

# (Experiment) Designing a Generic Unsupervised Learning Component: Knowledge.dll

---

—

George Fisher,  
December 2005,  
Windsor, Ontario, Canada

# (Experiment) Designing a Generic Unsupervised Learning Component: Knowledge dll

## Contents

<b>OBJECTIVES</b> .....	<b>3</b>
<b>THE KNOWLEDGE SYSTEM</b> .....	<b>3</b>
OVERALL DESIGN.....	3
<i>The List Memory</i> .....	3
<i>Actions</i> .....	4
<i>Recognition of Win Situation</i> .....	4
<i>Outline of Strategizing</i> .....	4
CODE DESIGN.....	4
<i>Workings of the public Knowledge::stateUpdate function</i> .....	5
<i>Workings of the public Knowledge::effectRegistration function</i> .....	5
<i>Workings of the public Knowledge::move function</i> .....	5
<i>Workings of the Knowledge::pickBestFit function</i> .....	6
<i>Workings of the Knowledge::similar function</i> .....	9
<i>Workings of the Knowledge::nextAction function</i> .....	11
<b>ROOM SCENARIO GAME</b> .....	<b>12</b>
MAP LAYOUT.....	13
GAME SCREENSHOTS.....	14
GAMEPLAY.....	14
EXACT DETAILS.....	15
CONNECTION TO THE KNOWLEDGE DLL.....	15
<b>PERFORMANCE</b> .....	<b>16</b>
IN THE MACHINE ROOM EXPERIMENTAL GAME.....	16
<i>Gameplay/Entertainment Value (observations of gamers)</i> .....	16
<i>Sample Winning Strategies</i> .....	16
<i>Sample Learning Process</i> .....	17
<i>Experiment</i> .....	17
<i>Results of Experiment</i> .....	19
<i>Experimental Conclusions</i> .....	22
<b>CONCLUDING REMARKS</b> .....	<b>23</b>
WEAKNESS OF THIS AI.....	23
SELF AWARENESS.....	24
SUCCESS OF UNSUPERVISED LEARNING.....	24
SUCCESS OF THE LINEAR MEMORY.....	25
ENTERTAINMENT VALUE AND PRACTICAL USE IN COMPUTER GAMES.....	25
FINAL REMARKS.....	25
<b>APPENDIX</b> .....	<b>26</b>
TWO SECOND TOTAL PLAN EXPERIMENT DATA TABLE.....	26
THREE SECOND TOTAL PLAN EXPERIMENT DATA TABLE.....	26
<b>BIBLIOGRAPHY</b> .....	<b>27</b>

# 1. Objectives

To discover, through experimentation, the foundations of AI leading to software that appears to learn like a human being and as an eventual aim, to understand consciousness. Also considered here is the application of AI in video games, and what makes good videogame AI.

Artificial intelligence too often refers to computer programs that solve problems that are not mathematical in nature, or that were traditionally solved by human beings. Here, I would like the aim of AI to include skilled computer programs, but also to look beyond that structured approach at the idea of creating life, or with regard to video games, activity that feels intelligent (has apparent consciousness), rather than feeling like a series of optimized equations.

This experiment focuses on unsupervised learning – to form natural groupings or clustered patterns without explicit instruction. The ultimate aim is to create a system that can gain insight into the nature or structure of data independent of any former background knowledge. In particular to create a problem solving component that can then be applied to all sorts of situations.

Another aspect to be explored in this project is the idea of a linear memory, which does not categorize data in any way but stores a list of experiences and from that it derives the next step, by comparing the current sensory data with the data already collected.

More specifically, once the creation of the software is complete, it should be applied to a game in a 3d environment to judge its performance as both a problem solving system and a source of entertainment.

# 2. Knowledge dll System

## OVERALL DESIGN

With simplicity in mind, the Knowledge dll is the generic AI component designed for this project. It is based on two seemingly flawed premises. These premises were chosen because they would lead the project to bring about a greater understanding of AI. They are also easily to implement. These premises are:

- ✘ Experimenting with random behavior will lead to understanding
  - Flawed, because apparent random behavior comes through a enormous collection of facts gathered by humans over a time of many years. In short, humans are far worse than computers are generating anything random
- ✘ The best way to solve a problem is have some rigid rules to follow
  - Flawed because it does not lead to generalization, but workable when we have a very small dataset

However, these have been implemented to create a simple learning system that is quite effective, as will be revealed.

## The List Memory

The Knowledge dll works by being provided with a history of experience, and from that history, deriving what the best action to take to reach a “win state”. Therefore it needs to be provided with information about the environment and opportunities to respond to this information with actions of its own.

## Actions

This AI component does not learn from observation, but by experimentation. For example, if an operation is performed and that information is reported back to the AI as part of its observations, the AI cannot learn from that. Instead it has a limited number of actions that it can experiment with, and from the consequences of that experimentation and based on sensory data, it can derive sequences of response that will lead to win states.

## Recognition of Win Situation

When the AI has won or lost (succeeded or failed), the software using the AI reports to the Knowledge dll whether it has won or lost. This becomes important as the list memory grows larger, as the system can base its plans for success, and guesses (when it has no plans) on its previous experience, i.e. where in the past it succeeded or failed.

## Outline of Strategizing

A more detailed view of the exact algorithm used is presented below in the “code design” section.

All the strategizing and planning takes place when the system is asked to make a move. When the AI makes a move it follows these steps:

1. It calculates the best plans (based on previous experience of wins and losses)
2. It finds which plans are most appropriate for the current situation (based on the observations of the world the AI is applied to)
3. It selects the most appropriate, most effective plan
4. It picks the next action from that plan

If there is no data for the system to learn from, the AI picks a random choice and takes a record of it.

## CODE DESIGN

The functions shown here are only those that contribute to the AI logic. All others are concerned with data structures and are not explained here.

The public functions that are concerned with AI, and are presented first are:

- ✂ **stateUpdate** – the function to store any environmental data (the statement of the world) reported from the host application
- ✂ **effectRegistration** – the function to inform the AI that a state affecting the AI, such as a win or loose has been reached
- ✂ **move** – the function in which the AI is allowed to calculate a new move. It returns an action ID to be executed by the host application.
  - Calls **pickBestFit** function, which calls the **similar** function. Then **nextAction** is called the find the next action in the plan.

There are two major variables that are used repeatedly in these functions:

- ✂ *shortHistory* – stores all actions and events that have happened in the game since the last win or lose (everything from the current game).
- ✂ *log* – contains the list of all actions, events, and effects for the entire list memory.

## Workings of the public Knowledge::stateUpdate function

### Summary

The stateUpdate function is called to inform the AI that something has happened in the world it is interacting with. This could be a visible action, a movement, a location, or any information that the AI should base its understanding of the environment upon. Once this information arrives, the data is added to the list-memory

## Workings of the public Knowledge::effectRegistration function

### Summary

The effectRegistration function is called to inform the AI that a state affecting the AI, such as a win or loose has been reached. If the effect registered is -2, this indicates a win, a -1 indicates a loss. At this point, this information is not considered at all – it is simply added to the list memory.

