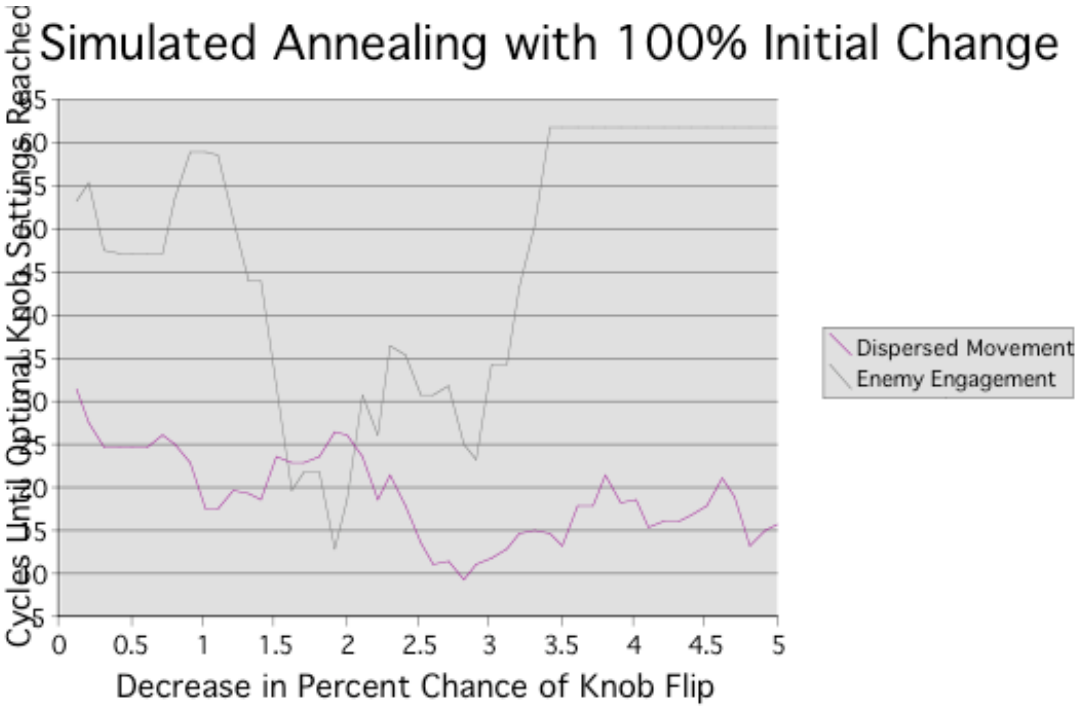


Figure 40

Appendix B: Presentation Slides

Learning in Goal Oriented Autonomous Systems

WPI Project Center :: Silicon Valley 2006

WPI Advisors	SRI Liaisons
David Finkel John Orr	Andy Poggio Carolyn Talcott Grit Denker

Presented By:

Abraão Lourenço, Stuart Floyd and David Casavant

sri06@wpi.edu

Decision Making in Uncertain Environments

- Its simple to make decisions based on expected results.
- In reality, there may be unexpected obstacles or conditions.
- Needs to be autonomous because human interaction is expensive and slow.

2/30

Maude

- Based on rewriting logic.
- This implementation developed at SRI.
- Define states, operations on them, and state transitions.
- Abstract data types: sorts, operator and equations.

```
fmod SAMPLEMODULE is

  protecting STRING .

  op myTerm : -> String .
  eq myTerm =
          "Hello, World." .

endfm


red myTerm .
```

3/30

PAGODA

- Software agents explore the environment to achieve goals.
- Examples:
  - Soldiers' radios adjust automatically.
  - Autonomous robots explore areas that are hazardous to humans.

4/30

PAGODA

- Implemented in Maude.
- Architecture for goal oriented autonomous agents.
- Modular: communication, learning, interaction, etc are handled by pluggable modules.

