Graph Learning: A Three Step Approach to On Board Learning

Patrick McDowell PM@SELU.EDU
Theresa Beaubouef TBEAUBOUEF@SELU.EDU
Southeastern Louisiana University
Computer Science Department, SLU 10847
Hammond, LA 70402 USA

Brian Bourgeois BSB@NRLSSC.NAVY.MIL
Naval Research Laboratory
Stennis Space Center, MS 39529

Don Sofge DON.SOFGE@NRL.NAVY.MIL
Navy Center for Applied Research in AI
Washington DC 20375

S.S Iyengar IYENGAR@BIT.CSC.LSU.EDU
Jianhua Chen JIANHUA@CSC.LSU.EDU
Computer Science Department
Louisiana State University
Baton Rouge LA 70803-4020

Abstract

This research develops a methodology for controlling robots that will enable them to function without an abundance of a-priori knowledge embedded in either databases, maps, models, or rule systems. With only a description of the goal state in terms of sensor/action values, it allows the system to avoid undue physical trials of solutions, by learning from previously experienced situations, and by using techniques such as sensor mirroring. The three step process presented in this work involves the collection of a recent memory, organization of this information into a directed graph where nodes represent situations encountered by the robot, and the use of a genetic algorithm that generates a feed forward neural network for robot control.

1. Introduction

Robot control systems present their designers with several challenges. In known static environments procedures can be preprogrammed into the robot's control software for each different situation that is to be encountered. In an ideal world, as long as the robot's control software can select the correct procedure for a given circumstance its operation will continue without error, assuming that the robot's sensors return consistent and reliable information to the decision making processes. But more often it is the case that the environment does change, or sensor performance degrades with time. If the designers are always close by, they can quickly tune the robot's software to handle the changes, but in cases where the robot must operate without human assistance for extended durations other approaches must be taken. The end-target application for this research, formation maneuvering with Unmanned Underwater Vehicles (UUV) guided by acoustic sensors, for use in search and survey operations is just such a case. On board learning is an attractive option for this application because it allows the robot to calibrate itself for its local operating conditions, allowing the robot some degree of independence and removing the burden from the designer to anticipate every possible operating scenario. This

paper presents a memory-based algorithm that relies on memories of recent events to allow goal directed behaviors to be generated or modified in response to the environment.

Learning onboard a robot presents it own set of challenges. Each exploratory motion that the robot executes, whether it is to learn more about the environment, or to test a solution, takes time and causes wear and tear on mechanical parts. The system must be able to observe its progress in order to detect when goals are not being met. The data available to the learning system must contain sufficient information to form a complete strategy for the situation at hand. And the learning system must be able to generate a new behavior strategy before the environment changes enough to make it irrelevant.

Anytime Learning (Parker, 2002) addresses these problems by pairing a high fidelity simulation with the robot's execution system. In this system the robot keeps the simulation up to date with its sensors and the simulation improves the robot's controllers by exhaustively testing various solutions in its simulated environment. An alternative approach called Q Learning, reviewed in Mitchell (1997), allows a robot or agent to operate in its environment without a-priori knowledge. Through the use of a random exploration and the concept of delayed reward the algorithm builds a table that relates optimal actions to sensor inputs. The research described in this paper combines the functional strengths of these two methods. The result is a process designed to produce situation dependent control algorithms for a robot in near real time without requiring that the system physically test all possible solutions. The main criteria for this system was that it should be able to function without an abundance of a-priori knowledge embedded in either databases, maps, models, or rule systems. The process has three steps and is based on the use of recent memories of situations encountered by the learning agent. The steps are outlined below.

1. Focused exploration of environment.
2. Topological organization of knowledge gained in exploration into a directed graph.
3. Learning and solution space reduction based on primitive rules applied to the directed graph.

The primary focus of this paper is the graph learning algorithm. The next section provides background information on reinforcement learning and related techniques. Section 3 provides a conceptual overview of the ideas and concepts of the graph learning algorithm and the data structures that it relies upon, and the methods of graph creation and learning from it are described in more detail. Section 4 provides a case study in which the algorithm was used in a simple leader/follower simulation. While the focused exploration of the environment is key to the algorithm, it is a subject unto itself, so in this work, we provide of only brief description of exploration method used in the case study.

2. Background

The graph learning algorithm is the fifth algorithm in a series of algorithms that were developed to automate the learning process for operation onboard robots in unstructured environments (McDowell, 2005). Its predecessors included a "human in loop" supervised method, two methods that used supervisory programs based on rules of thumb to filter positive examples from the recent memory, and one unsupervised technique that relied on sensor prediction to determine the optimal action for each situation. As a learning method, the graph learning algorithm is a step up from the other methods because it does not rely on rules of thumb applied at each time step to create training examples, thus lessening the reliance on a-priori

knowledge. It is better than the sensor prediction based methods because it does not estimate results to rank solutions; it uses what actually worked and learns from that. While somewhat similar in function to Q Learning, a key merit of the graph method is that it creates a graph structure that can provide functionality in addition to learning, such as solution space reduction. These ideas are discussed in the next section.

Q Learning and its many variants fall into a class of algorithms called Reinforcement Learning. In general Reinforcement Learning algorithms provide an agent with the ability to reach its goal by providing it with a means to select optimal actions. In Q Learning, the agent is assumed to be able to acquire information about its environment, i.e. sense its state, and its actions are assumed to change its state in a measurable way. The use of delayed reward is key to Q learning. Delayed reward is used because the supervisor or trainer only knows when the agent achieves success, which is defined by reaching a predefined goal state. The idea behind delayed reward is that the total available reward for reaching the goal state is discounted by the number of steps that it takes to reach the goal state.

The discounted rewards for each state and each action are calculated using the Q function. The Q function is defined as the sum of the immediate reward available at the current state plus the discounted reward. Its equation is shown below.

```
s  - current state
s' - the state in the set of states adjacent to state s
     that contains a'.  (s', a') is the optimal
     state/action combination adjacent to s.
a  - current action
a' - action that returns the greatest reward at state s'
d  - amount that rewards are discounted, range is a real
     number between 0 to 1
r  - immediate reward available for current state.

Q (s, a) = r + d*max Q(s', a')
```

The **max** function in the equation indicates that the state, action pair (s′, a′), adjacent to (s, a), that delivers the maximum reward is selected. Of particular note is that the optimization function in Q Learning is similar to the shortest path calculation used in Floyd's algorithm (1962), but since its output is stored in a table it does not retain the temporal information collected by its exploration function.

A good example of Q Learning is the program that Tesauro (1995) created called TD-Gammon that plays backgammon at the world class level. It uses Q Learning to modulate error assignment parameters in order to train a multiple perceptron neural network. It learns to play by playing games against itself. Initially the moves are completely random, so the first games take several times longer than the usual 50 to 60 moves required by human players. As it learns, the length of the games is reduced to human like levels. In the latest version, it played 1.5 million games in order to reach the world champion level. The program has been entered in backgammon tournaments and has been competitive at the world champion level. While it uses a learning strategy to train neural networks as does the graph learning algorithm presented in this paper, its input space is constricted to a discrete non changing environment, and since it does not operate physical machinery, there is no attempt to limit physical trials.

