A Modified Experts Algorithm: Using Correlation to Speed Convergence With Very Large Sets of Experts

by

Jeremy Schwartz

Submitted to the Department of Mechanical Engineering in partial fulfillment of the requirements for the degree of Master of Science in Mechanical Engineering at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2006

© Jeremy Schwartz, MMVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author ......................... .................................. Department of Mechanical Engineering May 12

Certified by .................................................. Daniela Pucci de Farias Assistant Professor Thesis Supervisor

Accepted by .................................................. Lallit Anand Chairman, Committee on Graduate Students
A Modified Experts Algorithm: Using Correlation to Speed Convergence With Very Large Sets of Experts

by

Jeremy Schwartz

Submitted to the Department of Mechanical Engineering on May 12, in partial fulfillment of the requirements for the degree of Master of Science in Mechanical Engineering

Abstract

This paper discusses a modification to the Exploration-Exploitation Experts algorithm (EEE). The EEE is a generalization of the standard experts algorithm which is designed for use in reactive environments. In these problems, the algorithm is only able to learn about the expert that it follows at any given stage. As a result, the convergence rate of the algorithm is heavily dependent on the number of experts which it must consider.

We adapt this algorithm for use with a very large set of experts. We do this by capitalizing on the fact that when a set of experts is large, many experts in the set tend to display similarities in behavior. We quantify this similarity with a concept called correlation, and use this correlation information to improve the convergence rate of the algorithm with respect to the number of experts. Experimental results show that given the proper conditions, the convergence rate of the modified algorithm can be independent of the size of the expert space.

Thesis Supervisor: Daniela Pucci de Farias
Title: Assistant Professor
## Contents

1 Introduction ............................................. 11
   1.1 Experts Algorithms ................................. 12
       1.1.1 The Repeated Prisoner’s Dilemma ............ 12

2 Discussion ............................................... 15
   2.1 Definition of Variables .............................. 15
   2.2 The Exploration-Exploitation Experts Algorithm .... 16
       2.2.1 The Repeated Prisoner’s Dilemma, Revisited .. 17
   2.3 Convergence rate of EEE With Large Sets of Experts .. 18

3 Correlation .............................................. 21
   3.0.1 The Correlated Experts Algorithm ............... 21
   3.0.2 Discounting Correlation .......................... 22
   3.0.3 Determining the Correlation Factors .............. 23
   3.0.4 A Note on Exploitation Phases ................... 24

4 Experiment ............................................... 27
   4.1 Setup ................................................. 27
       4.1.1 Generating Experts .............................. 28
   4.2 Simulation 1: Directing a Bot to a Point on the Field 29
   4.3 Simulation 2: Managing Multiple Bots and Multiple Tasks on the Field 30
       4.3.1 The Agent and the Experts ...................... 31
       4.3.2 Expert Generation .............................. 32
4.3.3 The Importance of Good Correlation Values  
4.4 Running Simulation 2
  4.4.1 Opponents
  4.4.2 Expert Sets
  4.4.3 The Algorithms vs. the Best Expert
  4.4.4 Additional Variables at Play
4.5 Analysis
  4.5.1 Group A opponents
  4.5.2 Group B opponent
  4.5.3 Group C opponents
  4.5.4 Group D opponent
5 Conclusion
A Data
B Simulation Code
  B.1 Agent.java
  B.2 Bot.java
  B.3 DisplayPanel.java
  B.4 DisplaySwing.java
  B.5 DistributionExpert.java
  B.6 Expert.java
  B.7 ExpertList.java
  B.8 HistogramExpert.java
  B.9 Locatable.java
  B.10 Location.java
  B.11 RecencyExpert.java
  B.12 Target.java
  B.13 TargetManager.java
List of Figures

4-1 Sample behavior with different correlation factors .................. 34
A-1 Distributions of expert performance ................................. 50
A-2 Average performances against opponent 1 .......................... 51
A-3 Average performances against opponent 2 .......................... 52
A-4 Average performances against opponent 3 .......................... 53
A-5 Average performances against opponent 4 ......................... 54
A-6 Average performances against opponent 5 .......................... 55
A-7 Average performances against opponent 6 .......................... 56
A-8 Performance of best expert against opponent 1 .................... 57
A-9 Performance of best expert against opponent 2 .................... 58
A-10 Performance of best expert against opponent 3 ................... 59
A-11 Performance of best expert against opponent 4 .................. 60
A-12 Performance of best expert against opponent 5 .................. 61
A-13 Performance of best expert against opponent 6 .................. 62
A-14 Performance of CEA against opponent 1 .......................... 63
A-15 Performance of CEA against opponent 2 .......................... 64
A-16 Performance of CEA against opponent 3 .......................... 65
A-17 Performance of CEA against opponent 4 .......................... 66
A-18 Performance of CEA against opponent 5 .......................... 67
A-19 Performance of CEA against opponent 6 .......................... 68
A-20 Performance of EEE against opponent 1 .......................... 69
A-21 Performance of EEE against opponent 2 .......................... 70
A-22 Performance of EEE against opponent 3 . . . . . . . . . . . . . . . . 71
A-23 Performance of EEE against opponent 4 . . . . . . . . . . . . . . . . 72
A-24 Performance of EEE against opponent 5 . . . . . . . . . . . . . . . . 73
A-25 Performance of EEE against opponent 6 . . . . . . . . . . . . . . . . 74
List of Tables

1.1 Row player rewards for Repeated Prisoner’s Dilemma ............ 13

4.1 Number of experts generated with each distribution length ........ 37
Chapter 1

Introduction

Many problems that we regularly encounter require that we make a sequence of decisions. Most of these problems can be put into the framework of the repeated game: at each stage in the game, the agent chooses an action, and then receives a reward. This reward can depend on the action chosen, or the state of the environment, or both. Additionally, the action chosen may or may not directly affect the state of the environment. This last issue depends on whether the environment is oblivious or reactive.

The type of environment is a major distinction between different types of repeated games. In an oblivious environment, the agent’s actions do not have any effect on the state of the environment. An example of this situation is an individual investor in the stock market. Because of the size of the market, none of the actions taken by the investor have a noticeable effect on the market.

A reactive environment is more akin to a board game, such as chess. In this case, it is clear that every action we take fundamentally alters the state of the game board, and affects the opponent’s strategy. In such an environment, it is more difficult to compare the value of two actions, because each action leads to a different state. In investing, it is natural to look back and recognize how you could have improved your performance. In chess, on the other hand, there is no easy way to look back at a game and declare with certainty that an alternate strategy would have been better.
1.1 Experts Algorithms

An Experts Algorithm is a particular methodology for playing a repeated game, given a set of strategies. In the case of Experts Algorithms, the agent does not have direct knowledge of the environment. Instead, the agent has access to a set of experts, or strategies, which do have knowledge of the environment. Each of the experts advises an action at each stage, and the agent must decide which expert to follow.

The algorithm tests out various experts by following their advice for a number of stages, comes up with an estimation of the performance of each expert, and converges over time upon the expert that performs the best on average. In many cases, we can prove that these algorithms will, in the limit, perform at least as well as the best single expert. Additionally, in many cases, we can prove bounds on the rate at which the algorithm will converge to this optimal performance.

Notice that the type of environment — oblivious or reactive — has a huge effect on the performance of the algorithm. In an oblivious environment, the algorithm is able to gather information about all of its experts at each stage — even experts that it did not follow. Thus, the algorithm is very quickly able to discover the value of each expert.

In reactive environments, at each stage the algorithm can only learn about the particular expert that it chose to play on that stage. Additionally, there is rarely (if ever) an opportunity to consider the performance of two different strategies in the exact same state. After all, each time the agent takes some action, it alters the state of the game itself. Thus it makes sense that the convergence rate of an experts algorithm in a reactive environment is worse than in the case of an oblivious environment.

1.1.1 The Repeated Prisoner’s Dilemma

For a simple depiction of the difference between an oblivious environment and a reactive one, consider the repeated prisoner’s dilemma. At each stage, we are given the opportunity to cooperate with our opponent (C) or defect from him (D). Our opponent has the same choice. The object in the game is to minimize our own cost,
Table 1.1: Row player rewards for Repeated Prisoner’s Dilemma

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>-1</td>
<td>-5</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>-3</td>
</tr>
</tbody>
</table>

which in this case can be likened to time in jail.

If both players cooperate, we both receive a small cost. If both players defect, we both receive a high cost. However, if one player defects and the other player cooperates, the defecting player receives the lowest cost while the cooperating player receives the highest cost. The payoffs are described in Table 1.1. Define the agent as the row player, the opponent as the column player, and the values in the table as the agent’s payoff given the combination of actions by the agent and the opponent.

If we know that our opponent chooses his actions oblivious to our agent, then the optimal strategy is very simple: defect every time. Simple minimax theory shows that this is the best strategy; whether our opponent chooses C or D, we are better off choosing D than C.

On the other hand, if we know that the opponent’s next action depends on our own past choices, the game is not so simple. For example, consider the case where the opponent is playing the tit-for-tat strategy. With this strategy, at the first stage the opponent plays C. After that, on each stage the opponent plays whatever action we played on the previous stage.

If our agent were to choose the strategy of constant defection in this situation, we would end up with an average payoff of -3 per stage. On the other hand, if our agent were to choose constant cooperation, we would end up with an average payoff of -1. Thus, the strategy that was clearly optimal in the oblivious environment is not necessarily optimal in a reactive environment.

In general, the difference in convergence rates for oblivious and reactive environments is a factor of \( r \), the number of experts that the algorithm must consider. This is the problem which we hope to ameliorate in this paper. We provide a framework for using the relationship between different experts to speed the learning process about
all the experts in the set, and therefore increase the convergence rate of the algorithm.
Chapter 2

Discussion

2.1 Definition of Variables

These variables are used throughout this paper. For the sake of clarity, we will define all of our variables here.

- $r$ denotes the total number of experts which are considered by the algorithm.
- $M_i$ denotes the average score of the algorithm after $i$ phases. Without confusion, let $M_s$ denote the average score of the algorithm after $s$ stages.
- $M_e$ denotes the estimation of expert $e$ after it has been played for $i$ phases, \textit{without correlation}. Note that this estimation is a \textit{weighted} running average.
- $\hat{M}_e$ denotes the estimation of expert $e$ after it has been played for $i$ phases, \textit{with correlation}. Note that this estimation is a \textit{weighted} running average.
- $S_e$ denotes the number of stages in which expert $e$ has been played, \textit{without correlation}.
- $\hat{S}_e$ denotes the number of stages in which expert $e$ has been played or virtually played with correlation.
- $p_i$ is the probability with which we will play an exploration phase at stage $i$. In order to ensure convergence, we must only have: $\lim_{s \to \infty} p_i = 0 \ [2]$. 

15
• $e_i$ is the expert chosen at phase $i$.

• $n_i$ is the number of stages played in phase $i$.

• $C_f(i)$ denotes the correlation factor between expert $f$ and the expert which was played during phase $i$.

• $\beta$ denotes a discount factor for the correlation values. We decrease the effect of correlation as time goes on by multiplying it by $\beta^i$.

• $\gamma$ denotes a past/present weighting factor for $M_e$ and $\tilde{M}_e$. If $\gamma = 1$, then the estimation of each expert will be an exact running average. If $\gamma > 1$, then the past will be weighted more than the present. If $\gamma < 1$, then the present is weighted more heavily than the past.

• $R_s$ denotes the payoff received at stage $s$.

• $\bar{R}(i)$ denotes the average payoff received during phase $i$.

2.2 The Exploration-Exploitation Experts Algorithm

In some situations involving Experts Algorithms, the agent is given the entire cost vector at every stage. This is equivalent to saying that the agent is made aware of the cost of playing any of its experts at the previous stage. In other words, this corresponds to the oblivious environment described above.

In other situations, the agent is only given the cost of the particular action that it played on any given round. These situations correspond to reactive opponents, where one's actions directly affect the actions of the opponent or the state of the environment.

The Exploration-Exploitation Experts algorithm (EEE), is a generalization of the standard Experts algorithm that allows for a reactive opponent [2]. As a result, the agent only receives information about the expert that it follows on a given turn; it cannot see the expected payoff for taking any other actions in that turn. The algorithm works as follows:
1. Initialize $M_x = N_x = S_x = \hat{S}_x = 0$ ($x = 1, \ldots, r$) and $i = 1$.

2. With probability $p_i$, perform an exploration phase, and with probability $1 - p_i$ perform an exploitation phase; denote by $e_i$ the expert chosen to be followed and by $n_i$ the number of stages chosen for the current phase.