## Workings of the public Knowledge::move function

### Summary

Calling the **move** function allows the AI to make a move, and it will return an action ID which is the action it wants to be executed.

### Technique

```
int noClearAction = 0;
int response = -1;
ostringstream ss;
```

Based on the environmental data collected since the AI last won or lost (shortHistory), the best plan to reach a win state is chosen.

```
plan = pickBestFit(shortHistory);
```

If a plan wasn't found, it is recorded as such, otherwise, the chosen plan is compared to the current history, the next step is chosen from the plan, which is then stored in the variable response.

```
if (plan.length() == 0)
{
    noClearAction = 1;
}
else
{
    response = nextAction(shortHistory, plan);
}
```

If no action was found, or no good step was found in the plan, we pick a random action. A potential addition here is the avoidance of plans that have failed before.

```
if (noClearAction || response == -1)
```

```

    {
        response = actions[(rand() % actionTotal)];
        //response = checkNotPartOfFailedStratergy(response);
    }

```

The response is replaced with a random choice every 3/100 times. This is done to encourage the system to experiment with different ideas even when it has a solution. Ideally, this kind of thinking should occur based on environmental data alone, and if this project is extended, this will be one of the first parts of the algorithm to be altered.

```

    if ((rand() % 100) < 4) // && certainty == LOW
    {
        response = actions[(rand() % actionTotal)];
        //response = checkNotPartOfFailedStratergy(response);
    }

```

A record is kept of the chosen response, and the action code is returned to the application using the AI.

```

addItem(response);
return response;

```

## Workings of the Knowledge::pickBestFit function

### Summary

The **pickBestFit** function takes the history since the last win or lose state (or game start), stored in *shortHistory*, and finds the best solution based on successful and failed plans already experienced.

*Note: Obviously this function could be much faster if it simply maintained the data already collected in tree form.*

### Technique

#### Received data

*shortHistory* - the actions and events occurring during the current game (since the last win or lose event).

This function begins by looping through the number of lines in the entire history (all games ever played) in the vector array *log*.

```

int scout = 0;

for (int i = 0; i < itemCount; i = i + 1)
{

```

Are we analyzing a place in the log where the game was restarted? (-1 is the restart/lose code - the human player has won)

```

    if (log[i] == -1)
    {

```

As the game we are studying is a unsuccessful one (-1 indicates a loss) we score negatively against this and all strategies that we have already discovered, that are similar to this one.

```
noNewStrategy = 0;
```

Search through all the strategies that we have found so far.

```
for (int j = 0; j < scout; j = j + 1)
{
```

If we find a similar strategy, we subtract two<sup>1</sup> from its score. This means that if we have seen a strategy win and then loose, we make sure it is not used again. This is the adaptability functionality.

The logic here for finding a similar strategy is:

*(strategy under investigation) IS SUBSTRING (strategy in list) AND NOT((strategy in list) IS SUBSTRING (strategy under investigation))*

```
if (similar(trackStrategy,
            workableStrategyDefns[j], 1, 0))
{
    workableStrategyScore[j] =
        workableStrategyScore[j] - 2;
}
```

We reset our memory of the current strategy, as we don't need to remember failed strategies unless they (above) are failed versions of successful strategies.

```
trackStrategy = "";
```

```
}
```

Have we reached a win state in the current strategy?

```
else if (log[i] == -2)
{
```

As the strategy we have found is a successful one, we need to remember it. However, if we have already encountered the winning strategy then encountering it again increases that strategy's value (as winning more than once is a great achievement)

```
for (int j = 0; j < scout; j = j + 1)
{
```

If we find a similar winning strategy, we add one to its score.

The logic here for finding a similar strategy is:

*(strategy under investigation) IS SUBSTRING (strategy in list) AND NOT((strategy in list) IS SUBSTRING (strategy under investigation))*

```
if (similar(trackStrategy,
            workableStrategyDefns[j], 1, 0))
{
```

---

<sup>1</sup> A bird in hand is worth two in the bush (Anon)

```

        workableStrategyScore[j] =
            workableStrategyScore[j] + 1;
    }
}

```

The new strategy is added to the list even if a similar strategy is found. The reason for this is that the **similar** function finds similar strategies – if this strategy is more efficient, it should be kept even if it is similar to a winning strategy (it could be better).

```

workableStrategyDefns.push_back(trackStrategy);
workableStrategyScore.push_back(1);
scount = scount + 1;

```

We clear our memory of the current strategy.

```
trackStrategy = "";
```

```

}
else
{
}
}
}

```

If the state is neither a win or loose, we simply add the data to the current study of the strategy (strings are used as a legacy of the Perl version of this software).

```

trackStrategy.append(intToString(log[i]));
trackStrategy.append("|");

```

Now that all the winning strategies have been found and have been scored, we need to sort them all.

```

sortedStrategies = workableStrategyScore;
sort(sortedStrategies, sortedStrategiesIndex);

```

(pseudocode)

We search through the sorted strategies until we find one that's suitable, then we return it.

```

int score, index;
for (int i = 0; i < scount; i = i + 1)
{
    score = sortedStrategies[i];
    index = sortedStrategiesIndex[i];

    Does the strategy work? More specifically, does it map to our current shortHistory? If
    so it's usable, and we return it.
    if (works(shortHistory, workableStrategyDefns[index]) &&
score > 0)
    {
        return workableStrategyDefns[index];
    }
}

```

Otherwise we return the strategy as empty.

```
return "";
```

## Workings of the Knowledge::similar function

### Summary

The **similar** function takes two strategies stored in strings s1 and s2, and then compares how different they are from one another. Cost1 indicates the price of a difference in string s1, and cost2 indicates the price of a difference in string s2.

### Use of Function

This function is useful when comparing strategies (stored as lists), especially if we throw in random elements. This allows a strategy with a slightly different step to be counted the same as another strategy that we have some history on. It means we can take partial knowledge and assume it's the same as something we already know.