In summary, the main strength of Q Learning is that it allows a robot or agent to operate in its environment without a-priori knowledge. It does this by learning through the use of random exploration and delayed reward. The results are stored in a table that maps state, action pairs to reward values. The entry in the table for state s that has the highest reward contains the action

that moves the agent to its goal state quickest. The work described in this paper differs because the exploration of the environment is not random, it is goal driven. Additionally, by using a directed graph instead of the Q function and its associated table, spatial and temporal information associated with the situations that the agent encounters is preserved. This extra information can be used to facilitate other functions, such as Observer (graph) control, next state prediction, and solution space reduction, all of which are described in subsequent sections.

The next paragraphs describe an onboard learning method that has had success in many tasks, including team box pushing, and determining walking gaits. A key feature of this method, Anytime Learning (Grefenstette, Schultz 1994), is that it teams a simulation with an execution system.

The basic idea behind Anytime Learning is that the execution system and the learning system run together continuously in a complementary manner. The learning system contains a simulation of the environment in which the execution module operates. Its job is to continuously test strategies in the simulation so it can identify and develop improved strategies that it can then provide the execution module. Meanwhile, the execution module is operating in the environment collecting relevant parameters about performance of the strategies and any changes in the operating environment. This information is used to improve the fidelity of the simulation. Figure 1 below shows a conceptual diagram depicting the major components of Anytime Learning and their relationship to one another.

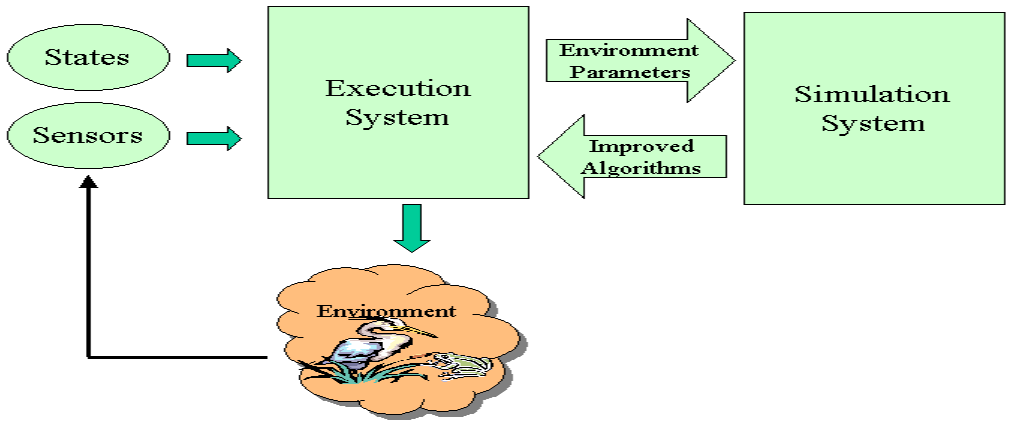

Figure 1. The basic components and their relationship to one another in a typical Anytime Learning System. The execution system accepts inputs from states and sensors, makes decisions and controls actuators, which in turn influences the environment. Meanwhile it sends environmental parameters to the simulation system, which uses them to improve the fidelity of the simulation. The simulator takes advantage of the computers speed and finds improved control strategies, which it in turn provides to the execution system.

Parker used a variant of Anytime Learning, punctuated Anytime Learning, to develop gaits for hexapod robots and to evolve team behavior in box pushing tasks (Parker, 2002). The term *punctuated* means that the learning system is not in the robot with the execution system. Since it is not close at hand, it does not have immediate access to sensor data, so the simulation in the learning system is not continually kept in sync with the environment. Punctuated Anytime Learning is a modification that compensates for the lack of feedback, precise sensors, and high computational power in the robot.

Fitness Biasing and Co-Evolution are the methods that Parker uses to update the offline simulator that learns using a genetic algorithm (GA). In Fitness Biasing, after n generations are run in the simulator, the entire population of solutions is tested in the real environment. The results of the environment are used to bias the simulator so that its results are more in tune with those of the real environment. In Co-Evolution, the simulator itself is evolved along with the solution. Its parameters are encoded in a GA and the fitness is based on how close the results and the real results match the physical situation. While it has been shown that Anytime Learning and its variants are effective in saving the robot time in testing trial solutions, because it relies on a simulator, thus requiring a measure of a-priori knowledge, it is not as well suited for the unstructured environments for which the graph learning algorithm was designed.

The above discussion concentrates on learning methods and the supporting architectures. The next paragraphs details research done in which the application was similar to the target application, that is each of these works used acoustic sensors and a learning system to guide the action of robots.

In a pair of studies by Shah (2003), and Kushleyev, and Vohra (2004) system that were loosely based on the human ability to localize a sound using two ears were developed. The direction of the sound source was determined by examining both the amplitude and phase differences of the sound arriving simultaneously at two microphones. Their algorithm determines the direction of the sound by comparing amplitude and phase differences to a table of theoretically generated curves in order to determine the sound direction. They address the problems of range determination along with correct hemisphere detection (the process of determining if the sound is in front of or behind the detection equipment) by using snapshots of data taken at separate time intervals. Using these techniques they developed a rule based controller that worked well in robot following tasks.

Reeve and Webb (2003) used mobile robots to investigate cricket phonotaxis (the ability of female crickets to find male crickets by localizing on their song) in order to better understand the physiology and neurobiological processing in the cricket brain. They modeled the crickets using mobile robots with similar speed and hearing characteristics as crickets. To emulate the crickets neurological functions they used neural oscillators (a type of recurrent neural network whose key feature is excitory and inhibitory nodes) to process the sound and control the robots motion. The weights for the networks were found by solving for them using a closed form solution and then tuning the system for the lab conditions. Experiments were done incorporating the effects of various lighting conditions and oscillatory motions. Two different brain models were used, a less sophisticated, less physiologically faithful model, and a more complex, physiologically accurate model. In both cases the robots were able to seek to a sound source playing cricket songs. The more complex brain was able to more accurately emulate real cricket behavior.

Rucci, Wray, and Edelman (2000) built a robotic barn owl in order to study the neurobiological structures responsible for localization behavior in owls of auditory and visual targets. This work is mentioned because while their goal was different, they used similar acoustic hardware, and a similar approach in that they trained neural networks to control their robot. Their goal was twofold; first they wanted to gain a greater understanding of how the various nervous system structures of a real owl worked, and second, they wanted to provide an alternative to current calibration techniques of equipment by developing a system that could operate in different sensor modalities. The neural network in the robotic owl was trained using a reinforcement learning algorithm with the weights being updated using a modified version of Hebbian learning, reviewed in Hagan, Bemuth, Beale (1996). They report good results after training for a few hours during which about 15,000 targets are presented to the system. Their system showed adaptability in the face of changing environments, but unlike the graph algorithm, there is no emphasis on reduction of physical motions of the robot.