3. Follow expert $e_i$'s instructions for the next $n_i$ stages. Increment $N_{e_i} = N_{e_i} + 1$ and update $S_{e_i} = S_{e_i} + n_i$. Define $\gamma$ to be a past/present weighting factor. Denote by $\tilde{R}$ the average reward accumulated during the current phase of $n_i$ stages and update

$$M_{e_i} = (1 - \frac{n_i}{S_{e_i}})\gamma M_{e_i} + (1 - \gamma + \gamma \frac{n_i}{S_{e_i}})\tilde{R}.$$ 

4. Increment $i = i + 1$ and go to step 2.

Note that we are free to choose $n_i$, the number of stages played in each particular phase. For the length of this paper, we will consider the case where the number of stages per phase increases linearly with the number of phases that we have played a particular expert: $n_i = i\varepsilon$.

We generally want $n_i$ to increase as time goes on. This is another result of playing a game in a reactive environment. It is often reasonable to assume that the environment has a finite memory, and only reacts to the more recent actions that we have taken. Still, this memory may be large, or unknown, or both. As such, if we only play a particular expert for short lengths of time, our estimation of it may be corrupted by environmental reactions to previous experts. This is the motivation for playing an expert for continually increasing lengths of time—the longer we play a single expert, the more accurate our estimation of that expert will be.

### 2.2.1 The Repeated Prisoner’s Dilemma, Revisited

For a simple example describing this effect, consider again the problem introduced in the last chapter. Say we have two experts to choose from: expert $a$ always recommends cooperation, and expert $b$ always recommends defection. Say the opponent is
playing the tit-for-tat strategy, where it at each stage it plays the action that we our
agent played on the previous stage.

At the start of the game, we randomly select expert \( b \), which we follow for one
stage. Since the opponent plays \( C \) on the first round with the tit-for-tat strategy,
our reward is 0, according to the payoff table. Then say we select expert \( a \) for the
following round. The opponent plays \( D \) in the second round, since we played \( C \) in the
first round. Thus, our payoff is \(-5\). Thinking that expert \( a \) is bad, we would go back
to expert \( b \) during exploitation rounds. Indeed, every stage that we played \( a \) after
playing expert \( b \), our payoff would be \(-5\). If we did not increase the number of stages
we dedicate to an expert each time we played it, then the average performance of
expert \( a \) would wallow around \(-5\). On the other hand, if we increased the number of
stages we dedicate to following expert \( a \) each time we chose it, then it would converge
it its true value – in this case, \(-1\).

In fact, without increasing phase lengths, we cannot guarantee convergence to a
ture estimation of each expert. The EEE solves this problem, among others. [2]

One issue with the EEE is that it must play a particular expert for a reasonable
length of time in order to have any estimation of the performance of that expert.
Thus, the convergence rate of the algorithm is heavily dependent on the number of
experts that the algorithm must consider. When the set of experts is small, this
dependence is manageable. However, in situations where the number of experts may
be combinatorially or exponentially related to the size of the problem, then the time
to convergence rapidly becomes very large.

2.3 Convergence rate of EEE With Large Sets of
Experts

Consider the performance bounds shown for the EEE in [2]. In the case of linearly
increasing phases and polynomially decreasing exploration, the dependence on \( r \), the
number of experts, is at least \( O(r^{1.866}) \). In many problems, such as those detailed in
the Experiment section of this paper, the number of experts can grow combinatorially or exponentially with the size of the state space. Thus, the convergence rate of the algorithm can, in some cases, have an exponential dependence on the size of the problem. This is clearly not desirable. Our task is to modify the EEE to consider situations where the set of experts is very large. The technique by which we achieve this is called correlation.
Chapter 3

Correlation

The key insight is that with most problems involving large sets of experts, many of these experts are only slight variations of each other. If we could find a way to quantify the similarity between experts, then we could generalize the information that we learn at each step. Rather than just gather estimation information about the expert that we play, we can update our estimates of each expert that is related to our chosen expert. The correlation framework provides the means to accomplish this generalization.

Each pair of experts is given a correlation value between 0 and 1, with 0 meaning the two experts are entirely dissimilar, and 1 meaning the two experts are functionally identical. Then, each time we play an expert, we update every expert in the set to a degree dictated by the correlation value between it and the expert we played.

3.0.1 The Correlated Experts Algorithm

The CEA runs as follows:

1. Initialize $M_e = \hat{M}_e = N_e = S_e = \hat{S}_e = 0$ ($e = 1, \ldots, r$) and $i = 1$.

2. With probability $p_i$, perform an exploration phase, and with probability $1 - p_i$ perform an exploitation phase; denote by $e_i$ the expert chosen to be followed and by $n_i$ the number of stages chosen for the current phase. (Note that during exploitation phases, experts are ranked based on $\hat{M}_e$, and not $M_e$.)
3. Follow expert $e_i$’s instructions for the next $n_i$ stages. Increment $N_{e_i} = N_{e_i} + 1$ and update $S_{e_i} = S_{e_i} + n_i$. Define $\gamma$ to be a past/present weighting factor. Denote by $\bar{R}$ the average reward accumulated during the current phase of $n_i$ stages and update

$$M_{e_i} = (1 - \frac{n_i}{S_{e_i}})\gamma M_{e_i} + (1 - \gamma + \frac{n_i}{S_{e_i}})\bar{R}.$$ 

4. Additionally, for every expert $f$ in the set, denote $C_f(i)$ to be the correlation value between expert $f$ and the expert currently being followed. Let $\beta$ be a constant correlation discount factor. Then update:

$$\hat{S}_{e_i} = \hat{S}_{e_i} + C_f(i)n_i\beta^i.$$ 

$$\hat{M}_{e_i} = (1 - \frac{C_f(i)n_i\beta^i}{S_{e_i}})\gamma \hat{M}_{e_i} + (1 - \gamma + \frac{C_f(i)n_i\beta^i}{S_{e_i}})\bar{R}.$$ 

5. Increment $i = i + 1$ and go to step 2.

Note that at each stage, we update every expert that has a nonzero correlation factor with the expert that we played. Further, the means by which we update each expert is intuitive. We simply use $C_f(i)n_i\beta^i$ in place of $n_i$ for the chosen expert. The average payoff, $\bar{R}$, is unchanged. Thus, we act as if we played the correlated expert for fewer stages, with the received payoff. The received payoff will accordingly be weighted less in the long-run average estimate of the correlated expert than in the chosen expert, and it will be weighted more in more highly correlated experts than in less correlated experts.

### 3.0.2 Discounting Correlation

The technique of using the same received payoff to update all experts keeps the algorithm simple, but has a subtle-long term effect that must be handled by discounting
the correlation factors over time.

If the correlation factors stayed constant, then consider what would happen once we settled upon a single best expert. Say expert $f$ is the best expert in a particular environment. At some point, the algorithm will discover this, and begin to play expert $f$ with high probability. As we played expert $f$ for longer and longer, all correlated experts would have their score weighted more and more by the score of expert $f$.

In the limit, the score of all experts correlated to expert $f$ would be equal to expert $f$'s score. Thus, at some point we may stop playing expert $f$ in favor of a worse expert. Since we want to have the behavior of settling on the best expert (in the cases where a single expert is best), we must allow for the correlation information to disappear over time.

The disappearance of the correlation information also makes sense from an intuitive perspective. The correlation information is used to very quickly build a reasonable estimate of a large set of experts after playing only a few members of the set. As time goes on, and the algorithm has enough time to try every expert in its arsenal, the need for correlation information disappears.

If we allow for the discounting of the correlation factors over time, then we can prove that the CEA will perform better than the best expert in the limit.

3.0.3 Determining the Correlation Factors

It is evident that with the algorithm described above, everything depends on the quality of the correlation information. If all correlation factors are zero, then this algorithm behaves exactly as the original EEE algorithm. If all correlation factors are equal to 1, then this algorithm will modify all experts equally until the correlation factors disappear, and thus it will actually impede learning.

Ideally, we would like that algorithm to determine the correlation factors online, so that it needs no a priori information in order to achieve the boost in performance. Currently, a reliable method for achieving this has not been discovered. Indeed, there is no current method to determine what the ideal set of correlation factors might be for a given set of experts. All we are able to do at this point is venture a reasonable guess
as to a good correlation metric, and test out our guess with experimentation. This issue is one of the largest open problems with the CEA today. There are many possible solutions to this question, and with additional time and work, we are confident that online correlation will be achieved.

To see an example of the method by which we determine good correlation factors for specific problems, read the Experiment section in this paper.

3.0.4 A Note on Exploitation Phases

With most versions of Experts Algorithms, the exploitation phase simply involves choosing the single expert which has the best average score so far. With the CEA, this technique is not ideal.

Consider the situation where the algorithm randomly selects an expert that happens to perform quite well. Say, for the sake of clarity, that we choose expert $e$. Expert $e$ is good, but not the best expert for this particular game. However, expert $e$ is highly correlated with expert $f$, which does happen to be the best expert for this game.

Over the course of the phase in which we play expert $e$, its score shoots up higher than any of the other experts. Because the algorithm essentially scores correlated experts with less weight, expert $f$ will also have its average score increase, but its score will remain lower than the score of expert $e$.

The next time we are prompted for an exploitation phase, if we simply choose the single best expert, we will choose the expert that we just played — that is, expert $e$. Even though the score for expert $f$ is nearly as high as expert $e$, it will never actually be chosen until it is randomly selected in an exploration phase. This behavior is exactly the same as the original EEE algorithm; we are left to “discover” expert $f$ at random. The whole purpose of the correlation framework is to increase the probability with which we play expert $f$, if expert $f$ is good.

The clear solution is to modify our behavior during the exploitation phase. Rather than choosing the single best expert, we collect the $N$ best experts, and choose among them stochastically, with the probability for choosing each expert weighted by its
score. The ideal "best group" size $N$ is determined experimentally, but it does not have to be very large. It just needs to be large enough to allow for the algorithm to travel among the best experts, and determine which one is the single best performer.

It should be noted that this modified behavior during exploitation phases is not necessary in order to show convergence to optimal behavior. This modification does, however, greatly increase the performance of the algorithm based on experimental results.
Chapter 4

Experiment

4.1 Setup

Both of the following experiments were set up with a specific application in mind. The driving goal of this thesis was to apply the advanced concepts of Experts Algorithms to a situation akin to distributed robotics problems.

As such, both experiments involve directing one or many individuals (hereby referred to as “bots”) on a flat playing field. All of the bots have a single, very simple capability; on each turn, the bot is given a direction in which to travel, and it travels a unit distance in that direction.

A bot never has the concept of reaching a destination, and so it never stops moving. Also, the playing field itself is continuous. For both of these reasons, since both of the experiments involve getting a bot to an exact location, the bots will “vibrate” over a desired location by moving rapidly left and right over the desired area. Since the bot will always remain within a unit distance of the desired location when it vibrates in this fashion, we must simply build a small tolerance into the system: if a bot is within a unit of a target, we consider the bot to be “touching” the target.
4.1.1 Generating Experts

With standard Experts Algorithms, the handful of experts that the algorithm must consider are often hand-picked. They are created from common optimal algorithms for different classes of problems.

In some problems, though, we can only narrow down our experts to an optimal class of experts, all of which are a parameterized version of a general idea. For example, consider the following simple game:

We have a system with five queues. For each queue $x$, at each stage a unit will be added to that queue with some unknown probability $q_x$. Also, for each queue we are free to select $a_x$, the rate at which units are removed from the queue. The object of the game is to keep exactly $n$ units in every queue.

If $n$ is known, then a simple expert could be proposed for this game. It would try some set of actions $A = a_1, a_2, a_3, a_4, a_5$. If a queue gets too short, the expert slows the corresponding removal rate for that queue, and if a queue gets too long, then the expert speeds the removal rate for that queue.

Now consider what happens if $n$ is unknown. Instead of knowing $n$, we simply receive a score each round, based on how far away from $n$ each queue stands. In this situation, instead of increasing the complexity of our experts, we could simply generate a separate expert for each possible value of $n$. Taking this game one step further, if each queue had its own target number $n_x$, then we can see how the number of possible experts suddenly becomes exponential in the number of queues. At the same time, many of our experts will be very similar to each other, because they may only have slightly different estimations of $n_x$.

This type of problem is the motivation for the Correlated Experts Algorithm. Though we may have a small number of different types of experts, variation of parameters causes the number of individual experts to grow aggressively with the size of the problem. In both simulations described below, the number of experts becomes exponentially or combinatorially dependent on a single variable.
4.2 Simulation 1: Directing a Bot to a Point on the Field

The first simulation that we constructed was a very simple one, designed only to test the viability of the correlation concept. The problem presented here is readily solvable, and easier to solve, using algorithms such as gradient descent. Nonetheless, it served to prove that correlation information improved the performance of the standard EEE algorithm in this problem. We will summarize this problem here for completeness, without offering experimental data on the problem. Our analysis of experimental data is focused on the second simulation, which is a much more interesting application of this algorithm.