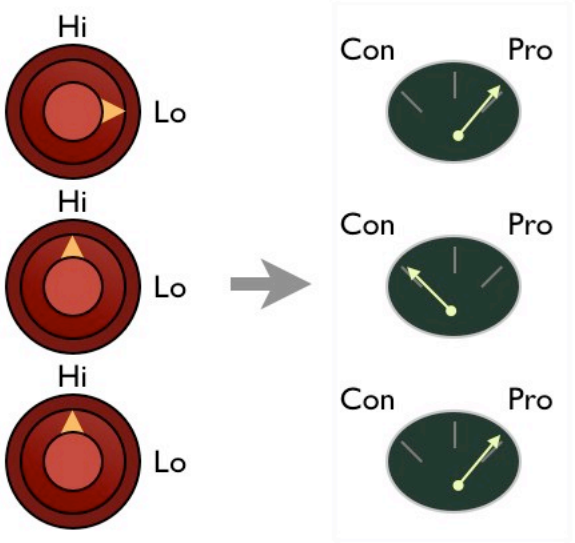
5/30

Phases and Goals

- Phase 1: Parking Lot
  - Goals: Bandwidth, Error Rate
- Phase 2: Dispersed Movement
  - Goals: Connectivity, Resources
- Phase 3: Enemy Engagement
  - Goals: Reliability, Security

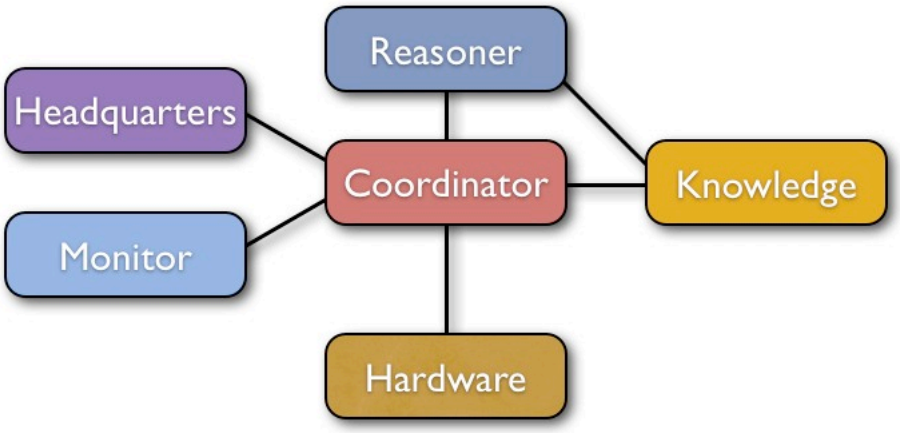
6/30

Knobs and Sensors



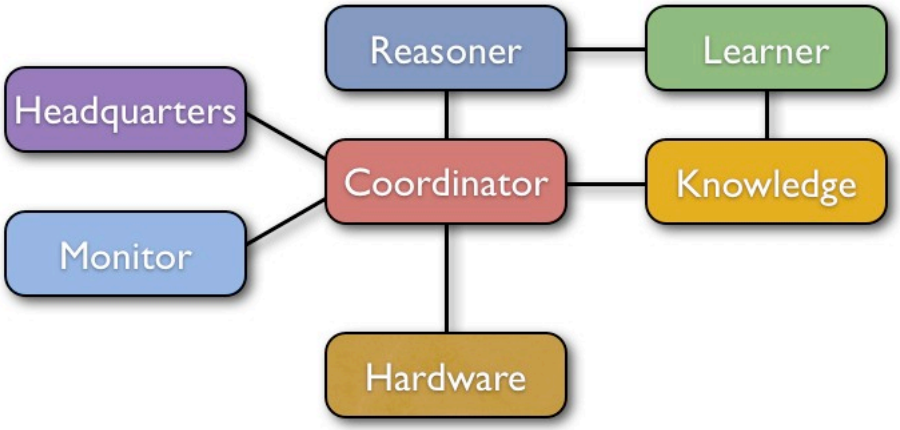
7/30

PAGODA



8/30

PAGODA



9/30

Learner Component

- Learns about the environment by trying experiments and recording results.
- Learns from results according to the algorithm used.
- Can handle adverse or unexpected conditions if the algorithm supports it.

10/30

Learner Component

- Experiments: trying different values for binary parameters. (eg. Power [Lo/Hi])
- Parameters have effects. (eg. Power improves connectivity)
- Sensors detect effects. (eg. Signal strength detects connectivity)
- Goal: adjust knobs to maximize effects.

11/30

Design Decisions

- Implementation of the Learner component.
- Role of the Reasoner.
- Choice of algorithms to implement.
  - Hill Climbing.
  - Inductive Logic Programming.