### Technique

#### Received data

s1 - list of actions  
s2 - list of actions  
cost1 - the cost of a move in string s1  
cost2 - the cost of a move in string s2

First, a reject difference is set: this is, the maximum difference between the two lists of actions before we reject them outright as being different:

```
// a third of the length seems to be a good scale for difference  
int maxdiff = items / 3;
```

**C++ code**

Then, the function then enters a loop through the actions in list s1 and list s2. In each iteration of the loop the following logic is executed:

1. Is the difference greater than the maximum acceptable difference? If so, the two lists are not similar

```
if (difference > maxdiff)  
{  
    return 0;  
}
```

**C++ code**

2. Are we at the end of either of the two lists and we have actually traversed something? (one list could be empty - the other might have lots of items in it) Then the two lists are similar.

```
// are we at the end of either of the lists?  
if ((s1i >= items) || (s2i >= items2))  
{  
    if ((difference <= maxdiff) && !foundNothing)  
    {  
        // yes these two are similar  
        return 1;  
    }  
    // otherwise exit loop (the strings are not similar)  
    slide = 0;  
}
```

C++ code

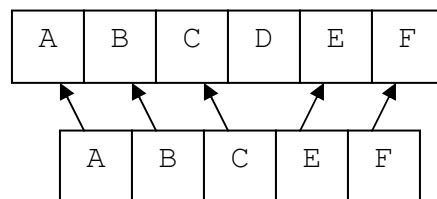
3. Are the two list items the same? Then the point of synchronization is remembered, and we shift along in both lists

```
if (s1L[s1i] == s2L[s2i])
{
    difference = difference + 0;
    lastSync1 = s1i;
    lastSync2 = s2i;
    indexer = 0;
    foundNothing = 0;

    s2i = s2i + 1;
    s1i = s1i + 1;
}
```

C++ code

4. Are the two list items *not* the same? Then we shift one space along list two. If we're shifting more than the maximum difference, or off the end of the list, we go back and start shifting along list 1. This is to allow desynchronized lists:



**maxdiff** = 6 / 3 = 2

Therefore, these two lists are similar (D on list 1 was skipped - if the cost1 is 1)

```
else
{
    // we skip one on s2
    s2i = s2i + 1;
    indexer = indexer + 1;

    // If we're finding no match on our sliding
    // scale, then we backtrack and study comparison
    // the other way.
    if (indexer > maxdiff || indexer > items2)
    {
        s1i = lastSync1 + 1;
        s2i = lastSync2;

        // not strictly true (not last sync) but
        // stops infinite loops
        lastSync1 = s1i;
        lastSync2 = s2i;

        // we allow substrings (up to cost
        // maxdiff)
        if (!foundNothing)
        {
            difference = difference + cost1;
        }
    }
}
```

```

        }
    }
    else
    {
        // we allow substrings in
        if (!foundNothing)
        {
            difference = difference + cost2;
        }
    }
}

```

C++ code

## Workings of the Knowledge::nextAction function

### Summary

The nextAction function is provided with the history of actions in the current game, and the plan for the game. By comparing these two lists, it selects the next action to be performed.

### Use of Function

This function is called from the move function to determine what step should be taken next.

### Technique

#### Received data

*shortHistory* - the history of actions in the current game  
*plan* - a tried and tested plan for the current game

#### Break apart the *shortHistory* and the *plan*

```
StringTokenizer shSTOK = StringTokenizer(shortHistory, "|");
StringTokenizer pSTOK = StringTokenizer(plan, "|");
```

```
vector<int> sh = vectorize(shSTOK);
vector<int> p = vectorize(pSTOK);
```

```
int pval;
int pi = 0;
int shi = 0;
int lastPSync = 0;
```

Go through the short history and the plan at the same pace, until we reach a point on the plan which hasn't been executed.

```
while (shi < (int)sh.size() && pi < ((int)p.size() + 5))
{
    // this exists to allow us to overrun a bit
    if (pi < (int)p.size())
    {
        pval = p[pi];
    }
    else
    {

```

```

        pval = 0;
    }

```

If something matches on both the short history and the plan we move forward one space along each one. We also remember that this was a point of synchronization.

```

if (sh[shi] == pval)
{
    lastPSync = pi;
    pi = pi + 1;
    shi = shi + 1;
}
else
{

```

If something doesn't match on either one, we skip forward on the plan.

```

pi = pi + 1;

```

If we have not matched and we are already two spaces off from the existing plan, we sync back to the plan.

```

// we map back a bit
if ((pi - shi) > 2)
{
    pi = pi + 1;
    // not really a sync but good enough
    lastPSync = pi;
}
}

```

The synchronization synchronizes all elements, we are only interested in the next action we can perform. Therefore, we move forward from the last sync point and find the next action.

```

for (int m = lastPSync; m < (int)p.size(); m = m + 1)
{
    Check if the plan element is an action, if so, return it

    if (isAction(p[m]))
    {
        return p[m];
    }
}

```

Otherwise we return -1 which means there's a problem.

```

return -1;
}

```

### 3. Room Scenario Game

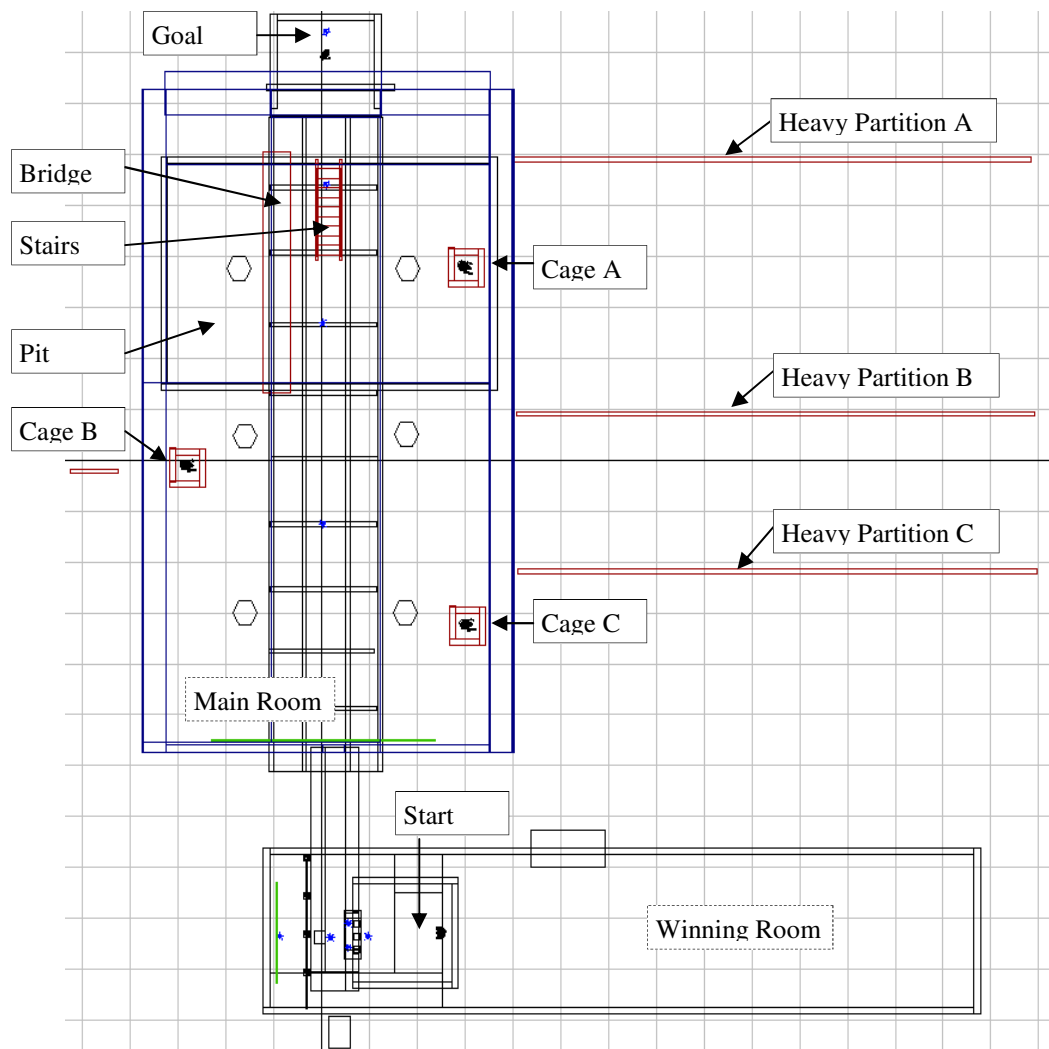
To create both grounds for experimentation and demonstration a game environment was designed and built in 3d game studio was designed to both immediately and clearly bring to life the operations and thinking of the Knowledge dll.