A single bot is placed on a field. Some point on the field is selected by the environment as the goal point. At each stage, the agent must decide on a direction in which to move the bot. The bot then moves a unit distance in that direction, and the environment gives the bot a score based on the bot’s current distance away from the target point.

In order to generate experts for this problem, we divided the field up into a grid of points. We assigned an individual expert to each point on the grid. Each expert treats its assigned point as the goal point. Thus, when the agent asks the experts for advice, each expert advises a direction that would lead the bot to that expert’s assigned point.

Given this setup, a few things are evident. First, the number of experts will grow with the size of the field squared. Second, if we constrain the true goal points to our defined grid, then we can guarantee that one of the experts will be the optimal expert. Finally, the distance between assigned points is an easy choice for our correlation metric between different experts.

Additionally, this setup serves as clear, intuitive testament to the power of correlation. Using the uncorrelated algorithm, the agent would make no connection between a high scoring expert and experts that are assigned nearby points. In this case, if the algorithm finds an expert whose assigned point is close to the target point, then this
algorithm will fixate on that expert until it happens to randomly select an expert which is assigned a closer point.

On the other hand, using the correlated algorithm, when the agent finds a high-scoring expert, it raises the value of all experts that are assigned points near that expert. Using the modified exploitation phase discussed earlier, the chance that one of the nearby experts will be selected on an exploitation phase is greatly increased.

Thus, the uncorrelated algorithm will often find an expert which is close to optimal, but the correlated algorithm will continue to step closer and closer as it tries correlated experts, until it finally hits the optimal point and the optimal expert. At this point, the correlated algorithm does have a chance of stepping off of the optimal expert during exploitation phases. However, since the correlation information is discounted over time, once the optimal expert is found, it is soon played exclusively during exploitation phases.

4.3 Simulation 2: Managing Multiple Bots and Multiple Tasks on the Field

The second experiment discussed in this paper is a task of resource management. While the first problem proved the superiority of the correlated algorithm over the uncorrelated algorithm for certain types of problems, this problem is much more interesting in that an easier solution is not currently known. Resource management is one of the most heavily studied problems for all kinds of different algorithms.

This problem is again set up with distributed robotics in mind. A number of bots are scattered on a field, and new tasks are constantly popping up at various locations. Each task has a deadline, after which it disappears. The objective of the game is to collectively accomplish as many of the tasks as possible.

Each task, when created, is placed at a location. In order for the task to be accomplished, one of the bots must remain at the location of the task for a few turns. If no bot accomplishes the task before its deadline runs out, then the task disappears
unaccomplished.

The agent controls the movement of all of these bots by assigning a direction to each bot on each turn. Each bot then moves a unit distance in its assigned direction each turn, as detailed before.

Scoring in this game is very simple; the agent receives a fixed score each time it accomplishes a task. For ease of reading, the score received for each task is the inverse of the average number of tasks generated per turn. Thus, if all tasks are successfully completed, the average score for the agent will be 1.

The opponent, in this case, is the algorithm that decides when and where to place the tasks on the field. This opponent can be deterministic or stochastic; for example, one opponent may place tasks according to a probabilistic distribution that is weighted heavily towards the four corners, while another opponent may place all tasks cyclically at ten particular points on the field.

Thus we arrive at the meat of the problem. The agent must decide where it wants its bots to loiter while waiting for new tasks, and whether or not to go after a task once it appears. This is exactly the problem presented to the experts.

### 4.3.1 The Agent and the Experts

Each expert receives the list of tasks currently on the field, and generates a list of targets that the agent should go after. *This list may include some, all or none of the tasks on the field.* Also note that this target list includes enough tasks to occupy every bot on the field, so the agent can provide an assignment to every bot.

The agent itself is then responsible for directing its set of bots to the list of targets that it receives from the expert. Note that the agent needs no knowledge of the actual tasks on the field in order to complete its job — it defers all of that knowledge to its experts.

The method by which the agent matches bots with targets is not a trivial one. This, in itself, is a problem that can be optimized. In our simulation, this algorithm is currently the greedy algorithm, and is far from optimal. Improving this algorithm would probably improve the efficiency of the agent. However, given that it is the same
for all experts, it should not greatly affect our central question of whether correlation helps our convergence rate.

4.3.2 Expert Generation

Given the setup of this problem, our possible set of experts grows exponentially with the size of the field and the number of bots on the field. Even considering a clever heuristic strategy, we can produce a very large set of experts by varying the parameters of that strategy.

For example, one of the currently implemented classes of experts is the histogram class. All of the experts in this class work off of the same heuristic. They divide the field into buckets, and take note of the frequency with which tasks are generated in each bucket. They then recommend a set of targets that move the bots towards the most concentrated buckets. However, within this set, we can vary the distribution that the expert recommends. One expert might recommend that all bots head towards the most populated bucket, while another expert may spread the bots evenly throughout the five most populated targets. Producing every viable combination of distributions results in a combinatorial explosion of experts.

In the case of the histogram class, a particular expert starts with a list of targets that simply represent the center of the buckets that it has chosen, according to its distribution. It then checks to see if any tasks sit within these buckets. If a task does sit in a chosen bucket, the expert will replace one of its false targets from that bucket with that task. Thus, the list generated by this expert will include all tasks which fall in the expert’s chosen range, and enough false targets to fill in the list.

Another class of experts that is currently implemented is the recency class. This class works off of the same rules of distribution as the histogram class. Common to every expert in this class is a list of the $x$ most recently generated target locations, where $x$ is the number of bots on the field. Each member of the class has a unique distribution of bots among those targets, just like with the histogram class.

Note that since the experts are required to produce exactly as many targets as there are bots, the distribution for any expert must add up to the number of bots on
the field. Also, for both the histogram and recency classes of experts, we limit the possible distributions to those that are monotonically decreasing. We do not allow all distributions — even though we are looking for an explosion in the number of experts, this implementation proved to explode too fast, and it became infeasible to run the simulation! This also makes sense intuitively, because a reasonable expert wouldn't assign more bots to a less important tasks than it would to a more important task.

4.3.3 The Importance of Good Correlation Values

As stated before, one large issue with the implementation of this algorithm is determining the correct correlation factors. Indeed, even certain very small changes in the set of correlation factors can have a large impact on the performance of the algorithm. We will demonstrate this effect while we explain how the correlation factors were chosen in this experiment.

Given the setup of this problem, it is clear that the distributions are a good metric for determining similarity between experts. Initially, we determined the correlation factors between two experts as follows: if two experts were of a different class, then they were deemed uncorrelated. If the experts were of the same class, then we summed up the element-by-element difference between the distributions. We then divided this by the maximum possible difference; in the case of monotonically decreasing distributions, this is equal to $L^2 - 2$, where $L$ is the length of the distributions. Call the resulting number $C$. Note that $C$ will always be between 0 and 1.

If $C$ was less than 0.5, we declared the two experts to be uncorrelated. Otherwise, we set the correlation factor between the two experts to be $C$. Note that this decision is rather arbitrary; we could have set the correlation factor to be $C^2$, or set the cutoff value to be 0.7 instead of 0.5, or had no cutoff value at all. As stated in the chapter on correlation, there is no simple way to determine the best formula. From a qualitative point of view, the importance of the cutoff value is to keep the effects of correlation only to closely related experts. Remember that if the correlation values are too high across the board, they can actually hinder performance by providing false estimates for entire classes of experts.
This formula worked very well in some instances, but in other instances, it led to very undesirable performance. We noticed through experimentation that since there was absolutely no correlation between the two classes, a bad performance by a couple of experts in a class could drag down the entire class, and limit exploration of that class. We fixed this problem with the addition of a single nonzero correlation value.

Let us denote the distribution that sets one bot to the \( x \) most frequented buckets, where \( x \) is the number of bots, to be distribution 1. We will call the histogram expert which utilizes this distribution \( H_1 \), and the recency expert which utilizes the same distribution \( R_1 \). In order to fix the problem described above, we added a correlation value of 0.25 between \( H_1 \) and \( R_1 \). This small value acts as a "bridge" between classes and allows the algorithm to avoid fixating on one class of experts.

![Figure 4-1](image)

(a) No bridging correlation value  
(b) Bridging correlation value added

Figure 4-1: Performance of algorithm with different correlation factors. Note that when we add the "bridge" value between \( R_1 \) and \( H_1 \), there were no iterations that failed to converge to a good expert within the simulation time limit.
4.4 Running Simulation 2

The simulation described above was run in a host of different configurations in order to determine the true usefulness of the correlated algorithm. We had six different opponents for our algorithm to play against, labeled 1 to 6. Then, we had six different quantities of experts, ranging from 10 to 3150. Additionally, for each configuration above, we ran three different simulations: the single best expert, the correlated algorithm, and the uncorrelated algorithm. Finally, since this is a stochastic simulation, we ran each combination of opponent, expert set, and simulation 50 times. Each run of a simulation consisted of 50,000 stages.

4.4.1 Opponents

There are six different opponents, each designed to favor one of the experts over all others. As we will see in the analysis section, some opponents accomplish this task better than others. All opponents attempt to generate tasks at approximately the same rate that the bots can handle them when acting optimally. Since the score for each task is the inverse of the task generation probability, the optimal expert should have an average score of 1, as discussed above. Since some of these opponents are stochastic in nature, this mark isn’t always hit, and in some cases the performance of the optimal expert may be as low as .8 points per turn.

The reader may notice that a number of these opponents are small variations of each other. We must keep in mind that while the main purpose of this thesis is to display the power of the correlated algorithm in certain situations, we also have the desire to show the flexibility and adaptability of experts algorithms in general. Running these simulations, it is interesting to watch the bots mill around at first, but quickly start to gravitate towards the tasks, and even hover around the locations where the tasks are likely to pop up!

The first opponent is a stochastic opponent that places all tasks exclusively in four areas of the screen. Specifically, the four favored spots form the vertices of a square. Each time a task is generated, it has an equal probability to be in any of
the four areas. Additionally, within these areas there is a small amount of (uniformly distributed) random variation for the exact location of the task. The optimal expert for this opponent is the histogram expert which divides 1/4 of the bots to each of the four most frequented buckets.

The second opponent is a deterministic opponent. It generates its tasks continually in a circular pattern, with the diameter of the circle being nearly the length of the field. The number of points on the circle is equal to the number of bots on the field. As a result, this opponent greatly favors the evenly distributed histogram expert, $H_1$ as defined above.

The third opponent is a time-varying version of the first opponent. It favors the same areas as the first opponent, and has the same small stochastic variations in exact position. However, instead of generating a task in each of the four areas with equal probability, it focuses all of its tasks on a single area for large number of turns. At some point, it rotates to the next location, and generates all of its tasks there for a long time. This process continues, essentially looping back on itself. Against this opponent, the best expert is $R_1$, the recency expert with distribution 1.

The fourth opponent is also a stochastic opponent that moves its favored area over time. However, this movement occurs much faster than opponent 3. This opponent generates its tasks with a uniformly random x-coordinate, but a y-coordinate based on the stage number. The effect is to generate a “sweep” of tasks that travel up the screen. Once again, $R_1$ is the best expert for this opponent.

The fifth opponent is a slight variation on the fourth opponent. Instead of having a uniformly random x-coordinate, each task has an equal chance of having one of two x-coordinates. Visually, the effect is two dotted lines of tasks drawn vertically up the screen. $R_1$ is again the optimal expert here, as it was in opponent 4.

Finally, the sixth opponent has a grid of locations that it favors. It chooses an evenly distributed $x$ points on the field, with $x$ again being the number of bots. It then generates tasks with equal probability at any of these points. With this opponent, as in opponent 2, $H_1$ is the optimal expert. This opponent also shares other properties with opponent 2, which will be discussed in the results section.
### Distribution Length of Experts

<table>
<thead>
<tr>
<th>Distribution Length</th>
<th># of Experts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>44</td>
</tr>
<tr>
<td>12</td>
<td>154</td>
</tr>
<tr>
<td>16</td>
<td>462</td>
</tr>
<tr>
<td>20</td>
<td>1254</td>
</tr>
<tr>
<td>24</td>
<td>3150</td>
</tr>
<tr>
<td>28</td>
<td>7436</td>
</tr>
</tbody>
</table>

Table 4.1: Number of experts generated with each distribution length

#### 4.4.2 Expert Sets

As described before, all of the experts in this simulation have a class and a distribution. This distribution is simply an array of numbers which sum to the total number of bots on the field. It represents the manner in which an expert will distribute the agent's bots over the most interesting locations; with the recency experts, these locations are simply the $x$ most recently generated tasks, whereas for the histogram experts, these are the $x$ most frequently populated buckets on the field. Again, $x$ is the number of bots on the field.

By design, the length of a distribution is equal to the number of bots on the field. This allows for every possible distribution, from one that's simply $x$ followed by $x - 1$ zero's, to one that is simply $x$ one's.