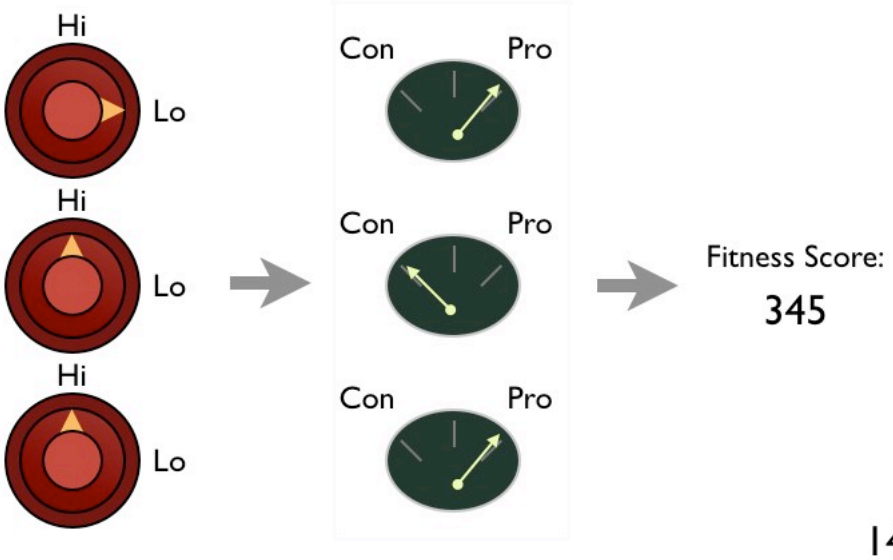
12/30

Hill Climbing Algorithm

- Dependence on the current state of knobs.
- Optimized by fitness function.
  - Steps maximize fitness function.
- Simplicity limits success in some scenarios.