The results from this experiment are described in part four.

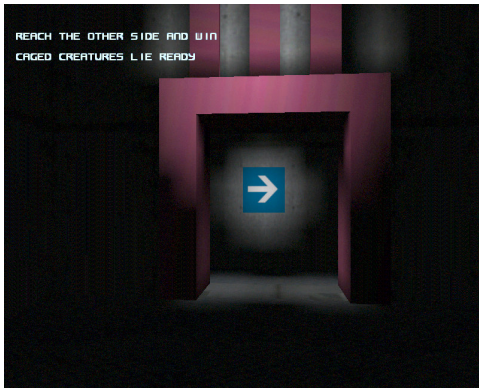
The game is a large room filled with various 'machines' that can be operated by simple commands – such that they may be operated by the Knowledge dll. These machines are various obstacles that the human player must get past. These include:

- ⌘ 3 x heavy partitions
- ⌘ 3 x caged monsters
- ⌘ 1 x stairs in put
- ⌘ 1 x bridge over pit

### Map Layout



## Game Screenshots



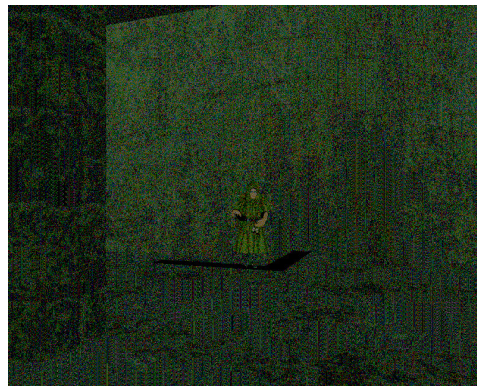
The opening view (gamma+160%), turning right takes the player in the large room



View of the main room (gamma+160%)



The bridge in motion and the stairs in rest position (gamma+160%)



A creature emerging from her cage (gamma+370%)

## Gameplay

The goal of the player is to successfully reach the other side of the main room. The Knowledge dll has the freedom to operate the machines at set periods. The partitions, stairs, and bridge are all components that can be used to prevent the player from advancing across the room. The cages each contain creatures that are programmed to follow the player until they catch him or her.

When the partitions, bridge or stairs commands are executed, each is closed/moved, but only for a limited period of time.

If a player is caught, then they have lost. If a player crosses to the other side of the room, they have won.

Whenever a win or lose state occurs, the game is reset to the beginning, and the Knowledge dll is informed.

Therefore the challenge for the Knowledge program is to identify strategies that will successfully hurt the player. For example, closing a partition and letting a creature loose. The player, having nowhere to run, will be caught.

### Exact Details

Partition A (id: 1001), B (id: 1002), and C (id: 1003)

Closes in ~2.525 seconds  
(fast to deal with catching the user before they reach the partition)  
Stays closed for ~22.5 seconds  
Opens in ~4.75 seconds

Bridge (id: 1008)

Closes in ~9.25 seconds  
Stays closed for ~22.5 seconds  
Opens in ~9.25 seconds

Stairs (id: 1004)

Moves in ~ 1.875 seconds  
Stays closed for ~22.5 seconds  
Moves back in ~5.625 seconds

Cage A (id: 1005), Cage B (id: 1006), and Cage C (id: 1007)

Opening time: ~5 seconds

Walking time

Partition to partition is approximately ~3 seconds (enough time to close a partition before the player reaches either the first or the second).

### Connection to the Knowledge DLL

Data can be passed to the Knowledge dll at different points of the game, changing the way it learns. The quantity of information fed to it also changes the complexity and skill involved in the learning process.

There are two techniques which define different techniques the game uses for learning.

Event State Reaction

The Event State reaction allows the Knowledge dll to operate two machines every time the player enters a new region of the room. This means that there has to be some relation of where the user is to exactly what devices are controlled.

Five Second Total Plan

In this state the software is only guessing sequences of events that will lead to a win-state.

- ✘ Every five seconds the Knowledge dll is permitted to make a move in the game.

- ✕ No feedback is recorded except a win or lose state.

## 4. Performance

### IN THE MACHINE ROOM EXPERIMENTAL GAME

**We first look at the entertainment value of this AI component, and then we extend this to a more formal investigation as to the capabilities of the AI.**

#### **Gameplay/Entertainment Value (observations of gamers)**

It is not entirely possible to quantify the success of the Knowledge dll component based on gameplay, as this depends upon many other factors, most importantly the design of the game itself. However in tests of gamers aged 14-19 the entertainment value of trying to continuously outwit the computer was enough to keep each person wanting to play for about 20 minutes – a considerable success for such a simple game. The game was less appealing to women, who were more excited by the fear caused by the darkness and sound effects of the sample game environment.

The most appealing aspect of the AI to the players who played was how the Knowledge dll was able to come up with different winning strategies each time, and did not resort to anything that was recognizable. It seemed a more human AI – it was more playful.

In the room game the Knowledge dll is very effective because:

- There are not very many choices of actions to take, and it is simple to find winning combinations
- The player is unpredictable, therefore the ability to reject failed strategies and then to adapt becomes very useful

When controlling the machines in the room, the random experimentation is sufficient to find a combination that will prevent the player from winning. The more that the game is played, the more skilled the software becomes. As it is relatively easy to

When a player first attempts the game, they are not completely familiar with the objects in the room and therefore tend to be easily catchable at the beginning, causing the AI to reject strategies that worked when the player was more cautious; this extended the game's interest further, as the player learned how to outwit the traps controlled by the AI.

#### **Sample Winning Strategies**

These strategies were derived automatically by the Knowledge dll.