As a result, one clear way to increase the number of experts is to simply increase the length of the distributions. (Note that this change also has broader effects, since it allows experts to consider additional locations. This effect is described later in "Additional Variables at play.") We ran our simulation with six different distribution lengths: 4, 8, 12, 16, 20, and 24. The number of distributions generated from each length, and the corresponding number of experts created, are summarized in table 4.1. Remember that we only allow monotonically decreasing distributions.

Note that as the distribution length increases linearly, the number of experts increases combinatorially. As we will see, this combinatorial increase in the size of the expert set can be a big problem for the uncorrelated algorithm.
4.4.3 The Algorithms vs. the Best Expert

For each combination of six different opponents and six different expert sets, we must run three separate simulations. For ease of comparison, each of the opponents was constructed in such a way that the best expert is logically clear. Thus, we are able to run one simulation where we simply “cheat” and exclusively play the best expert. After that, we run two additional simulations: the correlated algorithm, and the uncorrelated algorithm. In terms of implementation, the uncorrelated algorithm works exactly the same as the correlated algorithm, except $\hat{M}_e$ is never computed, and we simply use $M_e$ to determine the best expert for purposes of exploitation.

4.4.4 Additional Variables at Play

In a controlled experiment, we want to keep as many variables as possible constant, so that we are better able to test the particular issues that we are considering. In this case, the single variable in which we are interested is the size of the expert set. The intent of this experiment is to note the difference between the performance of the correlated algorithm and the uncorrelated algorithm, and the effect on the performance of these algorithms as we increase the number of experts in play.

However, the problem presented in this case, while an interesting application of Experts Algorithms, is far from ideal in this sense. One issue already mentioned concerns the distributions. While increasing the length of distributions is an excellent, straightforward way to quickly increase the size of the expert set, it also fundamentally changes the experts themselves. When the distribution for an expert increases in length by 1, that expert becomes able to consider 1 additional location of interest. For example, a recency expert with distribution of length 5 can consider up to the 5 most recently generated targets, whereas a recency expert with distribution of length 6 could possibly consider the 6 most recently generated targets.
Distribution Length and the Number of Bots

We also must ask what happens if the number of bots and the length of the distribution were different. Remember that the sum of all the elements in the distribution must be equal to the number of bots on the field. Remember, also, that we only allow monotonically decreasing distributions. So now, let us consider two cases; the case where the distribution length is higher than the number of bots, and the case where the distribution length is lower than the number of bots.

If we set the distribution length higher than the number of bots, then we immediately see a problem. Since we only allow monotonically decreasing distributions, we cannot actually create any additional distributions by increasing the distribution length! Given \( x \) bots, the longest possible distribution that we can have is a row of \( x \) 1's. Thus, setting a distribution length which is larger than the number of bots has no effect.

If we set the distribution length lower than the number of bots, then we have the reverse issue. Basically, since we still constrain the distribution to sum to the number of bots, we end up simply sending the extra bots to duplicated targets. Thus, the bots on the field end up acting as if they were simply fewer in number, in terms of actually accomplishing tasks. Another issue with this situation is that it slows down the rate of increase of the size of the expert set as we increase the distribution length.

At this point, the reader may ask, why do we require that the distribution sum to the number of bots? Since the sum of the distribution equals the number of targets created by an expert, we could simply allow experts to recommend more or fewer targets than there are bots, and let the agent figure out how to assign bots. Suffice it to say that this also runs into design problems. If there are fewer targets than bots, then bots once again simply end up acting as duplicates. On the other hand, if there are more targets than bots, then the question of how the agent decides how to assign its bots to targets becomes very important. Further, the performance of large groups of experts starts to merge together, since any two experts that are similar to each other may perform identically if the agent assigns its bots in a certain fashion.
Given all of these issues, it is clear that the most advantageous situation is to simply tie the number of bots on the field to the length of the distribution. Of course, this design choice introduces additional effects as we increase our distribution length.

Most immediately, increasing the distribution length and the number of bots on the field also increases the number of tasks that the agent can process concurrently. Remember that in order to accomplish a task, a bot has to remain at that task's location for a number of stages — in this case, 50. Thus, on average, our agent can only process \( \frac{50}{50} \) tasks per stage. Any tasks generated beyond that will go unaccomplished, no matter how good our experts are. Since we want to have each different simulation run on approximately equal footing, it makes sense to scale the task generation probability for our opponents with the distribution length \( L \).

These are all the factors which we must vary in different simulations. However, there are other factors that deserve additional discussion. These factors are kept constant throughout the entire experiment, but we must note that their relative effect does change when we change the parameters of the experiment. These factors are detailed below.

**Exploration and Exploitation Particulars**

One factor kept constant throughout the entire experiment is \( p_i \), the probability with which we conduct an exploration phase. The function we chose here was \( p_i = i^5 \). We easily could have chosen \( i^3 \) or \( i^7 \), but this value provided the best performance tradeoff for our experiment.

There is a basic tradeoff that one must keep in mind when it comes to exploration. More exploration (i.e. more slowly degrading probability of exploration) gives a higher probability that the algorithm will find a very good expert relatively soon. (We are guaranteed, of course, to find the best expert in the limit.) However, more exploration also tends to result in lower average performance, because even if an algorithm does discover the best expert, it will spend more time exploring other experts and therefore lowering its score. Less exploration gives a higher average score when the algorithm finds the best expert, but also occasionally causes the algorithm to take a very long
time to find that expert.

Also, the reader should note that the formula for calculating the exploration probability does not depend in any way on size of the expert set. On the other hand, one might expect that with more experts, we would need more phases of exploration in order to explore the same fraction of the expert set. Indeed, this effect is seen in the experiment. As the number of experts increases, the behavior of the algorithm slowly changes as though we were decreasing the amount of exploration. Thus, even though we never change this formula throughout our experiment, the effects of this choice cannot be ignored.

Another design choice that should be mentioned here involves the modified exploitation phases. As discussed in the correlation section, the exploitation phases in our experiment are not "true" exploitation phases because they do not solely play the best-performing expert so far. Instead, they perform a very narrow exploration between the $N$ highest scoring experts. The value for $N$ in our experiment was chosen to be 6. Additionally, we chose to increase the probability that better scoring bots would be chosen. To do this, we had our algorithm choose its expert probabilistically based on the experts' scores raised to the 8th power, rather than the raw scores.

The effects of these choices are subtle, and not fully understood. The effect of raising the experts' scores to a higher power is to heighten the probability that the very best scoring experts will be picked over the $N_{th}$ best scoring expert. The effect of increasing $N$ is to increase the amount of exploration done during the modified exploitation phase. Both of these values were chosen after some experimentation.

Finally, a last constant which should be discussed is a past-discounting factor. We must remember that many opponents appear at first to favor one class of experts, but in the long run actually favor a different class of experts. In order to avoid penalizing the CEA for its early estimation, we discount the past estimation of an expert when we update it. This discount factor, $\gamma$, can have a range of $(0, \infty)$. A value of 1 corresponds to a proper average of past and current phases. A value less than one corresponds to a discount on the past, and a value greater than one corresponds to a discount on the present. The value chosen for this experiment is $\gamma = 0.75$. 

41
4.5 Analysis

In order to analyze the data from this experiment, we will separate the opponents into groups, according to the performance of our algorithms. We will discuss these groups in order, starting with the group in which the CEA had the most favorable performance, and ending with the group in which the CEA's performance was least favorable. All result data will be included in Appendix A for reference purposes.

In general, the correlated algorithm outperformed the uncorrelated algorithm in those cases where there is low concentration of highly performing experts. Additionally, in these cases, the performance of the uncorrelated algorithm dropped significantly as the size of the expert set increased, whereas the performance of the correlated algorithm was unaffected.

4.5.1 Group A opponents

Opponents 2 and 6 fall into this group. Group A opponents are the Correlated Experts Algorithm's chance to shine. In group A, we see all the effects that we set out to accomplish: as the size of the expert set increases, the performance of the uncorrelated algorithm drops drastically, while the performance of the correlated algorithm seems unaffected.

One major reason for these promising results is that with these opponents, there are very few highly-performing experts. This can be clearly seen in figure ??; for both of these opponents, fewer than 10 experts performed higher than 0.7. This is the key to the improved performance for the correlated algorithm.

Without correlation, the EEE does manage to find an expert that performs reasonably well, simply through plain exploration. However, once the uncorrelated algorithm finds a good expert e, its performance will be capped by the value of expert e until it happens by chance to explore a better expert. Further, even if it randomly chooses an expert which is better than expert e, it must play that expert for long enough to recognize that that expert is better than expert e.

On the other hand, when correlation is used, when the algorithm finds a good
expert $e$, it tries $e$'s neighbors with high probability, so it is very likely to find an expert near $e$ which performs better than $e$. In this way, the CEA is very efficient at "stepping up" from good experts to the best expert. Thus, in simulation, our correlated algorithm settled on the best expert nearly every time, while the uncorrelated algorithm usually found a good expert but rarely found the best expert.

On a related note, the uncorrelated algorithm usually did not manage to find the same expert on each iteration. This effect is illustrated in the individual performance graphs. Note that the performance of the CEA generally had very low variance, since it usually converged to the best expert in the set. On the other hand, the EEE had very high variance, particularly in the case of the class A opponents. This relates to the fact that the performance of the EEE was usually based on a single random discovery of a good expert, and did not quickly rise beyond the average performance of that expert.

In the case of opponents 2 and 6, as seen in figure A-1, there is a real difference in performance between the best expert and the "good" experts. Further, as the number of experts increases, the fraction of highly performing experts in the set decreases proportionately. As a result, the performance of the EEE drops as the number of experts increase, while the performance of the CEA is amazingly unaffected.

4.5.2 Group B opponent

Opponent 4 belongs to this group. The correlated algorithm also outperformed the uncorrelated algorithm against this opponent, and the difference in performance increased as the number of experts increased. However, starting at the expert set of size 154, the difference in performance of the two algorithms stays about constant. The reason for this is simple, and it can be observed in figure A-1. Although opponent 4 has a low concentration of highly performing experts, it is not as selective as the opponents in group A.

Further, we must remember that as the size of the expert set increased, the number of bots on the field and the number of targets considered by each expert increased as well. Considering the behavior of opponent 4, another trend develops. As we increase
the number of experts, more and more good experts are produced, and so we do not
decrease the fraction of highly-performing experts in the set. Thus, although there
are more experts to consider, the uncorrelated algorithm has approximately the same
probability of finding a good expert as it did before.

4.5.3 Group C opponents

Two of our opponents garnered the same average performance from both the corre-
lated and uncorrelated algorithms. These were opponents 3 and 5. Once again, the
explanation can be seen in figure A-1. As opposed to group A or B experts, these
opponents have a high concentration of highly performing experts. Further, this con-
centration does not decrease as the size of the expert set increases. Thus, the “good”
expert settled upon by the EEE performs nearly as well as the best expert, which is
settled upon by the CEA.

4.5.4 Group D opponent

We will denote the group where the CEA showed the worst performance, group D.
Opponent 1 is the only opponent belonging to this group, and it is the only opponent
where the average performance of the CEA was worse than the average performance
of the EEE.

Looking at figure A-1, opponent 1 looks like it should fall in line with the group C
opponents. But this is not the case. In order to understand why this is so, we must
understand a phenomenon we call “lag time.”

The concept of lag time is specific to this problem, though a similar effect may be
present in many different types of problems. Recall that in this problem, in order to
accomplish a task and receive a score, a bot must remain at that task for a number of
stages. Thus, bots can only really accomplish tasks when they are at their assigned
location. If our agent is playing one expert and then selects another expert, any bots
that receive reassignments must shift to their newly assigned location before they will
be of any use in scoring. As a result, depending on how different the newly chosen
expert is from the last one, there may be a dip in the performance for a period of time as the bots move to their new locations. The average length of this period is called the lag time.

In the case of opponents 3 and 5, the lag time between good experts and the best expert is relatively small. This is because the best experts are of the recency class, and newly generated targets are generally close to the other recently generated targets.

In the case of opponent 1, the lag time between the good experts and the best expert is very large. This opponent generates targets in four different areas. Thus, the best experts for this opponent are in the histogram class. But since these four areas are very far apart, each time we select a new histogram expert, one or more bots must travel a significant distance before reaching its assigned location and helping our agent score.

We have established that the lag time of opponent 1 is larger than that of opponents 3 or 5. But why does this cause the correlated algorithm to perform more poorly than the uncorrelated algorithm? The answer is exploration. Because the correlated algorithm is designed to play neighbors of good experts with high probability, it actually performs much more exploration (within a limited set of good experts) than the uncorrelated algorithm.