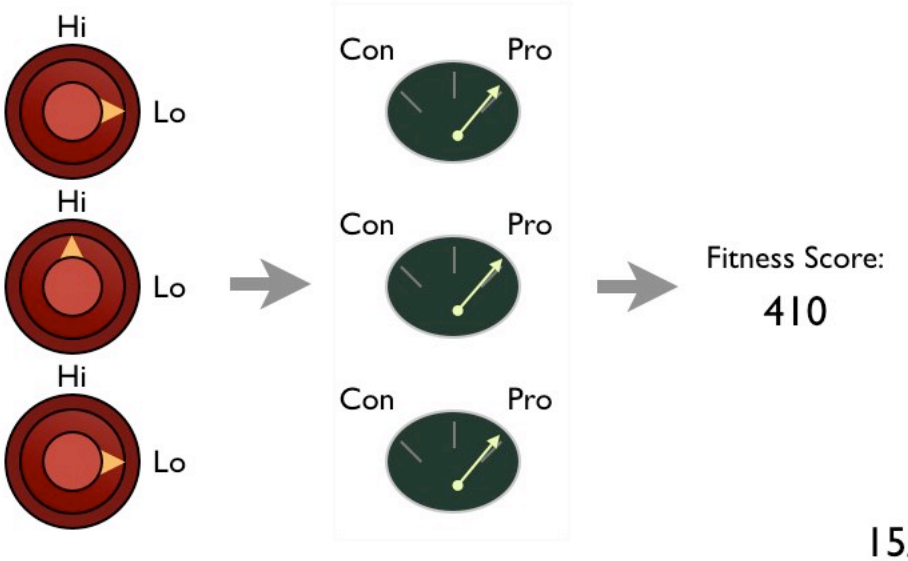
13/30

Knob Fitness



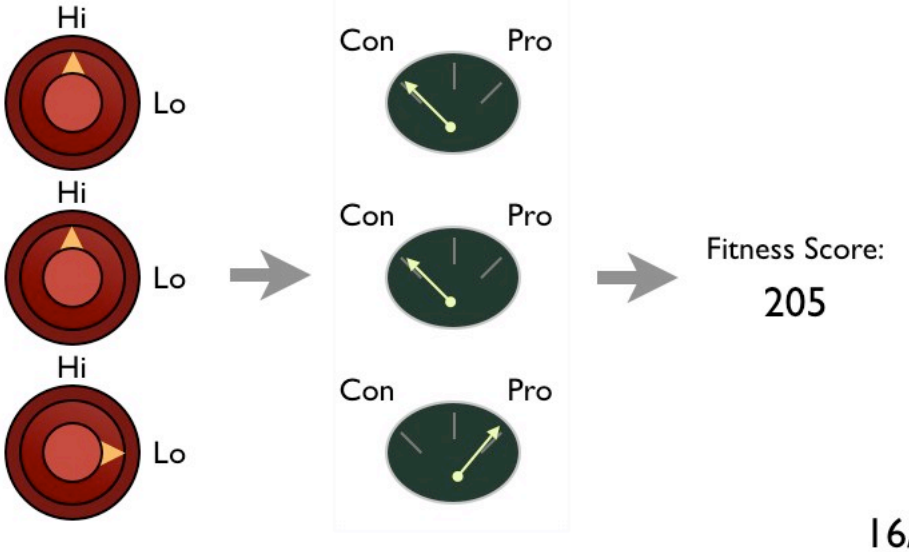
14/30

Knob Fitness



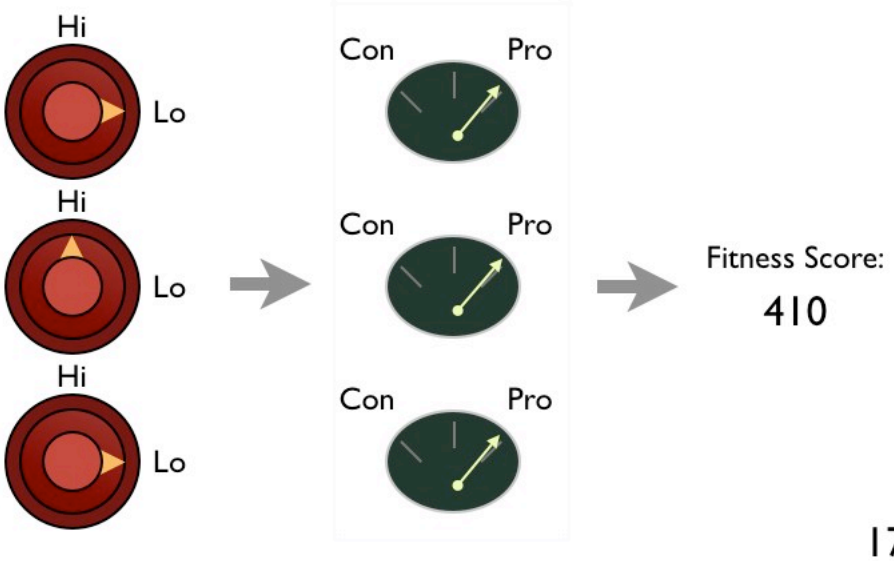
15/30

Knob Fitness



16/30

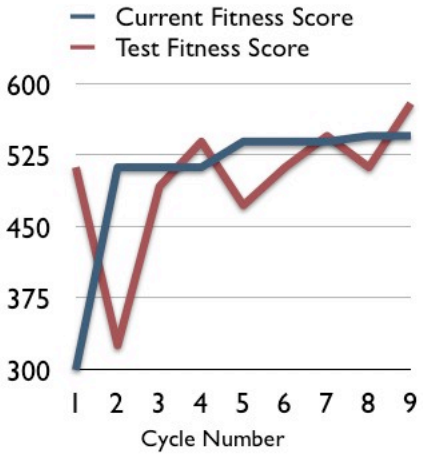
Knob Fitness



17/30

Simulated Annealing

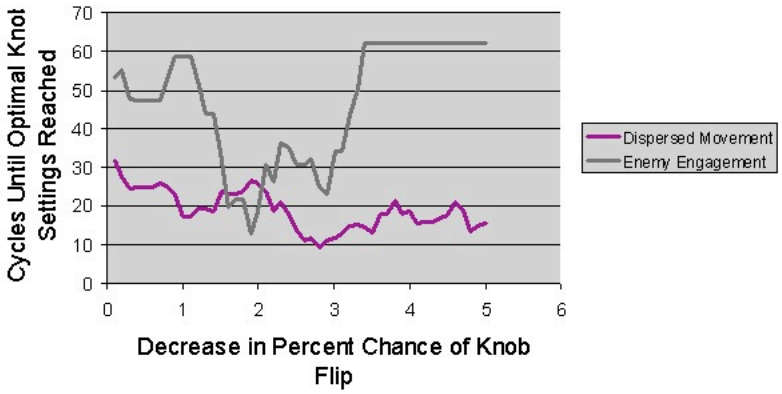
- Starts with a given initial percentage.
- Percentage of change in knobs decreases after each learning cycle.
- Allows the algorithm to handle more complicated situations.