3. Graph Learning Algorithm

The graph learning algorithm is a method of creating a controller whose purpose is to achieve a specific goal. The primary data structure used by the algorithm is a directed graph. It is built and updated from memories recorded as the robot explores and operates in the environment, thus the algorithm can adapt to changing conditions and does not have to rely on a-priori knowledge. Within the nodes and connections of the graph, the situations that the robot experienced, in terms of sensor /action pairs are stored and arranged in topological manner, which preserves the temporal sequencing information. Once the graph is constructed, it can be used to train a neural network to accomplish a goal, or by traversing the graph in real time, control can be exercised directly. In this section an overview of how the robot views/stores its own actions and surroundings in a recent memory is provided. Next the subject of graph construction is detailed, followed by a description of the graph, its components, and how it can be used for direct control. Finally, the creation of neural network behavior controllers from the graph will be discussed.

3.1 Recent Memories

The directed graph and the algorithms that it supports rely on the concept that the actions that the robot takes in its environment can be broken down into a series of situations. If the robot were to record its actions and the environmental circumstances in which they were taken, they would form a recent memory that could be viewed as a sequence of situations that the robot encountered. The word "situation" is used because it implies that for some finite time period the robot is doing an activity while immersed in a certain set environmental circumstances. The robot's activity is defined as one or more of its actuator functions, while the environmental circumstances are defined as the robot's sensor inputs and/or its internal state variables.

In this work the recent memory is recorded during an exploratory period in which the robot tests the results of different actions in its environment. Many methods for exploration exist; in a later section, the case study details the "random but purposeful" method.

In summary, at each time slice, the robot records a sensor/action pair. A recent memory is composed of a sequence of sensor action pairs, and a situation is denoted by a contiguous stream of similar sensor/action pairs. The organization of the robot's memory into a sequence of situations is key to the operation of the graph learning algorithm. Below is a more precise description of memory, given in the terms introduced above.

SA : sensor/action pair. A data structure in which the current sensor readings and/or internal states are paired with the actuator values.

M : recent memory consisting of sensor/action pairs SA[0] …. SA[n-1]

$M = SA[0] \ldots SA[n-1];$

S : situation, consisting of a contiguous sequence of sensor/action pairs from memory M such that

$S[i] = SA[k], SA[k+1], \ldots, SA[k+(\text{situation length} - 1)],$
$k >= 0 \text{ and } k < ((n-1) - \text{situation length});$

C: Set of all situations {S[0], S[1], …S[m]} found in M;

M_s : Recent memory viewed as a sequence of situations

$M_s = S[c] + S[a] + \ldots S[t];$

The following relationships for a recent memory of length n can be made:

$M = SA[0] \ldots SA[n-1];$
$M_s = S[c] + S[a] + \ldots S[t];$

$M_s \approx M;$

Note that M_s approximates M because the set of situations, C, is finite in length. If the length of C is small and M has a large degree of variation, the M_s approximation will be imprecise. Likewise, if the length of C is adequate for the amount of variation in M, the M_s approximation will be accurate.

The next section builds on the definitions in this section in order to describe the directed graph, its components, how it is generated, and how it can be used for direct control of the robot.

3.2 Graph Generation

Once a recent memory exists, it is used to generate the directed graph. As discussed in the previous section, there are two ways of viewing the recent memory, the first being a sequence of sensor/action pairs, and the next being a sequence of situations. The key to creating the directed graph is to find the situations within the recent memory. In this work, a radial basis function (RBF), reviewed in Mitchell (1997), clustering algorithm was used to cluster the memory. In an ideal sense, the RBF clusters are analogous to the situations. By regulating the inter-cluster distance and cluster size, the accuracy of the approximation M_s can be tuned. A large cluster distance results in less accuracy, while a cluster size that is too small results in situation lengths that are too short to be meaningful. An accuracy check is made by compressing the original recent memory, M, with the new set of clusters, C, referred to as the codebook, forming M_s. A high correlation between M_s and M combined with meaningful cluster sizes indicates that the clusters are accurate and of useful size to represent the situations in the memory.

Once the memory has been broken down into a group of clusters, M_s is used to find the connectivity of the situations that the clusters represent. For example, if the codebook, C, contained 5 clusters, each with a cluster size of 10, and the recent memory, M, had 100 SA pairs, the resulting list would have 10 cluster indices in it. Below is an example list of cluster indices:

$M = \{SA[0], SA[1], S[2] \ldots SA[98], SA[99]\}$
$C = \{S[0], S[1], S[2], S[3], S[4]\}$

For this example, let M_s have the following value:

$M_s = \{S[0] , S[1], S[2], S[4], S[0], S[1], S[4], S[3], S[0]\}$

Using this technique and relying on the one dimensional aspects of time, questions like "Which situation usually follows the situation that I am interested in?" can be answered by tracing the sequence of clusters that are used to compress the recent memory. From the example above, it can be seen that the situation represented by cluster 1 follows the situation represented by cluster 0 twice in the memory.

Using the sequence of clusters produced by the compression, an adjacency matrix can be formed. The adjacency matrix is the data structure that indicates which nodes in a graph are connected to each other. Table 1 below shows the adjacency matrix for the example above. Figure 2 shows an illustration of the directed graph based on the adjacency matrix.

Cluster number	0	1	2	3	4
0	0	1	0	0	0
1	0	0	1	0	1
2	0	0	0	0	1
3	1	0	0	0	0
4	0	0	0	1	0

Table 1. An adjacency matrix for the Cluster_indices example. By looking at the node numbers in the leftmost column it can be seen that cluster 1 follows cluster 0 in the example, as indicated by the 1 in column 1, row 0 of the table.

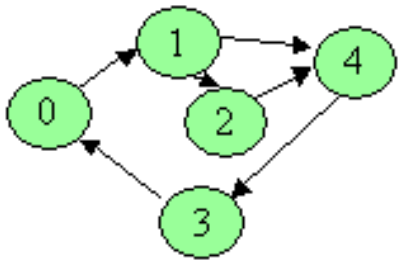

Figure 2. The graph representation of the adjacency matrix in Table 1.

Once a graph formed from recent memories is produced, it can be used to directly control the robot to achieve a specified goal.

3.3 Controlling the Robot with the Directed Graph

Controlling a robot with the directed graph relies on the assumption that the information used to create the directed graph is complete enough to form a usable strategy for the intended environment. Given that the information is adequate and the graph has been formed, control is a matter of finding the robot's current state in graph and traversing the graph to a pre-selected goal state. To accomplish this, the robot's current state, defined by its current sensor/action pair, can be compared to the graph nodes. Once the closest match is found, the shortest path between that node and the goal state can be followed. The shortest path can be found using a standard graph algorithm such as Dijkstra's shortest path algorithm (1959). By iteratively comparing the robot's state to the graph, new actuator commands can be issued when the robot transitions to each successive node on the path to the goal node.

Consider an example in which a "cat" robot uses a microphone set as a pair of ears to track a "mouse" robot that squeaks at a predefined frequency. The cat's goal is to get as close to the mouse as possible. In terms of sensor values, it wants to maximize and equalize the noise intensity of mouse squeaks in its two ears. Figure 3 shows the graph that is used as the cat's controller graph. The goal node is shown in yellow because it has the highest intensity and it is equal in both microphones, and no directional changes are needed to achieve this state. Each

node in the graph has a state number, a left relative microphone value, L, a sound intensity, I, a right relative microphone value R, and an action, A, associated with it. Actions include moving left (L), right (R) or straight (S).