As we said before, once the uncorrelated algorithm finds a reasonably good expert, it tends to stick with it. (The lag time is actually also a factor in this effect, since an algorithm must explore a great expert for longer than the lag time in order to recognize it as a great expert.) In the case of opponent 1, the reasonably good expert is good enough to produce the performance seen in figures A-2 and A-20.

The CEA, on the other hand, upon finding a good expert, continually tries to do better. Because the lag time is so large for this opponent, the CEA is effectively punished for this additional exploration.
Chapter 5

Conclusion

As with most research done in academia, the work presented in this thesis introduces more questions than it answers. Experimental results show that, given the right circumstances, the CEA can capture enough information through correlation to perform much better than the uncorrelated EEE algorithm. Indeed, with a proper formula for correlation factors, the performance of the CEA seems to be independent of the number of experts considered in the algorithm.

This is a very promising result, since it opens the door to new possible solutions to many problems. In many problems, a better approximation of the optimal behavior often comes with a tradeoff in problem complexity and convergence rate. In this case, rather than increasing the complexity of our expert, it seems we are free to simply add vast numbers of simpler experts to solve the problem.

Of course, there is much work that still needs to be done. One large problem left unaddressed in this paper is actual process time of the algorithm. All of the data for our experiments are based on a constant number of stages. However, as we add experts to our set, the average computation per stage becomes greater, since we must check and update more correlated experts at the end of each phase. There are many possible avenues to explore towards tackling this issue; expert sampling is one example.

Another large open question with the CEA is determining a better means to compute the correlation factors themselves. Right now, the correlation factors are
designed by hand, a priori, and we do not even have a guarantee that these factors result in the best possible performance. Indeed, we do not have any guarantee that the best correlation factors for one environment would be optimal in another environment, given the same set of experts. A good solution for both problems is to discover an efficient method to calculate correlation factors on-line. We believe a means of doing this will be forthcoming with more research.

A proof of performance is also an open question. A proof that shows that the correlated algorithm performs better than the best expert in the limit is already completed, but there is no proof yet that the CEA will perform any better than the EEE. Given the results seen in this paper, it seems that with certain assumptions about our environment, we should be able to prove better convergence bounds than we have for the EEE.

These are just some examples of the research that must still be done. Nonetheless, the results in this paper make it clear that correlation information can be utilized to great effect, in the proper situation. And since experts algorithms can so easily be applied in a great many areas, many such situations will arise. Indeed, with the additional power of correlation, the Experts Algorithm may become a feasible solution for many new types of problems.
Appendix A

Data

This appendix includes all experimental data taken, as described in the experiment section.
Figure A-1: Distributions of expert performance for each opponent. All tests were conducted with the set of 462 experts. Each expert was run exclusively for 20,000 turns, and the final average score was recorded as that expert’s value. The height of each bar represents the number of experts whose value was near the accompanying value on the x axis.
Figure A-2: Average performances against opponent 1. The red line marks the best expert, the blue line marks the correlated algorithm, and the green line marks the uncorrelated algorithm.
Figure A-3: Average performances against opponent 2. The red line marks the best expert, the blue line marks the correlated algorithm, and the green line marks the uncorrelated algorithm.
Figure A-4: Average performances against opponent 3. The red line marks the best expert, the blue line marks the correlated algorithm, and the green line marks the uncorrelated algorithm.
Figure A-5: Average performances against opponent 4. The red line marks the best expert, the blue line marks the correlated algorithm, and the green line marks the uncorrelated algorithm.
Figure A-6: Average performances against opponent 5. The red line marks the best expert, the blue line marks the correlated algorithm, and the green line marks the uncorrelated algorithm.
Figure A-7: Average performances against opponent 6. The red line marks the best expert, the blue line marks the correlated algorithm, and the green line marks the uncorrelated algorithm.
Figure A-8: Individual performances for the best expert against opponent 1. The red line is the average. In this case, the best expert is the histogram expert with distribution $[\frac{X}{4}, \frac{X}{4}, \frac{X}{4}, \frac{X}{4}, 0, \ldots]$ where $X$ is the number of bots on the field.
Figure A-9: Individual performances for the best expert against opponent 2. The red line is the average. In this case, the best expert is $H_1$, the histogram expert with distribution $[1, 1, 1, \cdots]$
Figure A-10: Individual performances for the best expert against opponent 3. The red line is the average. In this case, the best expert is the recency expert with distribution [1, 1, 1, ∞].
Figure A-11: Individual performances for the best expert against opponent 4. The red line is the average. In this case, the best expert is the recency expert with distribution \([1, 1, 1, \ldots]\).
Figure A-12: Individual performances for the best expert against opponent 5. The red line is the average. In this case, the best expert is the recency expert with distribution $[1, 1, 1, \cdots]$
Figure A-13: Individual performances for the best expert against opponent 6. The red line is the average. In this case, the best expert is the histogram expert with distribution $[1, 1, 1, \cdots]$
Figure A-14: Individual performances for the CEA against opponent 1. The red line is the average.
Figure A-15: Individual performances for the CEA against opponent 2. The red line is the average.
Figure A-16: Individual performances for the CEA against opponent 3. The red line is the average.
Figure A-17: Individual performances for the CEA against opponent 4. The red line is the average.
Figure A-18: Individual performances for the CEA against opponent 5. The red line is the average.
Figure A-19: Individual performances for the CEA against opponent 6. The red line is the average.
Figure A-20: Individual performances for the uncorrelated algorithm against opponent 1. The red line is the average.
Figure A-21: Individual performances for the uncorrelated algorithm against opponent 2. The red line is the average. Notice the large variance. This results from the EEE's tendency to stick with the first good expert it finds for a long time. In this case, there is a large difference between good experts and the best expert.
Figure A-22: Individual performances for the uncorrelated algorithm against opponent 3. The red line is the average.
Figure A-23: Individual performances for the uncorrelated algorithm against opponent 4. The red line is the average.
Figure A-24: Individual performances for the uncorrelated algorithm against opponent 5. The red line is the average.
Figure A-25: Individual performances for the uncorrelated algorithm against opponent 6. The red line is the average.
Appendix B

Simulation Code

B.1 Agent.java

```java
package jeremy;

import static jeremy.TargetManager.*;

import java.util.*;

public class Agent {

    private List<Bot> bots;
    private ExpertList experts;
    private Expert currExpert;
    private int phase = 1, stage = 0, stagesInAll = 0;
    private double M = 0;
    private final double alpha = .997;
    private double alphaPow = 1;
    private double avgScore;

    // ******************************************************
    // Constructors
    // ******************************************************
```
public Agent()
{
    int i, xpos, ypos;
    float radians = 0;
    bots = new ArrayList<Bot>();

    for (i = 0; i < NUMBOTS; i++)
    {
        radians += 2 * Math.PI / (NUMBOTS);
        xpos = (int) Math.round(100 * Math.cos(radians));
        ypos = (int) Math.round(100 * Math.sin(radians));
        bots.add(new Bot(xpos, ypos));
    }

    experts = new ExpertList();
    System.out.println("Initialization complete!");
    currExpert = chooseExpert();
}

// Utility Functions
public List<Bot> getBotList()
{
    if (bots == null)
        throw new RuntimeException("bots is null");
    return bots;
}

// Function: play
// Purpose: Takes in the list of real targets, and decides where to
// move all of the bots.
// Notes: This function is run once each stage.
public void play(List<Target> realTargets)
{
    // we will want to play the chosen expert for Ni stages. Notice
    // currExpert has been initialized before the first round.
    List<Target> givenTargets;

    givenTargets = currExpert.advise(realTargets, bots);

    // first we must check to make sure we have at least as many ~
    // targets
    // as we have bots. If not, then we want to duplicate the target
// list until we do.
for (;;)
{
    if (givenTargets.size() < bots.size())
        givenTargets.addAll(new ArrayList<Target>(
            givenTargets));
    else
        break;
}

// This is the current assignment algorithm. It is greedy, simply
// matching each phantom target with the nearest bot. To be a little
// bit smarter, once it assigns a bot to a target, it will hold that
// assignment until that target disappears.
double direction, distance, minDistance;
Target chosenTarget = null;
for (Bot myBot : bots)
{
    minDistance = 2 * FIELDSIZE;
    direction = 0;

    // if this bot is headed towards a legit target, then
    // we let it keep going.
    if (givenTargets.contains(myBot.getDestination())
        chosenTarget = myBot.getDestination();
    else
        { 
            for (Target myTarget : givenTargets)
            {
                distance = Location.getDistance(myBot, myTarget);
                if (distance < minDistance)
                {
                    chosenTarget = myTarget;
                    minDistance = distance;
                }
            }
            myBot.setDestination(chosenTarget);
        }
    if (chosenTarget == null)
throw new IllegalArgumentException("No Target Chosen");
direction = Location.getDirection(myBot, chosenTarget);
myBot.moveBot(direction);
givenTargets.remove(chosenTarget);
}

// Function: updateScore
// Purpose: This function receives the agent's score from the target
// manager. Since this means that the agent has finished a stage, this
// function first increments its stage, and then updates its experts
// accordingly. If it reaches the end of a phase, then it resets the stage
// values and chooses a new expert.
public double updateScore(double score) {
    double Ni = currExpert.getNi();
    avgScore += score;
    stage++;
    stagesInAll++;

    if (DISPLAY) {
        try {
            Thread.sleep(TIMEDELAY);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    if (stage == Ni) {
        // we've reached the end of the phase.
        // Now we want to update the our estimation for this expert.
        avgScore /= Ni;
        // System.out.println("avgScore = " + avgScore);

        // M is a running average score for the agent.
        M = M + (Ni / stagesInAll) * (avgScore - M);
        currExpert.update(avgScore, Ni, 1, 1);
        // The third term here is Cef, and the fourth
// term is alpha^-i. Both these terms are used
// when dealing with correlated experts, so we
// just set them to 1.

// This is where we use our correlation information to 
// generalize
// our
// knowledge about other experts.
if (CORRELATION)
{
    for (Expert myExpert : experts)
    {
        if (!currExpert.equals(myExpert))
        {
            // note that we are now using the 
            "cheating" correlation
            // vals.
            myExpert.update(avgScore, Ni, 
            currExpert.correlate(myExpert) 
            , alphaPow);
            //myExpert.update(avgScore, Ni, 
            experts.getC(currExpert, 
            // myExpert), 
            alphaPow);
        }
    }
}

// now we want to increment the number of phases that this 
// expert has been played.
currExpert.increment();
// Then we move to the next phase, then 
// choose a new expert and reset our values.
phase++;
if (CHEATING == true)
    // we're cheating for the best expert.
currExpert = cheatForBestExpert();
else if (EXPERTTEST == true)
    currExpert = experts.getListOfExperts().get( EXPERTNUM);
else
    currExpert = chooseExpert();

stage = 0;
avgScore = 0;
// Finally, we want to slowly decrease the effectiveness of
// our correlation values.
alphaPow *= alpha;

return M;

// *******************************************************
// Function: chooseExpert
// Purpose: chooses an expert to play for the next phase.
// Notes: once an expert is chosen, it's played for n[i] rounds, where i
// is the number of times the expert has been played.
private Expert chooseExpert()
{
  double p;
  Expert e;

  Random rand = new Random();
  // note that phase is initialized to 1.
  p = Math.pow(1.0 / phase, .5);
  if (rand.nextDouble() < p)
  {
    // then we want to explore.
    e = experts.getRandomExpert();
    // System.out.println("random expert is: "+ e);
    return e;
  }

  // else we want to pick a good expert.
  e = pickAGoodExpert(getNBestExperts(6));
  // System.out.println("chosen expert is: "+ e);
  return e;
}

// *******************************************************
// Function: cheatForBestExpert
// Purpose: cheats for the best expert in this game. This is used
// as a comparison against the algorithm.
private Expert cheatForBestExpert()
{

  int i;
  Expert e = null;
  List<Integer> tmp = new ArrayList<Integer>();
switch (OPPONENT) {
    case 1:
        for (i = 0; i < 4; i++)
            // note DISTRO_SIZE must be a multiple of 4 for this to work.
            tmp.add(DISTRO_SIZE / 4);
        for (; i < DISTRO_SIZE; i++)
            tmp.add(0);
        e = experts.getHistogramExpert(tmp);
        break;
    case 2:
    case 6:
        for (i = 0; i < DISTRO_SIZE; i++)
            tmp.add(i);
        e = experts.getHistogramExpert(tmp);
        break;
    case 3:
    case 4:
    case 5:
        for (i = 0; i < DISTRO_SIZE; i++)
            tmp.add(i);
        e = experts.getRecencyExpert(tmp);
        break;
    default:
        throw new IllegalArgumentException("impossible opponent");
}
// System.out.println("cheating expert is: "+ e);
return e;
}

private Expert pickAGoodExpert(List<Expert> nBestExperts) {
    int i;
    double expertValues[] = new double[nBestExperts.size()];
    double total = 0, soFar = 0;
    Random rand = new Random();
    for (i = 0; i < nBestExperts.size(); i++)
expertValues[i] = Math.pow(nBestExperts.get(i).getExpectedValue(), 8);
total += expertValues[i];

if (total == 0)
    System.out.println("total is 0");
for (i = 0; i < nBestExperts.size(); i++)
{
    soFar += expertValues[i] / total;
    if (rand.nextDouble() <= soFar) // then we want to return this
        // expert.
        return nBestExperts.get(i);
}
throw new RuntimeException("Something is wrong in pickAGoodExpert");