18/30

Hill Climbing Data

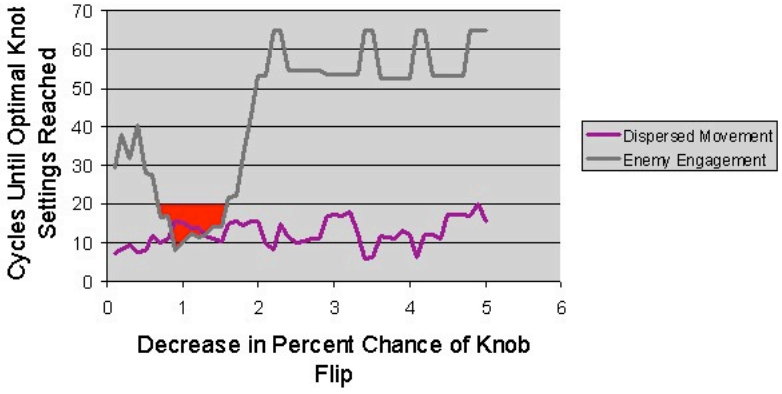
Simulated Annealing with 100% Initial Change



19/30

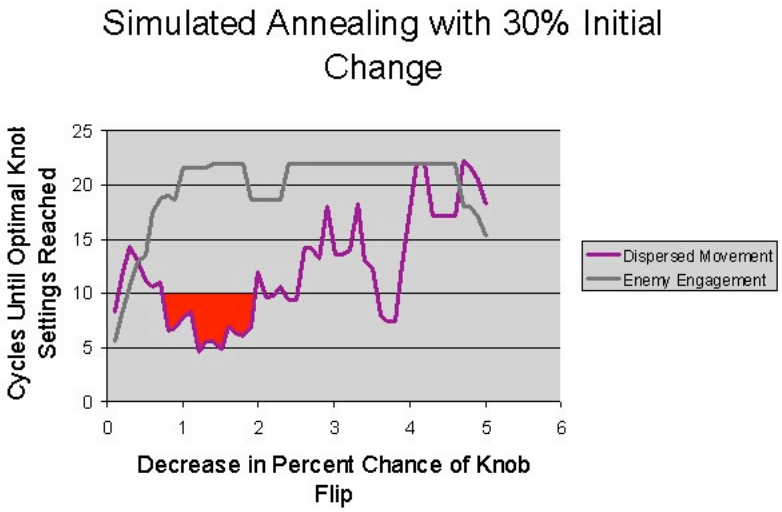
Hill Climbing Data

Simulated Annealing with 50% Initial Change



20/30

Hill Climbing Data



21/30

Hill Climbing Algorithm

- Simulated annealing helps with local maximum.
- Parameters need to be maximized.
- Find solution quickly.

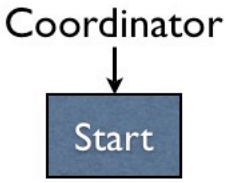
22/30

Inductive Logic

- Three main components:
  - Background knowledge.
  - Positive and negative examples.
  - Hypothesis.
- Runs experiments and records results.
- Analyzes results to produce next guess.

23/30

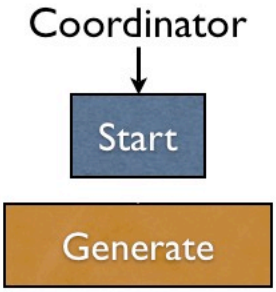
Inductive Logic Algorithm



- Receive start message from Coordinator.
- Fetch background knowledge from the Knowledge Base.
- Generate first set of parameters

24/30

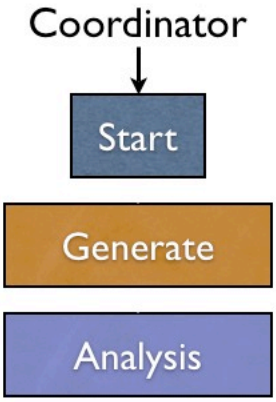
Inductive Logic Algorithm



- Generate set of parameters and record results.
- Maintain best result so far.