If the cat finds its current state to be closest to the node marked "State = 3", it can use Dijkstra's shortest path algorithm to determine that it should do the actions on path {3, 2, 1, 0} to most quickly reach its goal of being next to the mouse. If instead of heading right at state 3, it goes left, it will most likely take the long path of {3, 4, 5, 6, 10, 1, 0} to reach its goal, a less efficient choice in this case.

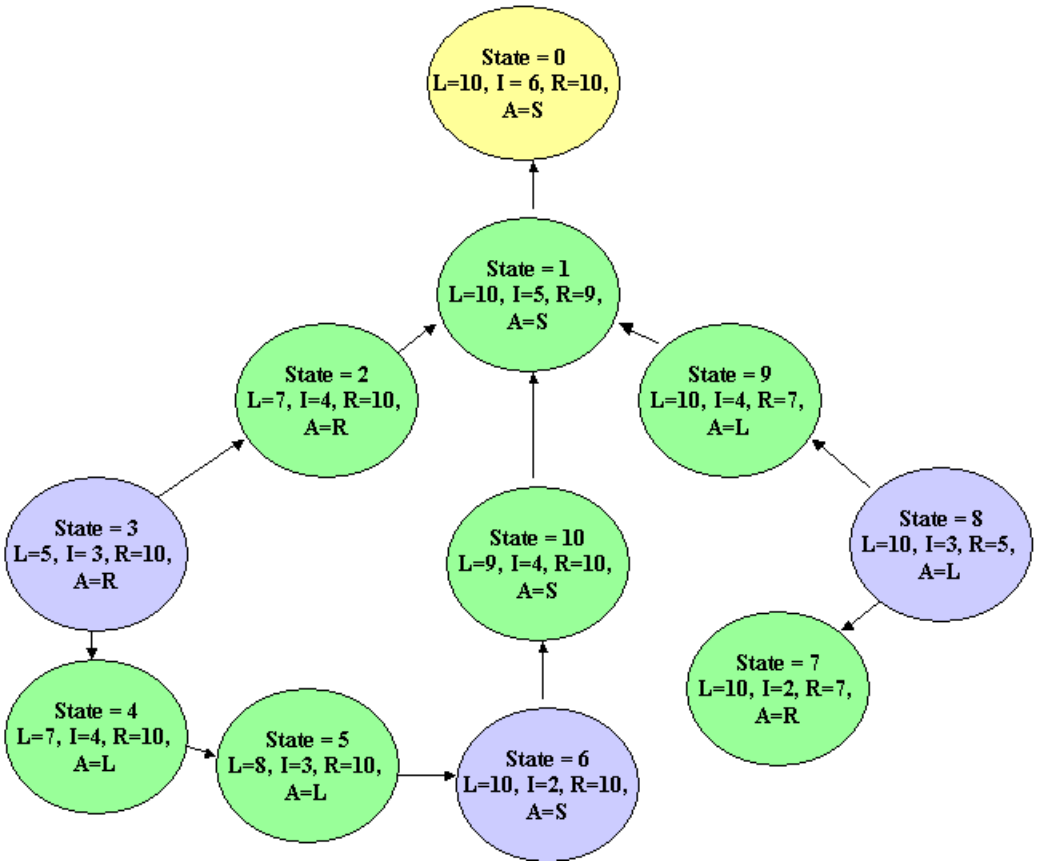

Figure 3. A directed graph of clusters from a recent memory. In each graph node there is a state number, which can be used as an index of an adjacency matrix, a left, L, and right, R, relative microphone parameters, and an intensity, I, parameter which indicates the intensity of the source. The yellow node is the goal state G, because it has the highest intensity and it is equal in both microphones, and no directional changes are needed to achieve this state. The purple states represent starting states of example paths.

Control using the graph has the advantage that alternate paths to the goal state can be found. Also, by closely monitoring states and their locations in the graph, future sensor values can be anticipated, and progress towards goal states can be tracked. In this work the graph is used to create a feed forward neural network (FFNN) controller. The FFNN has the ability to interpolate between nodes of the graph, eliminating the need to constantly search the graph for the closest matching node. So, while the graph itself is not usually used for direct control of the robot in this work, it provides the basis for an observer module that would work closely with a FFNN based control module in a robot architecture.

In regard to using FFNNs as controllers, it is a more straightforward process to develop a program that produces usable neural networks than to create programs that write programs using symbolic methods (coding of algorithms in a language such as C or LISP). While it is possible to write programs that write other programs, the process involves making sure that the machine produced programs are syntactically correct and that they run in an environment in which they can be contained. FFNNs were selected because these problems can be avoided, and their hidden layers afford them more flexibility than other neural networks (Mitchell, 1997) such as ADELINES or Perceptrons. This increased flexibility gives them the ability to represent boolean functions, continuous functions, and arbitrary functions. The next section details these methods.

3.4 Training Set Generation Using the Graph

As in the cat and mouse example in the previous section, learning using the graph relies on finding the most direct path from less desirable situations to more stable and desirable situations. The paths can then be traced and the progression of states can be used to train a FFNN. Clusters that lie on the paths from the lesser states to the goal state are used to filter raw data from the recent memory data to form a training set. By belonging to a cluster that is on a path to success, data is declared worthy of being used to train a FFNN, and is added to the training set. The training set is comprised of sequences of data from the recent memory in which the action is consistent and the data falls within the group of "filter" clusters identified earlier. A completed training set is then used in conjunction with the genetic algorithm (GA) as in (McDowell et al. 2002) to create a FFNN in a process. The entire process, starting with recent memory collection, is summarized below.

1. A recent memory is collected.
2. The recent memory is clustered resulting in a codebook, C. The codebook elements are the individual clusters. The clusters are analogous to situations.
3. The recent memory is then compressed using the codebook so that the progression of situations in the recent memory can be traced. The progression of situations is stored as a list of codebook indices.
4. From the list of indices produced by the compression, an adjacency matrix is formed. The adjacency matrix is a data structure that describes a directed graph. This graph describes which situations follow each other.
5. A cluster from the codebook is selected to represent the goal state **G**. In the robot following task this state is one in which the robot's sensors are approximately equal, and the intensity is near the mean of the range of intensities experienced while the robot is not changing its course.
6. Graph algorithms are used to find the shortest paths from all clusters whose energy is less than that of **G** to **G**.
7. The clusters used in the shortest paths are put into a set of desirable clusters.
8. Raw data from the recent memory that is in the set of desirable clusters is used to create a training set.
9. The training set is used along with the GA to create a FFNN as described in earlier.

The first four items are described in the previous sections. Here we provide more detail on items 5 through 9.

The selection of **G** is crucial to the success of this strategy. If **G** is not consistent with goals then the wrong functionality will be learned. Also **G** must be obtainable and well represented in the recent memory. For a robot that is learning to follow a lead robot using acoustic sensors, the best state **G** would be one in which the robot is not turning, and its left and right acoustic sensors

Cluster number	0	1	2	3	4
0	0	1	2	3	2
1	3	0	1	2	1
2	3	4	0	2	1
3	1	2	3	0	3
4	2	3	4	1	0

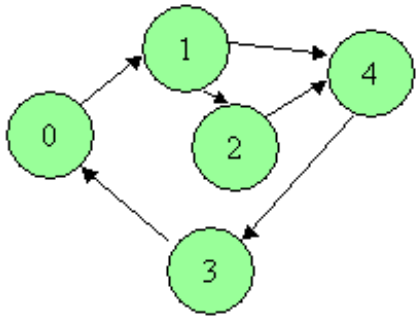

Figure 4. The distances of the shortest paths from every node to every other node using Floyd's algorithm. The adjacency matrix used for this example is table 1.

read a balanced mid level of energy. This is the state the robot is in when it is locked on to its leader at a maintainable distance. In this work, the researcher selects **G** from the clusters in the codebook. Once the goal state is identified, the paths from other less desirable states need to be found. It is anticipated that this process would be part of the initial conditions when these algorithms are integrated into a robot control architecture.