// ******************************************************************************
// Function: getNBestExperts
// Purpose: returns the N experts that currently have the highest expected value. If there aren't enough experts, we simply return all the experts.
// Notes: This function also calls the expert sort function, which has the aesthetic benefit of sorting the experts by score (for debugging purposes).

public List<Expert> getNBestExperts(int N)
{
    experts.sort();
    List<Expert> copy = experts.getListOfExperts();

    // System.out.printf("best %d experts:
% s \n", copy.get(0).
    // N));
    if (DISPLAY)
        System.out.printf("best experts:
% s \n", copy.subList(0, N));
    // else
    // System.out.printf("best expert:
% s \n", copy.get(0));

    N = Math.min(N, experts.numExperts());
    return copy.subList(0, N);
Function: getMostPlayedExpert
// Purpose: returns the expert that has been played for the most phases.
// Notes: This function is used solely for readout purposes.

```java
class Bot extends Locatable {
    private Location loc;
    private boolean playing = false;
```

B.2 Bot.java
private Target destination;

public static DecimalFormat threeSigs = new DecimalFormat("#.##");

// ************
// Constructors

public Bot()
{
    this(new Location());
}

public Bot(int xgiven, int ygiven)
{
    this(new Location(xgiven, ygiven));
}

// creates a copy of given location in order
// to avoid the accidental modification of an
// externally defined location
public Bot(Location givenLoc)
{
    loc = new Location(givenLoc);
}

// ************
// Utility functions

public void setLoc(double setx, double sety)
{
    loc.setLoc(setx, sety);
}

// creates a copy of given location in order
// to avoid the accidental modification of an
// externally defined location
public void setLoc(Location givenLoc)
{
    if (givenLoc == null)
        throw new IllegalArgumentException("something is wrong in setLoc");
    loc = new Location(givenLoc);
}

// returns a defensive copy of loc. This way
// we don't have to make loc immutable to ensure
// its safety from the outside world.
public Location getLoc()
{
    return new Location(loc);
}

public boolean isPlaying()
{
    return playing;
}

public void setPlaying(boolean setval)
{
    playing = setval;
}

public Target getDestination()
{
    return destination;
}

public void setDestination(Target destination)
{
    this.destination = destination;
}

// *****************************************************
// Function: computeCentroid
// Purpose: computes the Location of the centroid for the given
// array of Bots.
// Notes: The most common use of this function is to compute the
// centroid of allBots.
public static Location computeCentroid(Bot[] someBots)
{
    Location[] locs = new Location[someBots.length];
    int i = 0;
    for (Bot myBot : someBots)
    {
        locs[i] = myBot.getLoc();
        i++;
    }
    return Location.getCentroid(locs);
}
public void moveBot(double direction) {  
    int movex, movey;
    movex = (int) Math.round(2 * Math.cos(direction));
    movey = (int) Math.round(2 * Math.sin(direction));
    // we only want to move the bot if it stays within the boundaries.
    if (Math.abs(loc.getX() + movex) <= FIELDSIZE / 2
        && Math.abs(loc.getY() + movey) <= FIELDSIZE / 2)
        setLoc(loc.getX() + movex, loc.getY() + movey);

    // System.out.println("Player's location: "+loc.getXO + " " + loc.getYO);
}

public void drawBot(Graphics g) {

    if (playing)
        g.setColor(Color.green);
    else
        g.setColor(Color.blue);
    g.fillRect((int) (FIELDSIZE / 2 + loc.getX() + 2), (int) (-FIELDSIZE / 2 - loc.getY() + 2), 4, 4);
}
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.List;
import javax.swing.JPanel;
import static jeremy.TargetManager.*;

@SuppressWarnings("serial")
public class DisplayPanel extends JPanel {
  private List<Bot> allBots;
  private List<Target> realTargets;
  private List<Target> phantomTargets;

  public DisplayPanel(List<Bot> allBots, List<Target> realTargets) {
    this.allBots = allBots;
    this.realTargets = realTargets;
    Timer timer = new Timer(25, new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        repaint();
      }
    });
    timer.start();
    if (this.allBots == null)
      throw new RuntimeException("here it is");
  }

  public void setPhantomTargets(List<Target> phantomTargets) {
    this.phantomTargets = phantomTargets;
  }

  // *****************************************************
  // Function: paint
  // Purpose: paints a picture inside the display panel.
  // Notes: this function overloads the paint function in JPanel.
  public void paint(Graphics g)
B.4 DisplaySwing.java

```java
package jeremy;

import javax.swing.*;
import java.util.List;
import static jeremy.TargetManager.*;

public class DisplaySwing implements Runnable {
    /**
     * Create the GUI and show it. For thread safety, this method should be
     * invoked from the event-dispatching thread.
     */
    // world[] corresponds to allBots[]
    private static void createAndShowGUI(List<Bot> allBots,
                                          List<Target> realTargets) {
        // Create and set up the window.
        JFrame frame = new JFrame("DisplaySwing");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // frame.setLocation(1200, 400);
        DisplayPanel dPanel = new DisplayPanel(allBots, realTargets);
        frame.getContentPane().add(dPanel);
        // Display the window.
    }
```
frame.pack();
frame.setBounds(700, 100, FIELDSIZE + 10, FIELDSIZE + 35);
frame.setVisible(true);

public static void showDisplay(List<Bot> allBots, List<Target> realTargets)
{
    // Schedule a job for the event-dispatching thread:
    // creating and showing this application’s GUI.
    javax.swing.SwingUtilities.invokeLater(new DisplaySwing(allBots, realTargets));
}

private List<Bot> allBots;
private List<Target> realTargets;

public DisplaySwing(List<Bot> allBots, List<Target> realTargets)
{
    this.allBots = allBots;
    this.realTargets = realTargets;
}

public void run()
{
    createAndShowGUI(allBots, realTargets);
}

B.5 DistributionExpert.java

package jeremy;
import static jeremy.TargetManager.*;
import java.util.*;
public abstract class DistributionExpert extends Expert
{
    private List<Integer> distribution:
public String toString()
{
    return " with distribution " + distribution + "\n";
}

@Override
public abstract List<Target> advise(List<Target> realTargets,
        List<Bot> myBots);

//  *****************************************************
//  Constructors

DistributionExpert(List<Integer> distribution)
{
    this.distribution = new ArrayList<Integer>(distribution);
}

//  *****************************************************
//  Utility Functions

public void setDistribution(List<Integer> distribution)
{
    this.distribution = distribution;
}

public List<Integer> getDistribution()
{
    return new ArrayList<Integer>(distribution);
}

//  *****************************************************
//  Function: correlate
//  Purpose: produces a correlation value between this expert and
//  the argument expert. This function should fail if the two experts
//  are not of the same class.

@Override
public double correlate(Expert e) {
{
    double C;
    if (!(e instanceof DistributionExpert))
    {
        System.out.println("experts must be distribution experts!");
        return 0;
    }
}
double diff = getDifference((DistributionExpert) e);
if (!e.getClass().equals(getClass()))
    if (diff == 0 && distribution.get(0) == 1)
        return 0.25;
    else return 0;
    // throw new IllegalArgumentException("experts must be of the same class!");
    // maximum value of diff is (MAXTARGETS-1)+(NUMBOTS-1), assuming we
    // still limit our distributions to be monotonically decreasing.
    double maxDiff = DISTRO_SIZE*2 - 2;
    // double maxDiff = DISTRO_SIZE * 2;
    if (diff > maxDiff || diff < 0)
        throw new IllegalArgumentException("something is wrong with diff");
    C = (maxDiff - diff) / maxDiff;
    if (C < .5)
        return 0;
    // C = Math.pow(C, 3);
    return C;
    // return 0;
}

// *******************************************************
// Function: getDifference
// Purpose: sums the element-by-element difference between the
distributions
// of the experts.
// Notes: this function just assumes that the two experts are of the same
// class.
private int getDifference(DistributionExpert e)
{
    int diff = 0;
    for (int i = 0; i < distribution.size(); i++)
        diff += Math.abs(distribution.get(i) - e.distribution.get(i));
    return diff;
}
package jeremy;

//import static jeremy.TargetManager.*;
import java.util.List;

public abstract class Expert {

    // Average payoff of this expert. We need to start with a
    // value > 0 so poor experts will be taken out of the pool!
    private double M;

    // number of steps represented in M. We initialize this at
    // 10 steps to give 'weight' to the initial value of M for
    // each expert.
    private double S;

    // number of phases expert's been played
    private int i = 0;

    private static final double gamma = .75;

    // **********************************************
    // Constructors
    public Expert() {
        // Note that we must initialize M with a nonzero value, or else
        // the Agent.pickAGoodExpert function will fail.
        // M = .5;
        M = 0.001;
        S = 10;
    }

    // **********************************************
    // Utility Functions

    // public abstract int expertClass();

    public int geti() {
        return i;
    }

    public double getNi() {
    }
public double getExpectedValue() {
    return M;
}

public boolean isAHistogramExpert() {
    if (this instanceof HistogramExpert) {
        return true;
    }
    return false;
}

public boolean isARecencyExpert() {
    if (this instanceof RecencyExpert) {
        return true;
    }
    return false;
}

// basic EEE algorithm:
// S = S + Cef*n[i]
// M = M + (Cef*n[i](avgR - M))/S
// i = i + 1

// ************************************************************
// Function: advise
// Purpose: takes in the current position of the agent's bots
// and the (real) targets, and outputs a "phantom target list"
// for the agent to follow.
// Notes: When this function is called, the expert does NOT update
// its values.
public abstract List<Target> advise(List<Target> realTargets, List<Bot> myBots);

// ************************************************************
// Function: correlate
// Purpose: produces a correlation value between this expert and
public abstract double correlate(Expert e);

// Function: update
// Purpose: takes in the current position of the agent's bots
// and the current (real) targets,
// and updates the expert's values for M, S, and i.
// Notes: this function is now only run at the end of each phase.
// Cef represents the correlation between this expert and the played
// expert, and it is a value between 0 and 1. alphaPow is a
// memory variable in Agent, and it holds the value of alpha^i
// where i is the phase number and alpha is also set in the
// Agent class. Gamma is the past/present weighting factor.
public void update(double avgR, double Ni, double Cef, double alphaPow)
{
    double n = Ni * Cef * alphaPow;
    S = S + n;
    // we only want to try to update M if S isn't 0
    if (S != 0)
    {
        M = gamma * (1 - n / S) * M + (1 - gamma + gamma * n / S) * avgR;
    }
}

B.7  ExpertList.java

package jeremy;

import static jeremy.TargetManager.*;

import java.util.*;

public class ExpertList implements Iterable<Expert>
{
    private List<Expert> eList;
    private Set<List<Integer>> distributions;
14 // C is the matrix of 'cheating' correlation values, which are
15 // set a priori.
16 //private Map<PairOfExperts, Double> C;
18 // ************************************************************
20 public ExpertList()
22 {
23     eList = new ArrayList<Expert>();
24     distributions = new HashSet<List<Integer>>();
25     generateDistributions(distributions, new ArrayList<Integer>(),
26     DISTRO_SIZE);
27     System.out.println("Distributions generated...");
28     // first we generate 42 recency experts:
29     generateRecencyExperts();
30     System.out.println("Recency experts generated...");
31     // then we generate all our histogram experts:
32     generateHistogramExperts();
33     System.out.println("Histogram experts generated...");
34     System.out.println(numExperts() + " experts!");
36 }
38 // ************************************************************
40 // Utility Functions
42 public Expert getRandomExpert()
44 {
45     Random rand = new Random();
46     return eList.get(rand.nextInt(numExperts()));
48 }
50 // Returns a defensive copy.
52 public List<Expert> getListOfExperts()
54 {
55     return new ArrayList<Expert>(eList);
57 }
59 public int numExperts()
61 {
62     return eList.size();
64 }
66 public Iterator<Expert> iterator()


```java
public void sort()
{
    Collections.sort(eList, ExpertComparator.expertComparator);
}

// does NOT return a defensive copy. this is used for cheating.
public Expert getHistogramExpert(List<Integer> distro)
{
    for (Expert myExpert : eList)
        if (myExpert.isAHistogramExpert())
            if (((HistogramExpert) myExpert).getDistribution().equals(distro))
                return myExpert;
    throw new IllegalArgumentException("bad distribution");
}

// this is also used only for cheating.
public Expert getRecencyExpert(List<Integer> distro)
{
    for (Expert myExpert : eList)
        if (myExpert.isARecencyExpert())
            if (((RecencyExpert) myExpert).getDistribution().equals(distro))
                return myExpert;
    throw new IllegalArgumentException("bad distribution");
}

// *****************************************
// Generator Functions
// *****************************************

// Function: generateDistributions
// Purpose: This recursive function generates distributions using a
// very specific rule. A 'distribution' consists of a list of Integers,
// and it refers to the number of bots an expert will direct
// towards its top X targets. This function returns a set of all
// monotonic distributions, i.e. distributions where there's a
// monotonically decreasing number of bots assigned to each following
// target.
public void generateDistributions(Set<List<Integer>> allDistros,
```
List<Integer> thisDistro, int max)
{
    if (thisDistro.size() == DISTRO_SIZE)
    {
        if (getSum(thisDistro) == DISTRO_SIZE)
            return;