A simple winning strategy

1002 (first partition)

1007 (first cage)

...

Closing the first partition and opening the first cage causes the user to be trapped in with the let-loose creature.

Another strategy (more foolproof)

1004 (stairs)

1002 (middle partition)  
1006 (middle cage)  
1005 (first cage)  
1009 (no operation)  
1007 (third cage)  
...

The first move here, the stairs, has no effect on the player - the Knowledge dll is unaware of this, but doesn't care, because this strategy works anyway.

### Sample Learning Process

This is a real example of the learning process using the "total plan" approach where the location of the room is always reported to be the same.

The first attempt the AI tried the following actions (each one after 5 seconds):

1006, 1004, 1005, 1009

And the game responded:

-1 (user win)

The second attempt the AI is luckier:

1002, 1007, 1009, 1001

And the game responded

-2 (user dead)

So it tries it again:

1002, 1007, 1009, 1001

But nothing happens, so it starts to pick randomly:

1004, 1005, 1002, 1009, 1006, 1002, 1008, 1003

And the game responded:

-2 (user dead)

Now, the user goes again, and the AI tries

1002, 1007, 1009, 1001, 1004, 1005, 1002, 1009, 1006

And the game responded:

-1 (user win)

This time the user has won against a strategy that works. It's now unlikely that this strategy will be used again. The four actions \*may\* be used as it reoccurs in both two previous successful attempts.

In this example the location in the room was not factored in - this simply adds places where the strategy branches off depending upon the room location data.

### Experiment

#### Process of Evaluation

To demonstrate the use of the Knowledge dll to game programming it is necessary to work with it in a game environment. It would be possible to provide fixed input data, or use the console version to see how quickly the software can consistently win. However, this does not take into account the very slight variations in timing by a human player in a computer game which can enormously change an outcome. For example, if a player can dodge a creature by one meter in a game environment, how can this be factored into a scientific evaluation of the learning capabilities?

The only practical solution is the repeated testing of the Knowledge dll in the Machine Room Game, played to very strict behaviors. In addition, the visual environment allows for a greater insight into the thinking of the AI, and awareness of what possibilities exist for its improvement.

The theorem I base this on is the idea that the challenge of practical application gives rise to more questions than a theoretical example typically presents.

#### Strict Rules of Human Play

- ✘ The evaluation proceeds until three consecutive losses occur. At this point the game is considered to be 'mostly unbeatable'. Three consecutive losses indicate that the system has found a strategy that allows it to consistently win the game. These wins must be based on the success of the strategy not based on bad luck.
- ✘ Play must be as aggressive as possible – the creatures must be dodged as well as possible, by letting them follow across the space and then moving around the edge of the rooms. When repeatedly caught on the stairs, yet the bridge is unaffected, the bridge must be tried out.
- ✘ Always advance across the room, never hold back (it doesn't help anyway)
- ✘ Always choose to cross the bridge rather than take the steps
- ✘ Always climb steps to avoid creatures if the bridge is dangerous or down

#### Data Collected

The data that has been collected includes:

- ✘ The **number of trials** is the number of wins and losses before the 'mostly unbeatable' state occurs.
- ✘ The **number of wins** until the 'mostly unbeatable' state occurs.
- ✘ The **number of losses** until the 'mostly unbeatable' state occurs
- ✘ The **number of steps** in the winning strategy
  - From the three consecutive losses before the 'mostly unbeatable' state, of the last two, the number from the strategy with the lowest count of steps is used. This is because the first time the strategy is seen the user is taken by surprise, and it is likely the number of steps is incorrect.
- ✘ The total **number of operations** (machine movements) run in the level, to reach the 'mostly unbeatable' state.

#### Potential Inferences

The number of wins and losses indicate the level of strategy of the system. If the number of losses is only three, the AI is less interesting to the player, because the behavior is simply random until a successful strategy is found. If the player learns to dodge one strategy, the system must adapt and restart experimentation. It is that adaptation that is entertaining and makes the 'computer' seem more human. In trials of the entertainment value of the game, the impression that the computer was as devious as the human player made the game an exciting challenge.

Nevertheless, the aim of the AI is to reach three consecutive wins. Therefore, the lower the number of wins, the more effective the AI is.

It is important to consider the number of steps in the winning strategy. If this number is very high, it indicates that there is considerable redundancy, or that what we have is not a strategy, but a brute force solution.

A higher number of operations performed over the entire game cycle indicates a less efficient system.

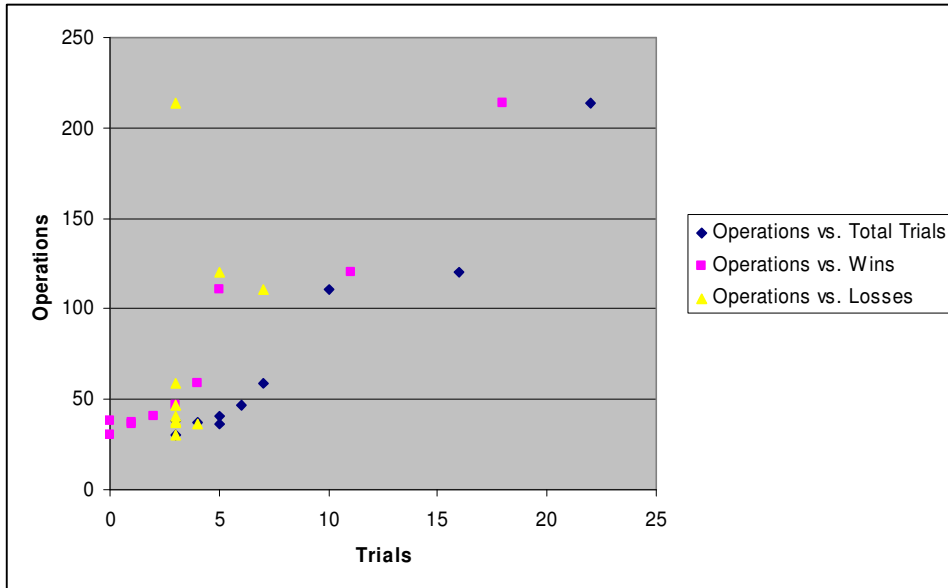
## Results of Experiment

195 attempts were performed on the Machine Room game of varying strategies, consisting resulting in 2294 operations.

### Five Second Total Plan (no location information, random ideas)

In this state the software is only guessing sequences of events that will lead to a win-state. Every five seconds the Knowledge dll is permitted to make a move in the game. No feedback is recorded except a win or lose state.

It is visually clear that the number of trials corresponds to the number of operations performed. Ideally we should see that the “Operations vs. Total Trials” should curve upwards, as the AI adapts towards the behavior of the user. This happens only in a couple of times. The furthest diamond point to the right represents an attempt where the system failed to come up with a strategy until 18 attempts had passed - an outcome that suggests randomness and not learning (and unsuitable in a much more complex system).