The paths from less desirable states to the goal state can be found using a standard graph algorithm such as Floyd's algorithm (Floyd, 1962). Floyd's algorithm is a dynamic programming (Aho et al. 1974) method that finds the shortest path from every node to every other node. Figure 4 depicts a graph and its corresponding adjacency matrix. Notice that the values along the diagonal of the matrix are assigned to zero, indicating that there is no cost for staying in a node. The algorithm is shown below.

Floyd's Algorithm

The variables are defined as follows:

- `number of nodes`: the number of clusters in the codebook of situations created from the recent memory M.
- `adj`: adjacency matrix created from compression process.
- `m`: square matrix that is of size number of nodes × number of nodes.
- `i, j, k`: indices

The algorithm is shown below:

```
/*  Initialize each m[i,j] with the distance between node
i, j. */
For(i = 0 to number of nodes -1) {
      For(j=0 to number of nodes -1) {
            if (i  ≠ j) {
                  m[iI][j] = adj[i][j];
            }
            else {
                  m[i][j] = 0;
            }
```

```
      }
}

/* Solve for shortest paths. */
For (k = 0 to number of nodes – 1) {
      For (i = 0 to number of nodes – 1) {
            For (j = 0 to number of nodes – 1) {
                  m[i, j] = min { m[i, j], m[i, k] + m[k, j]
                  }
            }
      }
}
```

Note that the 3rd nested for loop in the shortest path section is remarkably similar to the comparison statement (statement 4a) used in the Q Learning algorithm shown below.

Q Learning Algorithm

```
s  - current state
s' - the state in the set of states adjacent to state s
      that contains a'.  (s', a') is the optimal
      state/action combination adjacent to s.
a - current action
a' – action that returns the greatest reward at state s'
d - amount that rewards are discounted, range is a real
      number between 0 to 1
r – immediate reward available for current state
Q'(s, a) – estimate of Q value at state s with action a.


For each (s, a) initialize the table entry Q' (s, a) to
zero
Observe the current state s
Do forever:
1.    Select an action a and execute it
2.    Receive immediate reward r
3.    Observe new state s'
4.    Update the table entry for Q' (s, a) as follows:

   Q' (s, a) = r + d*max Q'(s', a')
```

This statement is adding the immediate reward available at state s to the maximum reward that is available for all actions at state s′. State s′ is the state that results from doing action a at state s. So this statement is taking the place of the inner loop of Floyd's algorithm. Note in this algorithm the reward is maximized as opposed to path length being minimized in Floyd's algorithm.

```
5.    s = s'
```

Effectively both algorithms are using dynamic programming methods to iteratively optimize their target functions. The main difference is that the result of the Q Learning algorithm is a table of states and scores for each action at each state. It does not have state-to-state transition information as does Floyd's algorithm. Floyd's algorithm not only finds the length of the shortest path from every node to every other node, it also can be used in conjunction with a path matrix so that the individual nodes along each route can be extracted.

By clustering the recent memory into groups of sensor/action pairs, the paths from lesser states to the desired state represent the best way to get from the lesser states to the desired state that has been experienced so far. It is possible that more direct routes exist between the states, but if they have not been experienced yet in the recent memory, the adjacency matrix will not contain the data to represent them.

The paths themselves are the key focus in learning using the graph. In the case of the following robot, a typical shortest path will start off at a low intensity level and be turning one direction or the other. Each of the subsequent nodes typically will have a higher intensity level and will be turning the same direction as the starting node. Then within one or two nodes before the goal node, a node similar to the goal node, but with less energy appears, often followed by another with still higher energy, and then finally the goal node is reached. Recall that Figure 3 from the previous section shows a graph for a recent memory.

Looking at the graph in Figure 3 it can be noted that shortest path from state 3 to the goal state, 0, is as follows:

Path [3 to 0] = {3, 2, 1, 0}

Similar paths exist from states 6 and 8 to state 0. They are shown below:

Path [6 to 0] = {6, 10, 1, 0}
Path [8 to 0] = {8, 9, 1, 0}

Once the paths have been formed, the nodes that comprise them are used as a "filter" to create training examples from the recent memory. The filter derived from the graph in Figure 3 is shown below:

Filter Clusters = Path [3 to 0] $\cup$ Path [6 to 0] $\cup$ Path [8 to 0]
Filter Clusters = {3, 2, 1, 0, 6, 10, 8, 9}

These clusters are deemed to be good clusters because, as stated earlier, each one is part of the most efficient route to the stable state. Data that fits into this set of clusters is defined as usable data. Clusters 4, 5, and 7 were not put into the filter because they were not on a shortest path to the goal state, 0. Being included in the filter is purely a matter of being on a shortest path to the goal state, which is determined using Floyd's algorithm.

In this example it is easy to see that cluster 4 would not be a good training cluster because it would try to teach the FFNN to turn left when the sound source is closer to the right sensor (in this case L = 7, which is less than R = 10, so the source is closer to the right). The wrong decision in cluster 4 leads to a lower intensity value in cluster 5. Cluster 5 is similar to 4, and 7 illustrates the same concept except the direction is different.

Identifying data to be put into the training set is done by looping through the recent memory and extracting sequences of data that are consistent in action and fall within the set of "usable" clusters. Using this scheme a training file would contain a finite number of examples. Within an example, the action is consistent. Only data that falls within the clusters in the "filter" is used.

As soon as the action changes or data is found that the filter rejects, the example is ended. This process is done until all the data in the recent memory is evaluated. An outline of the algorithm is shown below.

The variables are defined as follows:

- M : recent memory
- P : Pointer into M
- SA : Sensor/action pair
- DV : data vector comprised of a cluster size list of sensor/action pairs from M

  DV = {SA[P], SA[P+1], ….SA[P + (cluster size-1)]}

- codebook : the library of situations found using the RBF clustering algorithm
- codebook = C[0] …..C[number_of_clusters-1]
- closeIndex : index that points to a cluster in the codebook
- Filter Clusters : the set of clusters that lie on the shortest paths to the goal state **G**

The algorithm is shown below:

```
P = 0;
While (P has not passed the end of the recent memory M) {
      DV = cluster_size sensor/action pairs from point P in
                  the recent memory;
      closeIndex = index of the closest cluster in codebook
                  to DV;
      If ((all the actions in DV are the same) and
      (closeIndex ∈ Filter Clusters)) {
            Write DV to file as a usable example;
      }
      P = P+1;
} /* End while. */
```

The training set developed by the above process is written out to a file which was read by GA based FFNN generation system (McDowell et al. 2002). The resulting FFNN developed using this method was tested on a simulated following robot in the simulator system developed for this research. Results are presented in the next section.