        // At this point, we must still need to fill up the rest of this distro.
        max = Math.min(max, DISTRO_SIZE - getSum(thisDistro));
        for (int i = 0; i <= max; i++)
            { List<Integer> freshDistro = new ArrayList<Integer>(thisDistro);
                freshDistro.add(i);
                generateDistributions(allDistros, freshDistro, i);
            }
    }

    private int getSum(List<Integer> thisDistro)
    {
        int sum = 0;
        for (Integer i : thisDistro)
            sum += i;
        return sum;
    }

    // ******************************************************
    // Function: generateRecencyExperts
    // Purpose: Generates a whole bunch of recency experts. The return value is the number of experts created by this function.
    private int generateRecencyExperts()
    {
        for (List<Integer> distribution : distributions)
            eList.add(new RecencyExpert(distribution));
        return distributions.size();
    }
}
// ******************************************************************************
// Function: generateHistogramExperts
// Purpose: Generates a whole bunch of histogram experts. The return value is the number of experts created by this function.
private int generateHistogramExperts()
{
    for (List<Integer> distribution : distributions)
        eList.add(new HistogramExpert(distribution));
    return distributions.size();
}
// ******************************************************************************
// Class: ExpertComparator
// Purpose: simple comparator for experts, compares based on score.
// Notes: This class utilizes the singleton pattern.
public static class ExpertComparator implements Comparator<Expert> {
    private ExpertComparator()
    {
    }
    public int compare(Expert e, Expert f)
    {
        if (e.getExpectedValue() > f.getExpectedValue())
            return -1;
        if (e.getExpectedValue() < f.getExpectedValue())
            return 1;
        return 0;
    }
}

B.8 HistogramExpert.java

package jeremy;
import static jeremy.TargetManager.*;
import java.util.*;
import java.util.*;
public class HistogramExpert extends DistributionExpert
{
    private static class Bucket
    {
        private int x, y;
        private int width, length;
        private int numTargets;
        public Bucket(int x, int y, int length, int width)
        {
            if ((x + width > FIELDSIZE / 2) || (y + length > FIELDSIZE / 2))
                throw new IllegalArgumentException("bucket off the map!");
            this.x = x;
            this.y = y;
            this.length = length;
            this.width = width;
            numTargets = 0;
        }
        public Bucket(int x, int y, int size)
        {
            this(x, y, size, size);
        }
        public String toString()
        {
            return "B(\(x + width / 2\), \(y + length / 2\))";
        }
        public boolean containsTarget(Target myTarget)
        {
            double xTarget = myTarget.getLoc().getX();
            double yTarget = myTarget.getLoc().getY();
            if (xTarget >= x && (xTarget < x + width))
                && (yTarget >= y) && (yTarget < y + length))
                return true;
            return false;
        }
    }
}
public Target grabTarget(List<Target> realTargetsCopy)
{
    for (Target myTarget : realTargetsCopy)
    {
        if (containsTarget(myTarget))
        {
            realTargetsCopy.remove(myTarget);
            return myTarget;
        }
    }
    return defaultTarget();
}

public void incrementNumTargets()
{
    numTargets++;
}

public int getNumTargets()
{
    return numTargets;
}

public Target defaultTarget()
{
    return new Target(x + width / 2, y + length / 2, 1);
}

private static final int GRIDSQUARE = 20;

private static final int GRIDSIZE = FIELDSIZE / GRIDSQUARE;

private static final int offset = GRIDSIZE * GRIDSQUARE / 2;

private static List<Bucket> buckets;

package;
```java
static {
    int i, j;
    buckets = new ArrayList<Bucket>();
    for (i = 0; i < GRIDSIZE; i++)
    {
        for (j = 0; j < GRIDSIZE; j++)
            buckets.add(new Bucket(i * GRIDSQUARE - offset, j * GRIDSQUARE - offset, GRIDSQUARE));
    }
    // System.out.println("Buckets: " + buckets);
}

HistogramExpert(List<Integer> distribution)
{
    super(new ArrayList<Integer>(distribution));
}

// Utility Functions
@Override
public String toString()
{
    return "Histogram Expert " + super.toString();
}

// Function: updateHistogram
// Purpose: This function takes in each new target, and increments
// the bucket that this target falls into.
// Notes: This function expects to be called with every new target. It
// also assumes that the target is a legitimately initialized target.
public static void updateHistogram(Target myTarget)
{
    for (Bucket myBucket : buckets)
    {
        if (myBucket.containsTarget(myTarget))
        {
            myBucket.incrementNumTargets();
            return;
        }
    
    throw new IllegalArgumentException("target doesn't fall into any bucket!");
}
```
Function: getHighestFrequencyBuckets
// Purpose: This function returns a list of the X most highly populated
// buckets, where X must be <= DISTRO_SIZE.
// Notes: This function also has the advantage of ordering the buckets ←
// list
// according to frequency.

private static List<Bucket> getHighestFrequencyBuckets(int n) {
    if (n > DISTRO_SIZE)
        throw new IllegalArgumentException("n is too high");
    Collections.sort(buckets, BucketComparator.bucketComparator);
    // System.out.printf("best " + n + " buckets:
    // buckets.subList(0, n));
    return new ArrayList<Bucket>(buckets.subList(0, n));
}

@Override
public List<Target> advise(final List<Target> realTargets, List<Bot> ←
myBots) {
    List<Target> phantomTargets = new ArrayList<Target>();
    List<Bucket> bestBuckets = getHighestFrequencyBuckets(DISTRO_SIZE) ←
;
    int i, j;

    // we want to copy realTargets, because during the process of ←
generating
    // targets, we will be modifying copy.
    List<Target> copy = new ArrayList<Target>(realTargets);
    Target myTarget;
    for (i = 0; i < getDistribution().size(); i++)
        for (j = 0; j < getDistribution().get(i); j++)
            { ←
                // here we want to either grab the real targets ←
                // which are in
                // our chosen histogram buckets, or else we want ←
to generate
                // default targets in these buckets. NOTE: if ←
grabTarget does
                // in fact return a target which is in copy, then ←
it removes
                // that target from copy so it will not be ←
selected again!
myTarget = (bestBuckets.get(i).grabTarget(copy));
    // if (copy.contains(myTarget))
    // copy.remove(myTarget);
    phantomTargets.add(myTarget);
}

    // System.out.println("Phantom targets:" + phantomTargets);
    return phantomTargets;
}

// ******************************************************
// Class: BucketComparator
// Purpose: simple comparator for buckets, compares based on numTargets.
// Notes: This class utilizes the singleton pattern. Also, this comparator
// reverses the standard order of comparison: highest elements first.
public static class BucketComparator implements Comparator<Bucket>
{
    public static BucketComparator bucketComparator = new ->
    BucketComparator();

    private BucketComparator()
    {
    }

    public int compare(Bucket a, Bucket b)
    {
        if (a.getNumTargets() > b.getNumTargets())
            return -1;
        if (a.getNumTargets() < b.getNumTargets())
            return 1;
        return 0;
    }
}

B.9 Locatable.java

package jeremy;

public interface Locatable
{
    public Location getLoc();
}
package jeremy;

public class Location implements Locatable {

    private double x, y;

    public Location(double xgiven, double ygiven) {
        x = xgiven;
        y = ygiven;
    }

    public Location() {
        x = 0;
        y = 0;
    }

    public Location(Location loc) {
        this(loc.x, loc.y);
        // same as this = Location(loc.x, loc.y);
    }

    public Location getLoc() {
        return new Location(this);
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public void setLoc(double setx, double sety) {
        x = setx;
    }
}
public void moveLoc(double xinc, double yinc)
{
    x += xinc;
    y += yinc;
}

public String toString()
{
    return String.format("(%d, %d)", (int) x, (int) y);
}

public static Location getCentroid(Location locs[])
{
    int numLocs = 0;
    Location centroid = new Location();
    for (Location myLoc : locs)
    {
        centroid.x += myLoc.getX();
        centroid.y += myLoc.getY();
        numLocs++;
    }
    if (numLocs == 0)
        throw new IllegalArgumentException("no valid locations!");
    centroid.x /= numLocs;
    centroid.y /= numLocs;
    return centroid;
}

// ****************************************************************************
// Function: getDistance
// Purpose: returns the cartesian distance between two bots.
// Notes:
public static double getDistance(Locatable local, Locatable loca2)
{
    double x, y;
    x = Math.abs(local.getLoc().getX() - loca2.getLoc().getX());
    y = Math.abs(local.getLoc().getY() - loca2.getLoc().getY());
    return Math.sqrt(x * x + y * y);
}
// ****************************************************************************
B.11 RecencyExpert.java

```java
package jeremy;

import static jeremy.TargetManager.*;
import java.util.List;
import java.util.ArrayList;

public class RecencyExpert extends DistributionExpert {

    private static List<Target> recentTargets;

    private List<Integer> distribution;

    static {
        // This is an internally maintained list of the X most recent targets
        // that have popped up. It's ordered from back to front, i.e. the most
        // recent target is at position 0.
        recentTargets = new ArrayList<Target>();
        for (int i = 0; i < DISTRO_SIZE; i++)
        {
            recentTargets.add(i, new Target(0, 0, 1));
        }
    }
}
```
public RecencyExpert(List<Integer> distribution) {
    super(new ArrayList<Integer>(distribution));
}

// **********************************************************************************************
// Utility Functions

@Override
public String toString() {
    return "Recency Expert " + super.toString();
}

// **********************************************************************************************
// Function: updateRecencyList
// Purpose: keeps the static recency list updated for the entire class of recency experts.
// Notes: In order to function correctly, this function must be run every time a target is created. This function assumes that it is called with every newly created target.
public static void updateRecencyList(Target newTarget) {
    recentTargets.remove(recentTargets.size() - 1);
    recentTargets.add(0, newTarget);
}

@Override
public List<Target> advise(List<Target> realTargets, List<Bot> myBots) {
    List<Target> phantomTargets = new ArrayList<Target>();
    List<Target> copy = new ArrayList<Target>(realTargets);
    Target myTarget;

    int i, j;
    for (i = 0; i < getDistribution().size(); i++)
        for (j = 0; j < getDistribution().get(i); j++)
            {
                // note: makeTarget modifies the copy list!
                myTarget = makeTarget(copy, recentTargets.get(i));
                phantomTargets.add(myTarget);
                // phantomTargets.add(new Target(recentTargets.get(1) + (i)));
            }
private Target makeTarget(List<Target> realTargetsCopy, Target locus) {
    for (Target myTarget : realTargetsCopy)
        if (Location.getDistance(myTarget, locus) < 3)
            realTargetsCopy.remove(myTarget);
    return myTarget;
}

public Target(Target myTarget) {
    // ************Constructor************
```java
    this(myTarget.getLoc(), myTarget.getDeadline());
}

public Target(int x, int y, int myDeadline)
{
    this(new Location(x, y), myDeadline);
}

public Target(Location myLoc, int myDeadline)
{
    loc = myLoc;
    deadline = myDeadline;
    stagesToCompletion = PROCESSTIME;
}

// ************************************************************************
// Utility Functions

public Location getLoc()
{
    return loc;
}

public int getDeadline()
{
    return deadline;
}

public void decrementDeadline()
{
    deadline--;
    stagesToCompletion = PROCESSTIME;
}

public String toString()
{
    return loc.toString();
}

public boolean isFinished()
{
    if (stagesToCompletion < 0)
        throw new IllegalArgumentException("something is wrong with "+
                        + "this target!" + toString());
```
if (stagesToCompletion == 0)
    return true;
return false;

public void decrementStagesToCompletion()
{
    stagesToCompletion--;
}

// ************************************************************
// Function: drawTarget
// Purpose: draws the robots to the screen. Also draws the bot's
// correlation values to the chart.
// Notes: a bot that is moving will be colored green, where
// all other bots will be colored red.
public void drawTarget(Graphics g)
{
    g.setColor(Color.red);
    g.fillRect((int) (FIELDSIZE / 2 + loc.getX() + 2), (int) (FIELDSIZE / 2 - loc.getY() + 2), 4, 4);
}