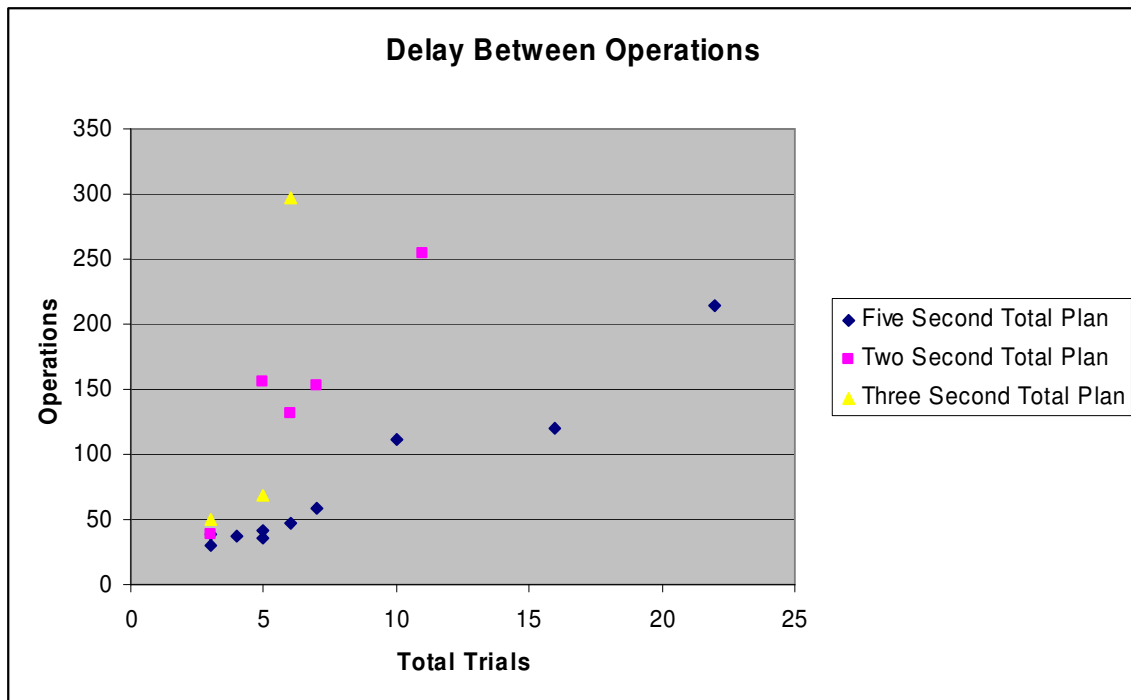
The data from the trials shows that the winning strategies were relatively efficient – with the average winning strategy under 9 operations, which indicates that the strategies are true strategies and not brute force methods. There are 10 possible operations, and assuming several of these are repeated (and therefore ignored) and considering that the events in a strategy must be synchronized to the time (as the player is moving across the room), therefore these strategies have meaning beyond randomness.

	Total Trials	Winning Strategy Steps	Operations	Wins	Losses	Notes
Attempt 1	6	8	47	3	3	trapped in pit
Attempt 2	5	8	36	1	4	closes first partition
Attempt 3	16	13	120	11	5	trapped on stairs
Attempt 4	3	10	38	0	3	trapped in pit
Attempt 5	4	7	37	1	3	closes first partition, opens two cages
Attempt 6	10	6	111	5	7	eventually trapped on stairs
Attempt 7	7	11	59	4	3	
Attempt 8	5	6	41	2	3	

Attempt 9	22	9	214	18	3	two partitions and the pit
Attempt 10	3	11	30	0	3	second partition
<b>Total</b>	<b>81</b>	<b>89</b>	<b>733</b>	<b>45</b>	<b>37</b>	
<b>Average</b>	<b>8.1</b>	<b>8.9</b>	<b>73.3</b>	<b>4.5</b>	<b>3.7</b>	

The Five Second Total plan technique works by allowing the Knowledge dll to submit an action every five seconds. To see how effective this is compared to brute force methods (where everything happens at once) the game was changed to request an action every two and every three seconds. The results are shown below.

It is clearly noticeable that the number of operations has increased considerably when the time delay between each operation is reduced. This is to be expected. But are the strategies any better?



It turns out that it takes longer for the player to be defeated using the two or three second delays, as the AI becomes it's own worst enemy. As it takes time for the partitions to open again once they are closed, the very frequent number of actions means that a lot of steps in the strategies are irrelevant. It also means that the stairs in the Machine Room is moved most of the time – and it becomes a useful place for the player to hide on, where the creatures cannot reach him/her. This leaves the AI throwing all the operations at the user and unable to have any effect.

Delay	Average Strategy Length
5 seconds	73.3
3 seconds	138
2 seconds	146

The situation is even worse if you look at the number of operations in the winning strategies:

Delay	Average Number of Steps in strategy that leads to 'mostly

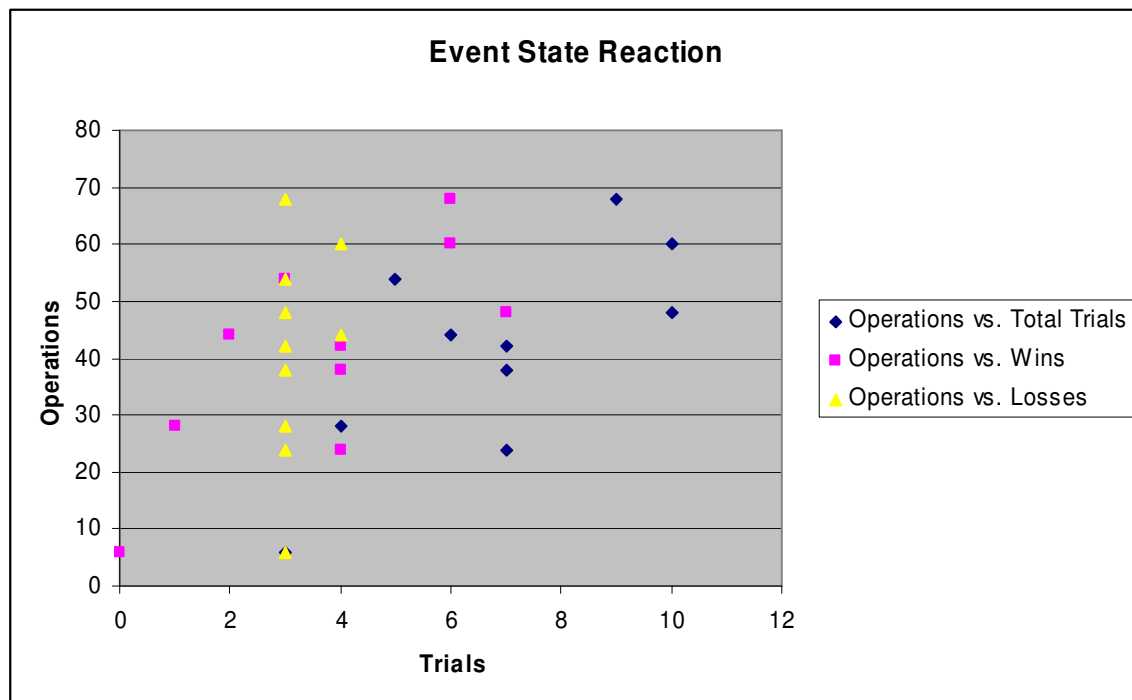
	unbeatable'
5 seconds	8.9
3 seconds	13
2 seconds	17.6

When the AI is allowed to run an operation every two seconds it expends a very wasteful amount of operations, which makes it very impractical for real world problems and not entertaining when part of a video game. This implies that one of the most important aspects of good AI is it's ability to choose wisely the timescale relative to it's task.