3 Case study: Simulation of Leader/Follower Robots using Acoustic Ears Sensor System

The primary motivation for this work was to develop an adaptable control system to be used in low cost underwater vehicles for the purpose of surveying and searching (McDowell et al. 2002). The idea was to use a highly capable unmanned underwater vehicle (UUV) with a low drift absolute positioning system as the team leader and extend its sensor footprint with a team of less expensive UUVs relying on relative navigation. Formation maneuvering using relative navigation based on acoustic sensors is vital to this operational concept because traditional navigation instruments such as GPS and LORAN do not operate effectively in the undersea environment. In the interest of keeping costs down a “two ear” acoustic system was developed.

It was first simulated using a brute force simulation, and then over time it was progressively migrated towards a physical testing scenario that was done using mobile robots, each mounted with two microphones as ears and a speaker so that lead robots could chirp while follower robots homed in on their signal. Testing of algorithms was carried out using both a computer simulation and lab testing with mobile robots. The tests of the graph algorithm have been done in computer simulation while the behavior algorithm used for comparison has been tested both in simulation and in the lab.

In order to test the graph learning method a simulation using LabView and C was created. The FFNNs developed using the graph learning algorithm were first tested for viability with a simple game of cat and mouse in which the robot using the FFNN chased an operator controlled robot. For formal measurements, a leader/follower task was set up in which the follower robot's task was to follow a computer controlled leader robot for two laps around an oval test track. Inter-robot distance and heading differences were recorded. As a comparison, a behavior controller that considered past values was also tested using the same testing procedures.

The FFNNs created using graph learning employed recent memories collected in the simulator using a "random but purposeful controller". Briefly, this controller is a non-deterministic goal oriented wander program that operates under the following simple rule set:

- If the goal parameter is stable or increasing, keep doing the current action.
- If the goal parameter is decreasing, randomly select an action.

Because there is no association of sensor readings with optimizing the goal, the effect of the controller is to provide a recent memory in which the robot has experienced a wide variety of situations. A graph was generated from the recent memory using the RBF clustering method described earlier, and the graph learning algorithm built a training set which was used by a GA based FFNN generator.

The FFNN produced were then tested in a game of cat and mouse in which the machine generated "cat" controller chased an operator guided "mouse". When it was evident that the "cat" could follow the mouse, a more formal test in which measurements could be made was undertaken. To provide consistency between algorithm comparisons, the test track shown in Figure 5 was used. As can be seen from the figure the track is oval, (really a square with rounded corners). The test run data is depicted in Figures 6, 7, 8, and 9. These results show that the robot was able to hold a stable distance from its leader, and that the path taken was free of oscillations. As can be seen on Figures 6 and 8, the inter-robot distance fluctuated eight times during the run. These fluctuations correspond to the time when the lead robot is turning and the follower is changing its course in order to remain close. The heading difference between the robots is shown in Figures 7 and 9. In these graphs the peaks correspond to the time periods when the course changes are being made.

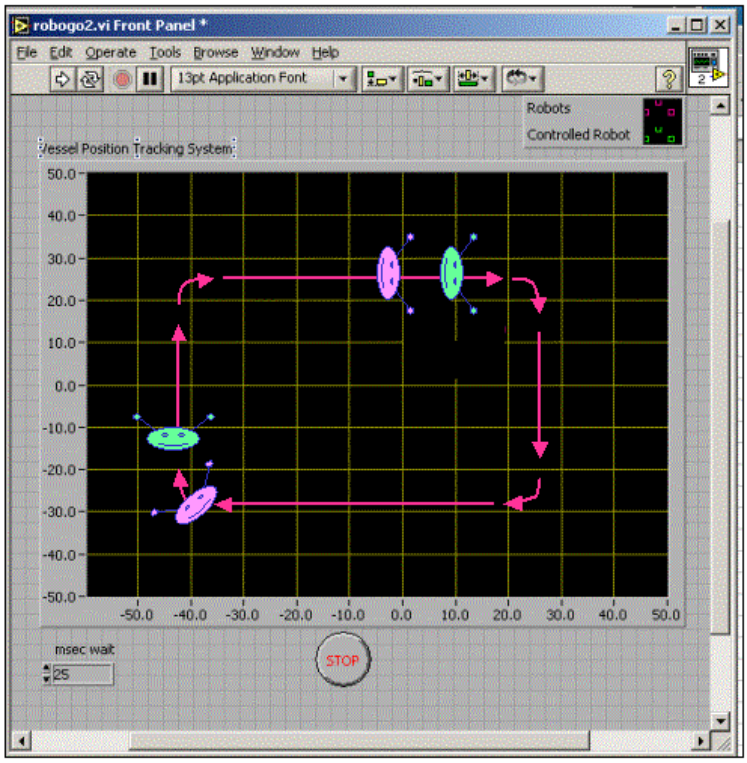

Figure 5. The test course for the following robots. The lead robot (green smiley face) is controlled by the simulator. The follower robot (pink smiley face) is controlled by the current controller being tested. Over two laps the inter robot distance and course differences are recorded.

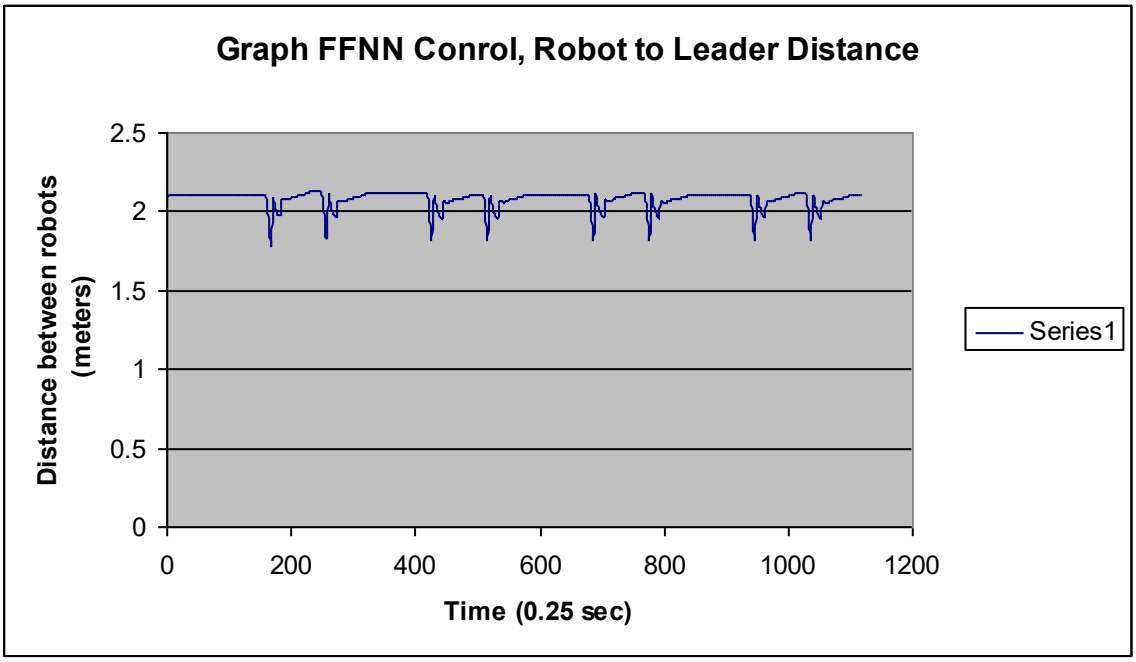

Figure 6. The leader/follower distance for a follower robot using a non-reactive FFNN created using the graph learning algorithm. Non-reactive in this case means the FFNN looks at current and past sensor values.