// ************************************************************
// Function: anyBotHitsMe
// Purpose: Takes in a list of bots, and checks to see whether
// any of the bots have 'hit' this target. A 'hit' means that
// some bot is within 2 spaces of this target.
// Notes:
public boolean anyBotHitsMe(List<Bot> botList)
{
    for (Bot myBot : botList)
    {
        if (Location.getDistance(this, myBot) <= HITDISTANCE)
            return true;
    }
    return false;
}
package jeremy;

import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Random;
import java.util.List;

public class TargetManager {
    // the length of the standard distribution
    public static int DISTRO_SIZE;

    // the number of bots in play. Should be a multiple of 4, for
    // the purposes of the cheating experts.
    public static int NUMBOTS;

    // number of gridpoints in each row of our goal-point grid.
    // in this case, we avoid the boundaries.
    public static final int FIELDSIZE = 300;

    public static final int TIME_DELAY = 5;

    // probability that a target will be generated at each stage
    public static double TARGENPROB = .1;

    // distance at which a bot can hit a target
    public static final double HITDISTANCE = 2;

    // minimum deadline for the targets.
    public static final int MIN_DEADLINE = 50;

    // determines which opponent we will use.
    public static int OPPONENT = 1;

    // determines whether we cheat for the best expert or not.
    public static boolean CHEATING = false;

    // determines whether we cheat for the best expert or not.
    public static boolean CORRELATION = false;

    // determines whether we make the sim run 'nice' for display
    public static boolean DISPLAY = false;
}
// determines whether we're running an expert test.
public static boolean EXPERTTEST = false;

// determines which expert we're testing.
public static int EXPERTNUM = 0;

public static void main(String[] args)
{
    // arg[0] specifies the size of the distribution. Must be at least 4,
    // for the purposes of cheating.
    // arg[2] specifies the style of simulation (cheating, correlation, no
    // correlation, or specific expert test runs.)
    // arg[4] specifies whether or not we display the simulation.

    DISTRO_SIZE = Integer.parseInt(args[0]);
    NUMBOTS = DISTRO_SIZE;
    OPPONENT = Integer.parseInt(args[1]);
    if (Integer.parseInt(args[2]) == 1)
    {
        System.out.println("cheating...");
        CHEATING = true;
    }
    else if (Integer.parseInt(args[2]) == 2)
    {
        System.out.println("correlation...");
        CORRELATION = true;
    }
    else if (Integer.parseInt(args[2]) == 3)
    {
        System.out.println("no correlation...");
    }
    else if (Integer.parseInt(args[2]) == 4)
    {
        System.out.println("expert test...");
        EXPERTTEST = true;
        EXPERTNUM = Integer.parseInt(args[3]) - 1;
    }
    if (Integer.parseInt(args[4]) == 1)
    {
        DISPLAY = true;
        playGameForNStages(50000, args);
        //playGameForNStages(20000, args);
    }
}
private static void playGameForNStages(int n, String[] args) {
    Agent myAgent = new Agent();
    List<Target> realTargets = new ArrayList<Target>();
    if (DISPLAY)
        DisplaySwing.showDisplay(myAgent.getBotList(), realTargets);
    Random rand = new Random();
    double M = 0;

    // String filename = new String("D:\CEA\CEA_" + DISTRO_SIZE + "_op"
    //   + OPPONENT + "_sim" + args[2] + "_itr" + args
    // [3] + ".m");
    String filename = new String("D:\CEA\CEA_dryrun_itr" + args
    // [3] + ".m");
    if (EXPERTTEST)
        filename = new String("D:\CEA\ET_" + DISTRO_SIZE + "_op"
        + OPPONENT + "_itr" + args[3] + ".m");
    System.out.println(filename);
    PrintWriter matlabWriter = null;
    try {
        matlabWriter = new PrintWriter(filename);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    StringBuffer h10 = new StringBuffer();
    String varName = new String("M");
    if (!EXPERTTEST)
        h10.append(varName + " = [");
    else
        h10.append(varName + " = ");

    for (int i = 0; i < n; i++) {
        if (rand.nextDouble() < TARGENPROB)
synchronized (realTargets) {
    generateTarget(realTargets, i, OPPONENT);
}
myAgent.play(realTargets);
synchronized (realTargets) {
    M = scoreAndCleanTargets(realTargets, myAgent);
}
if (!EXPERTTEST)
    h10.append(M + "");
else
    h10.append(";");
System.out.println("final avg score: " + M);
System.out.println("Best Expert: " + myAgent.getNBBestExperts(1));
System.out.println("Most Played Expert: " + myAgent.getMostPlayedExpert());
System.out.println("Played for " + myAgent.getMostPlayedExpert().geti() + " phases.");
System.out.println("Done!");
matlabWriter.println(h10);
//matlabWriter.println(String.format(" plot(%s)\n", varName));
//matlabWriter.println("hold on;");
if (matlabWriter != null)
    matlabWriter.close();
}

// ***************************************************************
// Function: scoreAndCleanTargets
// Purpose: Takes in the current lists of targets and of bots, and checks to // see how many bots 'hit' targets.
// Notes:

private static double scoreAndCleanTargets(List<Target> realTargets, Agent myAgent)
{
    int bullseyes = 0;
}
for (Target myTarget : new ArrayList<Target>(realTargets))
{
    // first, we check to see if this target is in the process of
    // getting completed.
    if (myTarget.anyBotHitsMe(myAgent.getBotList()))
    {
        myTarget.decrementStagesToCompletion();
        if (myTarget.isFinished())
        {
            // System.out.println("Target hit!");
            realTargets.remove(myTarget);
            bullseyes++;
        }
    } else
    {
        myTarget.decrementDeadline();
        if (myTarget.getDeadline() == 0)
        {
            // System.out.println("Target missed!");
            realTargets.remove(myTarget);
        }
    }
}

double score = bullseyes * 1 / TARGENPROB;
return myAgent.updateScore(score);

// ************************************************************
// Function: generateTarget
// Purpose: Creates a new target and adds it to the targetList.
// Notes: Right now the new the target is at a random location, but later
// the location will be based on some probabilistic distribution. The
// deadline is also random, from 20 to 70 turns.
// Note also that (MAYBE)
// this function exits without action if targetlist already contains
// the max number of targets.

private static void generateTarget(List<Target> tList, int stageNum,
int rule)
{
    // if (tList.size() >= MAXTARGETS)
    // return;
    Random rand = new Random();

Target myTarget = null;
switch (rule)
{
    case 0:
        Location randomLoc = new Location(rand.nextInt(FIELDSIZE) - FIELDSIZE / 2, rand.nextInt(FIELDSIZE) - FIELDSIZE / 2);
        myTarget = new Target(randomLoc, rand.nextInt(500) + MINDEADLINE);
        break;
    case 1:
        TARGENPROB = .01 * NUMBOTS;
        double i = Math.signum(rand.nextDouble() - .5);
        double j = Math.signum(rand.nextDouble() - .5);
        Location myLoc = new Location(100 * i - 10 + 20 * rand.nextDouble(), 100 * j - 10 + 20 * rand.nextDouble());
        myTarget = new Target(myLoc, rand.nextInt(100) + MINDEADLINE);
        break;
    case 2:
        TARGENPROB = .02 * NUMBOTS;
        myTarget = new Target(Loop.loop.getNextPos), rand.nextInt(200) + MINDEADLINE);
        break;
    case 3:
        TARGENPROB = .005 * NUMBOTS;
        // generateTarget(tList, stageNum/8, 3);
        i = Math.signum(rand.nextDouble() - .5);
        j = Math.signum(rand.nextDouble() - .5);
        if (stageNum % 8000 < 2000)
            myLoc = new Location(100 - 10 + 20 * rand.nextDouble(), 100 - 10 + 20 * rand.nextDouble());
        else if (stageNum % 8000 < 4000)
            myLoc = new Location(100 - 10 + 20 * rand.nextDouble(), -100 - 10 + 20 * rand.nextDouble());
        else if (stageNum % 8000 < 6000)
            myLoc = new Location(-100 - 10 + 20 * rand.nextDouble(), -100)
250 \[ \text{else} \]
\[ \text{myLoc = new Location(-100 - 10 + 20 \times \text{rand.nextDouble()},} \]
\[ \quad 100 - 10 + 20 \times \text{rand.nextDouble()});} \]
\[ \text{myTarget = new Target(myLoc, rand.nextInt(200) + \rightarrow} \]
\[ \quad \text{MINDEADLINE);} \]
\[ \text{break;} \]
\[ \text{case 4:} \]
\[ \quad \text{TARGENPROB = .005 \times NUMBOTS;} \]
\[ \quad i = \text{rand.nextDouble() - .5;} \]
\[ \quad \text{myLoc = new Location(260 + i, (stageNum \% 2000) / 8 - 125);} \]
\[ \quad \text{myTarget = new Target(myLoc, rand.nextInt(200) + \rightarrow} \]
\[ \quad \text{MINDEADLINE);} \]
\[ \text{break;} \]
\[ \text{case 5:} \]
\[ \quad \text{TARGENPROB = .005 \times NUMBOTS;} \]
\[ \quad i = \text{Math.signum(rand.nextDouble() - .5);} \]
\[ \quad \text{myLoc = new Location(125 + i, (stageNum \% 2000) / 8 - 125);} \]
\[ \quad \text{myTarget = new Target(myLoc, rand.nextInt(200) + \rightarrow} \]
\[ \quad \text{MINDEADLINE);} \]
\[ \text{break;} \]
\[ \text{case 6:} \]
\[ \quad \text{TARGENPROB = .02 \times NUMBOTS;} \]
\[ \quad i = \text{rand.nextDouble();} \]
\[ \quad \text{myLoc = new Location(0, 0);} \]
\[ \quad \text{for (double k = 0; k <= DISTRO_SIZE; k++)} \}
\[ \quad \quad \text{if (i < k / DISTRO_SIZE)} \}
\[ \quad \quad \quad \text{if (k <= DISTRO_SIZE / 4)} \]
\[ \quad \quad \quad \quad \text{myLoc = new Location(40 + k \rightarrow} \]
\[ \quad \quad \quad \quad \quad \quad \text{135, -125);} \]
\[ \quad \quad \quad \text{else if (k <= DISTRO_SIZE / 2)} \]
\[ \quad \quad \quad \quad \{
\quad \quad \quad \quad \quad \quad \text{k = DISTRO_SIZE / 4;} \]
\[ \quad \quad \quad \quad \quad \quad \text{myLoc = new Location(40 + k \rightarrow} \]
\[ \quad \quad \quad \quad \quad \quad \quad \quad \text{135, -50);} \]
\[ \quad \quad \quad \} \text{else if (k <= 3 \times DISTRO_SIZE / 4)} \]
\[ \quad \quad \quad \{
\quad \quad \quad \quad \quad \quad \text{k = DISTRO_SIZE / 2;} \]
public static Loop loop = new Loop();

private mt nextPos = 0;

private Loop()
{
    double radians = 0, xpos, ypos;
    for (int k = 0; k < cycleLocs.length; k++)
    {
        radians += 2 * Math.PI / cycleLocs.length;
        xpos = (int) Math.round(125 * Math.cos(radians));
        ypos = (int) Math.round(125 * Math.sin(radians));
        cycleLocs[k] = new Location(xpos, ypos);
    }
}

public Location getNextPos()
{
    myLoc = new Location(40 * k - 135, 50);
}

else
{
k -= 3 * DISTRO_SIZE / 4;
myLoc = new Location(40 * k - 135, 125);
}
break;
}

myTarget = new Target(myLoc, rand.nextInt(200) + MINDEADLINE);

// System.out.println("new target location is "+
// myTarget.getLoc().getX()
// + "," + myTarget.getLoc().getY() + ")");
tList.add(myTarget);
RecencyExpert.updateRecencyList(myTarget);
HistogramExpert.updateHistogram(myTarget);

private static class Loop
{

private Location[] cycleLocs = new Location[DISTRO_SIZE];

public static Loop loop = new Loop();

private int nextPos = 0;

private Loop()
{
    double radians = 0, xpos, ypos;
    for (int k = 0; k < cycleLocs.length; k++)
    {
        radians += 2 * Math.PI / cycleLocs.length;
        xpos = (int) Math.round(125 * Math.cos(radians));
        ypos = (int) Math.round(125 * Math.sin(radians));
        cycleLocs[k] = new Location(xpos, ypos);
    }
}

public Location getNextPos()
{
nextPos++;  
if (nextPos >= cycleLocs.length)  
    nextPos = 0;
return cycleLocs[nextPos];
Bibliography