**Event State Reaction** allows the Knowledge dll to operate two machines every time the player enters a new region of the room. This means that there has to be some relation of where the user is located to exactly what devices are controlled.

The Event State reaction produces a very scattered graph for the number of operations vs. the total trials. This is because Event State reaction is more clever at coming up with strategies that work. This is probably due to the ability for this mode to run more operations than the ability to think for itself. This is shown by the lack of adaptation involved – there are very few strategies that were successfully beaten by the player (as opposed to several, which is the occurrence with the five second total plan).

The fact that there is little or no adaptation means that this method, in this particular implementation is less entertaining for the player. In addition as the environment is less chaotic, the true strength of the Knowledge dll: adaptation and taking successful strategies and extending them, can never be used. This means that the AI seems far less human, and consequently, less fun to play against. For truly intelligent computer players in video games, it is necessary for them to think in real-time (if the game is set in real time), rather than being event triggered.



There is one benefit to event state reaction. As we are very conservative in terms of the number of operations used the winning strategies are much, much shorter than those with the five second total plan. They unusually have little or no redundant operations. Compare 3.6, the average number of steps in the event triggered approach to 8.9, the number of steps in the five second total plan approach.

This, as well as the low number of operations (41.2 average vs. 73.3 average) indicates that despite making a less entertaining game, learning from triggered randomness yields a far more efficient response than from being allowed to move after a fixed period of time.

	Total Trials	Winning Strategy Steps	Operations	Wins	Losses	Notes
Attempt 1	10	4	48	7	3	first partition, first cage
Attempt 2	7	2	24	4	3	first partition, second cage
Attempt 3	3	2	6	0	3	first partition, second cage
Attempt 4	5	4	54	3	3	first partition, second cage
Attempt 5	6	4	44	2	4	second partition, first cage
Attempt 6	7	4	42	4	3	chases onto bridge
Attempt 7	10	4	60	6	4	partition 1 and 2, cage 2
Attempt 8	9	6	68	6	3	caught on stairs
Attempt 9	4	4	28	1	3	
Attempt 10	7	2	38	4	3	trapped stairs/first partition
<b>Average</b>	<b>6.8</b>	<b>3.6</b>	<b>41.2</b>	<b>3.7</b>	<b>3.2</b>	
<b>Total</b>	<b>68</b>	<b>36</b>	<b>412</b>	<b>37</b>	<b>32</b>	

The best solution would be to combine triggered responses, location information, with a system that is allowed to move when it wants to. Forgetting the triggering, if location information is combined with a frequency based response (five second total plan) the Machine Room game becomes much more interesting. If the player walks slowly, quickly, or runs the AI knows to learn a different strategy to deal with that particular situation. This has the huge advantage that if a human behaves differently, formally successful strategies are not forgotten. For example, if someone can beat one strategy by a slightly different way of moving around the room, the AI remembers the formally successful strategy and is ready to use if again if necessary. The ability to store multiple strategies makes this mode seem the most intelligent.

The only drawback to this approach is that it means the learning time is increased considerably, to deal with all kinds of branches of scenarios.

### Experimental Conclusions

Learning how to solve any problem, and in this case, learning to defeat the player in the Machine Room problem benefits from more information about the environment. In this example, the system was mostly blind as to what was really happening in the room – but this relates back to Plato’s cave conundrum – are we really interfacing with the world, or are we interfacing an interface?

Nonetheless it is true that as the Knowledge dll is blind when it deals with the room, it is unable to generalize its knowledge to solve other problems. In order to become more intelligent the AI must be able to recognize elements (i.e. partitions in the game) as being the same as other elements, and therefore somewhat being able to guess what is going to happen when they are used.

It is true that the unsupervised learning used in the Machine Room game is what makes it so entertaining – it is the AI’s incompetence at the beginning and watching it become more educated – if the game knew that the monster was a monster, it would know to release it, and therefore make the game less fun. If any learning systems are used for entertainment value in a game, whether they are to fight or to communicate with, it is important that they should learn how things work from a very basic level. The AI systems must be ignorant of the very basic facts of life (i.e. gravity) to be able to create an entertaining separation which seems like they are really a true life form.

## 6. Concluding Remarks

### Weakness of this AI

The unsupervised learning strategy implemented was able to adapt and solve the situation it was faced with: the Room Scenario Game. However through this experiment it became clear that there were some factors upon which that the AI performed weakly:

- ✘ It has difficulty with larger and larger amounts of data, as it becomes very difficult to map causes to effects.
  - To solve this problem a constant mapping process should occur in the background, identifying events with potential actions. Sensory data should be clustered together, rather than treated as discrete elements. It should also be possible to be quantitative with sensory data, to be able to compare how values compare to one another.
- ✘ The AI makes the same mistakes over and over again.
  - Making the same mistakes is not easily solved, as the software cannot disregard a failed solution permanently, as at a future date, the conditions may have changed and the incorrect solution becomes the right one. A possible solution would be to run 'tests' in the mind of the AI before performing anything, to check that the planned actions are still possible. Implementing a system where unsuccessful strategies time out is a simple, but flawed solution – it introduces more randomness, it doesn't solve the problem, and doesn't help the software learn.
- ✘ In this AI, there is no love! No obsession! No appreciation for aesthesia. Humans are fascinating because they appreciate ideas and actions based on very good things they associated them with in the past. An AI that would appear to be interesting should have these qualities.
  - Solution: the AI should favor:
    - Items encountered at a very young stage in the learning process
    - Items which it is attracted to (this should probably apply to complex things the machine gets along well with – for example, things that are not simple to learn about, but due to specific skills, or previous appreciations, they are easy to interface with). Attractive items would be ranked more highly in the decision making process.
    - Items that benefit the AI itself, over other things. I think the answer here is to teach the concept of material possession (to allow the AI to recognize assets as its own, where an owned asset would be an asset at the will of the AI), although Marxists would disagree. Possession is a jealous love, and in humans and animals, I feel, it derives from the need to find an exclusive sexual partner for reproduction. Taking this into account – is it possible to develop AI without making it reproductive? Possibly not. Reproduction seems a daunting problem if it is thought about in terms of the AI rewriting it's own code, however animals beings do not rewrite their own genetic code in reproduction, it is an automated process. Mating is not a process of sharing information (although it could be), and the resulting child should be nurtured by the AI parents, rather than absorbing information itself. To relate this back to the AI choosing things that benefit itself, the solution is to place a very basic demand that anything to do with the aim of creating AI children (who live long enough to reproduce) is slightly more important than everything else. As a consequence, all materialist intelligence should fall into place in the system. This basic desire would probably have to be part of the mapping cycle (mentioned above in the