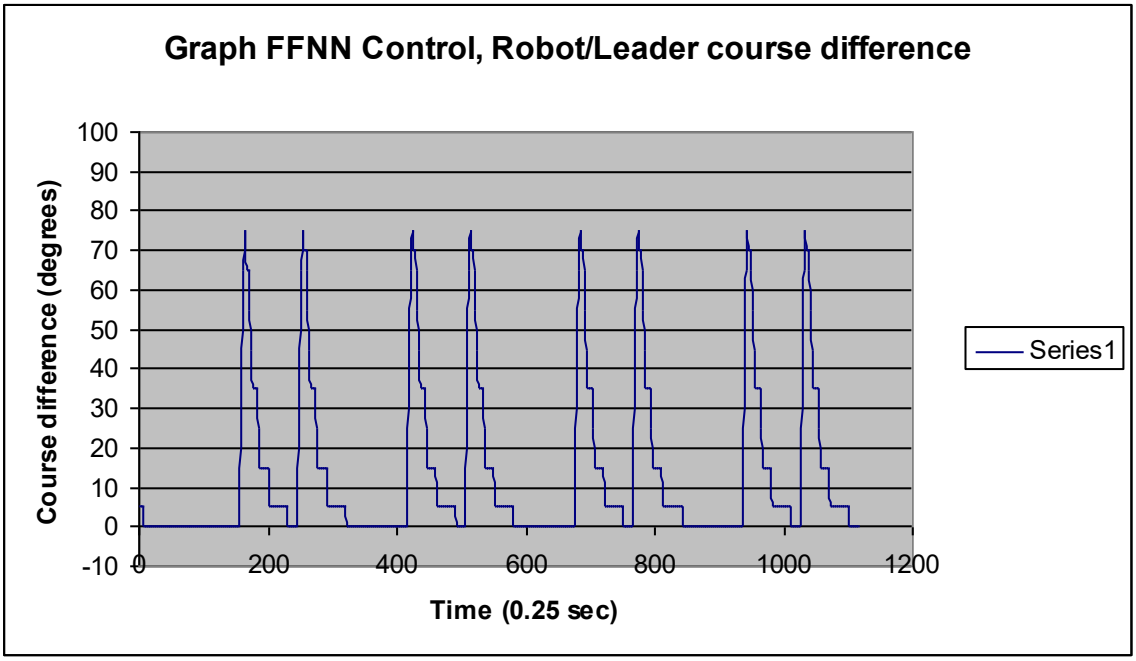

Figure 7. The leader/follower course difference for a follower robot using a non-reactive FFNN created using the graph learning algorithm.

Figures 6 and 7 above were made using a controller generated with the graph learning algorithm. The FFNN in these tests is characterized as non-reactive in that it looks at both current and past sensor values. Its output is a heading increment that can be one of three values: left, straight, or right. The inter-robot distance curve shows that the robot can maintain a stable distance behind its leader. In the cases where the robots are traversing the short side of the oval, there is almost not enough time for the distance to stabilize. The heading difference curve in Figure 7 shows that the robot is guiding itself in stable manner without oscillations.

As an evaluation tool a hand coded behavior based controller was tested (McDowell, 2005). The behavior controller was used as a comparison tool because it made use of past sensor values as did the FFNN controller. The behavior controller had three modes: follow, seek and search. Arbitration was based on sensor gradient histories. The behavior controller has been shown to be effective in both simulation and onboard mobile robots. Comparing Figures 7 and 9 it can be seen that the behavior method oscillates a little compared to FFNN. This is because as the behavior method transitions between seek and follow modes, its sensitivity to sensor oscillations is higher than if it is purely in follow mode. The following distance for the FFNN controller is a little larger than that of the behavior controller. This is due to the gain settings on the sensors used for this test.

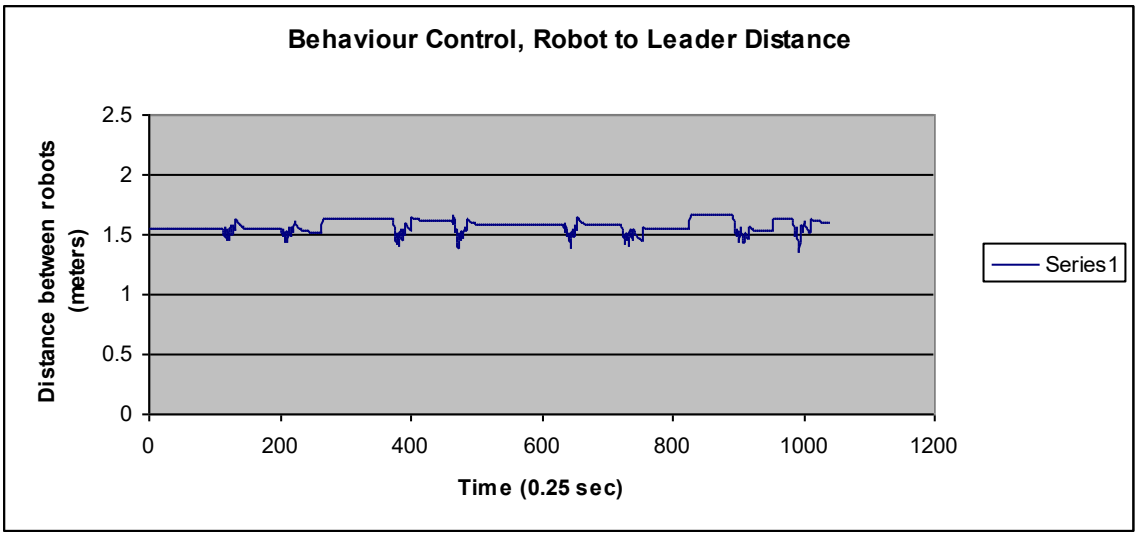

Figure 8. The leader/follower distance for the two lap test when the follower is using the behavior controller.