25/30

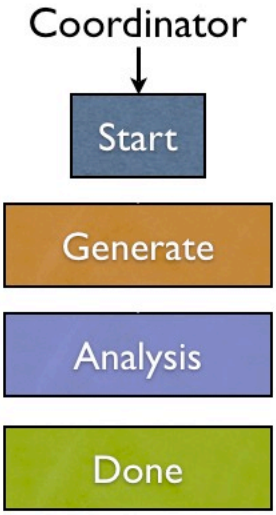
Inductive Logic Algorithm



- Find Con sensors in best result.
- Find another result with equivalent Pro sensors.
- Flip most important knobs.

26/30

Inductive Logic Algorithm



- Done if:
  - Found ideal parameters.
  - Maximum number of experiments exceeded.

27/30

Algorithm Success

Mode	Learning Component	Parking Lot	Dispersed Movement	Enemy Engagement
Normal	Base	3	?	5
	Hill Climbing	3	7	8
	Inductive Logic	3	3	6
Unusual	Base	3	?	5
	Hill Climbing	3	7	7
	Inductive Logic	3	3	6
Faulty	Base	1	?	3
	Hill Climbing	3	6	7
	Inductive Logic	?	?	?
Max Sensors		3	8	9

28/30

Acomplishments

- Added a pluggable learning component.
- Implemented a hill climbing algorithm.
- Implemented an inductive logic programming algorithm.

29/30

Learning in Goal Oriented Autonomous Systems

WPI Project Center :: Silicon Valley 2006

WPI Advisors	SRI Liaisons
David Finkel John Orr	Andy Poggio Carolyn Talcott Grit Denker

Presented By:

Abraão Lourenço, Stuart Floyd and David Casavant

sri06@wpi.edu

Appendix C: Knob Table

The table below shows the relationship between knobs and effects under the hardware's normal mode of operation. A plus sign indicates that the knob contributes to that effect, a while negative sign means that the knob detracts from that effect [PAGODAdoc]. This table is part of the documentation distributed with PAGODA.

	A	B	C	D	E	F	G	H
1	**Effect Category:**	improved	minimal resources	high bandwidth	low ER	diffserve	security	reliable
2		connectivity	(power, memory, cycles)	(effective)	(bit or pkt)	(priority enforced)		delivery
3	**Knob:**							
4	transmission power (high)	+	-					
5	" (low)	-	+					
6	transmission freq. (high)	-		+				
7	" (low)	+		-				
8	unused channel selection	+						
9	packet/fragment size (high - large)			+	-			
10	" (low - small)			-	+			
11	compression (high)		-	+				
12	" (low)		+	-				
13	node movement	+						
14	transport protocol (hi - large window)		-	+				
15	" (low - small window)		+	-				
16	error correction (high)		-	-	+			
17	" (low)		+	+	-			
18	routing algorithm (high - fast update)	+		-				
19	" (low - slow update)	-		+				
20	queueing retention (high - no drops)					-		+
21	" (low - low priority drops)					+		-
22	encryption level (high)		-				+	
23	" (low)		+				-	
24	cluster composition (clusters)	-		+				
25	" (no clusters)	+		-				
26	caching (high - on)	+	-					+
27	" (low - off)	-	+					-
28	soft state degree (high)	-		+				
29	" (low)	+		-				
30	directional rf transmission (high)	-					+	
31	" (low - nondirectional)	+					-	
32	casting specificity (high - uni)			-				
33	" (low - broad/multi)			+				

Appendix D: Sensor Table

The table below shows the relationship between sensors and effects under the hardware's normal mode of operation. A check mark indicates that the sensor is used to detect the effect [PAGODAdoc]. All sensors associate with an effect are usually used to detect that effect. This table is part of the documentation distributed with PAGODA.

	A	B	C	D	E	F	G	H
1	**Effect Category:**	improved	minimal resources	high bandwidth	low ER	diffserve	security	reliable
2		connectivity	(power, memory, cycles)	(effective)	(bit or pkt)	(priority enforced)		delivery
3	**Sensor:**							
4	Local							
5	location	x					x	
6	battery level		x					
7	processor idle percentage		x					
8	ram consumption		x					
9								
10	Pairwise (e.g. A transmits to B; B senses the following)							
11	signal strength	x						
12	bit error rate				x			
13	packet loss rate				x			
14	interference level	x						
15	multipath	x						
16	connectivity loss	x						
17								
18	Global (most or all nodes in aggregate):							
19	traffic priority					x		
20	throughput demands / traffic patterns			x				
21	throughput success assessment							x
22	environmental conditions (e.g., weather, terrain)	x						
23	node loss	x						