## Success of the Linear Memory

It is clear that the linear memory of the software could be represented in a tree structure, to speed up computation and to more logically calculate similar histories. This leads to an application that is primarily concerned with making decisions at points on a path, this is not necessarily the answer as it could be too structured. Perhaps we need a rough tree, a data structure with branches that don't always connect to each other and some that are not even part of the tree at all?

## Entertainment Value and Practical Use in Computer Games

The Knowledge dll provided entertainment to players of the Room Scenario game, but was this entertainment a result of the AI's stupidity? For example, a human controlling the room scenario would probably be able to more quickly, come up with a strategy that would defeat the player of the game, simply because they are aware of how the room is arranged.

If the AI is made more intelligent does it lose its entertainment value? I believe the answer is negative due to two reasons:

1. The implementation of human values such as obsession and caring will make experience playing against this AI more interesting
2. As the AI becomes more clever, the games it is given should get more complex (especially games that do not rely on logic)

Good AI processing takes enormous computing power (the larger the memory, the greater the time to search – if processing video, this AI would take a very long time to accomplish anything).

The question: “given a computing device of infinite capability, is it easier to create AI?” leads to the statement: “calculators can't be intelligent, therefore there must be a lower limit for the capacity of intelligence – perhaps the more simple a device, the more logical it has to be?” Does this have anything to do with the separation between human beings and animals? Are there more points of separation?

It will be interesting to see how these questions are answered in the next 100 years.

To be more practical, the AI must be able to continually think on its own in its own thread on the computer operating system that it is running on. If the Knowledge dll implemented as part of this experiment worked in a computer game (the Room Scenario), it seems that the AI studied here is suited for computer games, although perhaps it's not a very efficient solution.

## Final Remarks

It is true that the knowledge dll could probably learn to play tic-tac-toe, or even chess. However, it would be a terrible player because it cannot think properly for these situations, except basic logic. This is AI designed for real life, AI that if extended, and built into a computer game, could operate characters who you can really feel to be your friends. It is also aimed in the direction of household robotics. A household robot doesn't need to know how to perform mathematics, but it does need to know how to organize a house – a problem that is not terribly logical.

## Appendix

Two Second Total Plan Experiment Data Table

	Total Trials	Winning Strategy Steps	Operations	Wins	Losses	Notes
Attempt 1	5	27	156	2	3	everything in pit
Attempt 2	7	9	153	3	4	
Attempt 3	3	17	39	0	3	
Attempt 4	11	20	255	5	6	second partition, then trap in pit
Attempt 5	6	15	131	3	3	interesting, trapped in left
<b>Average</b>	<b>6.4</b>	<b>17.6</b>	<b>146.8</b>	<b>2.6</b>	<b>3.8</b>	
<b>Total</b>	<b>32</b>	<b>88</b>	<b>734</b>	<b>13</b>	<b>19</b>	

Three Second Total Plan Experiment Data Table

	Total Trials	Winning Strategy Steps	Operations	Wins	Losses	Notes
Attempt 1	3	15	50	0	3	Rapid part. 1&2
Attempt 2	5	11	68	2	3	same as above
Attempt 3	6	13	297	3	3	same as above
<b>Average</b>	<b>4.666666667</b>	<b>13</b>	<b>138.3333333</b>	<b>1.666667</b>	<b>3</b>	
<b>Total</b>	<b>14</b>	<b>39</b>	<b>415</b>	<b>5</b>	<b>9</b>	

# Bibliography

## References for Philosophy/Psychology

*Sources for ideas, these sources were not cited directly*

Stan Franklin, Artificial Minds, Boston MA USA, The MIT Press, March 1997.

Ray Kurtzweil, The Age of Spiritual Machines: How We Will Live, Work, and Think in the New Age of Intelligent Machines., USA, Texere Publishing, 1999.

George B. Dyson, Darwin Among the Machines: The Evolution of Global Intelligence, New York NY USA, Basic Books (Perseus Books Group), 1997.

Steven Johnson, Emergence: The Connected Lives of Ants, Brains, Cities and Software, New York NY USA, Touchstone Books (Simon & Schuster), 2001.

Wikipedia, "Psychosis", <http://en.wikipedia.org/wiki/Psychosis> (accessed December 30, 2005)

## References for Machine Learning

*Sources for ideas and terminology*

Nils J. Nilsson, "Introduction to Machine Learning", 1997, Robotics Laboratory, Stanford University.

Shawn Arseneau and Rene van Wijhe, "Unsupervised Learning and Clustering" (Summary of "Graph-Theoretical Methods for Detecting and Describing Gestalt Clusters" written by C.T. Zhan, and published in IEEE Transactions on Computers in 1971), <http://cgm.cs.mcgill.ca/~soss/cs644/projects/wijhe/>

Jennifer G. Dy (*Department of Electrical and Computer Engineering, Northeastern University*), Carla E. Brodley (*School of Electrical and Computer Engineering, Purdue University*), "Feature Selection for Unsupervised Learning"

John Goldsmith, University of Chicago, "Unsupervised Learning of the Morphology of a Natural Language"

## References for C++ Programming:

*Algorithm implementations and general coding advice*

Department of Computer Science, University of Regina, <http://www.cs.uregina.ca/>

California State University, San Bernardino, Computer Science Department, <http://www.csci.csusb.edu/>

Mark Sebern., Milwaukee School of Engineering, <http://people.msoe.edu/~sebern/>

Dept. of Computer Science, Bowling Green State Univ., <http://www.bgsu.edu/departments/compsci/>

Alavor Vasudevan, "C++ Programming - How to", 2002, LinuxSelfHelp.com, <http://www.linuxselfhelp.com/HOWTO/C++Programming-HOWTO.html>

Oracle Thinkquest Educational Foundation, "Quicksort implementation in C++", <http://library.thinkquest.org/>

Tim Love, Cambridge University Engineering Dept, <http://www.eng.cam.ac.uk/~tpl/>  
Lucy Garnett (Heriot Watt University, School of Mathematical and Computer Sciences)  
Building Business Applications Using C++, 1996, Addison Wesley Longman Inc., Reading MA, USA.