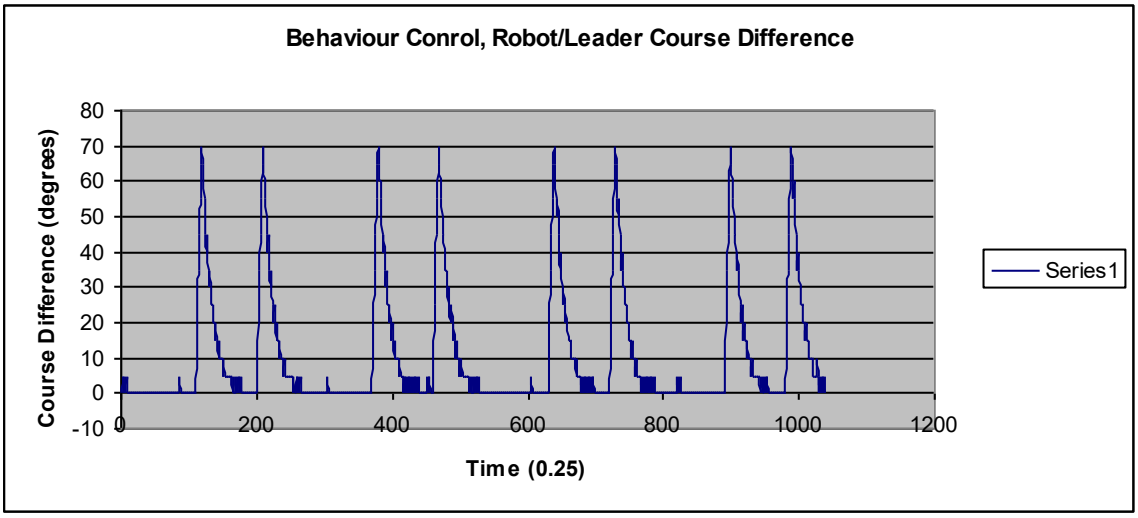

Figure 9. The course difference between the leader and follower when the follower is using the behavior controller. Each peak corresponds to when the leader is making a turn. Notice that this controller takes more time to complete turns than the reactive controllers. The thick blue lines at the end of each peak denote severe oscillation at the tail end of each turn. This is caused by the robot being in seek mode. In seek mode, the algorithm is more sensitive to sensor fluctuations enabling the robot to quickly turn to its source, but at this level, it no longer damps oscillations.

One promising feature of the graph learning algorithm is that the directed graph lends itself to solution space reduction. Using the graph, and the primitive rules of symmetry, interpolation, and extrapolation, the time required to explore the environment could be decreased. Figure 10 shows a graph that has had nodes inserted into it that were not experienced. These nodes could be marked so that when they were experienced, an observer module could confirm that they were correct. To test these ideas, symmetry methods were used to reduce the size of the recent memory that had to be experienced in order for a following robot to learn to turn both directions correctly. The recent memory to which the symmetry rules were applied had many more

examples of the robot turning right than left. Symmetry was exploited by mirroring the right turns to artificially increase the number and variety of left turn examples.

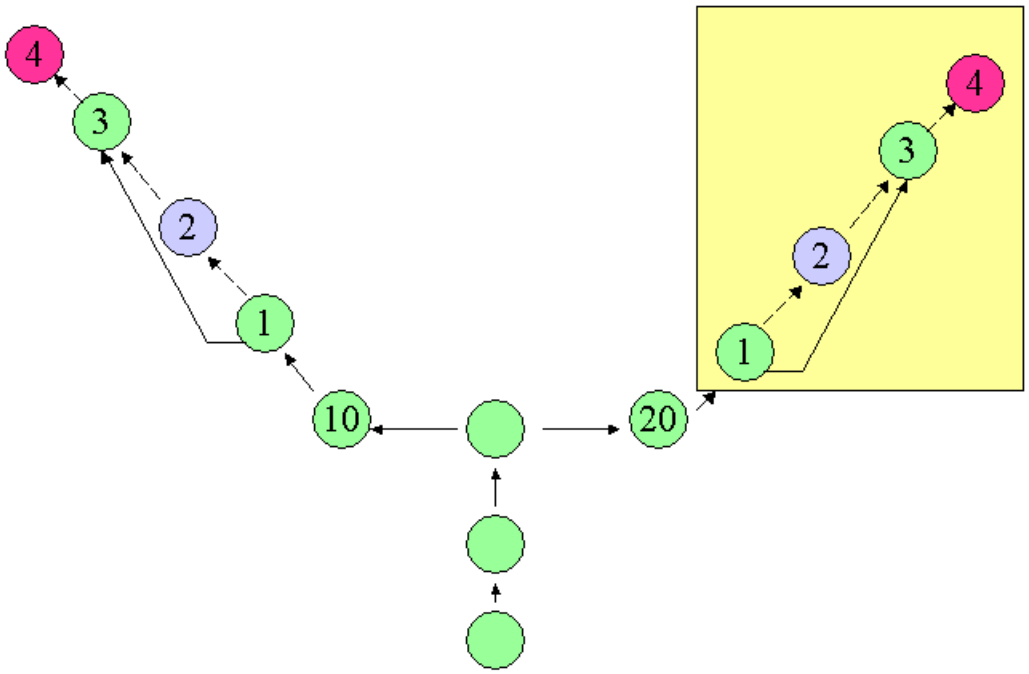

Figure 10. A situation graph that was built from a recent memory. The green nodes, excluding those in the yellow box were present in the recent memory. The arrow from state 1 to 3 is what was experienced. Using interpolation, node 2 was inserted, and using extrapolation, node 4 was inserted. Using symmetry, a symmetric copy of nodes 1 to 4 was built and connected at node 20. The yellow box around those nodes denotes that they are nodes created using symmetry.

Looking at the graph in Figure 10, seven of the thirteen total nodes were in recent memory. Assuming that the rules of symmetry, interpolation, and extrapolation were applied conservatively, the robot could have made an almost 50% reduction in environmental exploration. Using a simple sensor mirroring technique, the symmetry method has been tested in a robot following task (McDowell, 2005).

5 Conclusion

One of the major problems with learning routines is that almost without exception, in order to learn something, it had to be experienced. The learning routines developed here allow the learning from recent memories so that sequences of actions do not have to repeated until the desired result is obtained, but somewhere in the memory the desired result must be present. However, humans and animals can learn without having to directly experience each thing that is learned. For example, when a boy is learning to pound nails into boards he learns quickly that it hurts, and hurts a lot, when he hits his thumb with the hammer. Usually he holds the nail with his thumb and index finger, and he knows if he were to hit his index finger with the hammer it would also hurt. He also knows that if he had switched hands with the hammer and the nail and hit the thumb on the other hand, it too would hurt. So it stands to reason that certain assumptions are being made.

Symmetry was being used in the above example when the boy decided that it would hurt if he hit his thumb on the opposite hand. Other primitive assumptions may also be useful, such as

interpolation and extrapolation. For example, assuming that the index finger would hurt may be an extrapolation based on hitting the thumb, or if the thumb had been hit and the index finger had been hit, knowing the index finger would also hurt could be thought of as an interpolation.

From the outset, the goal of this research was to develop a methodology for controlling robots that would enable them to function without an abundance of a-priori knowledge embedded in either databases, maps, models, or rule systems. Furthermore, since the target application was a physical system, the method developed needed to avoid undue physical trials of solutions, so that time could be saved and mechanical systems not be overstressed. Although the goal of this research was to produce a general methodology, there was a specific test application in mind, autonomous vehicle/robot formation maneuvering, with the end target application being formation maneuvering with UUVs.

The graph learning method presented in this paper is a three-step process summarized by the following bullets:

- Collection of a recent memory with sufficient information to fulfill the requirements behavior generation.
- Organization of the recent memory into a directed graph. The nodes represent situations encountered by the robot and are found by clustering the recent memory. The graph's connectivity is determined using a compression process.
- Learning accomplished by identifying a goal state in the graph, then finding the shortest paths from lower states to the goal state. These paths are used to create a training set for a GA process that generates a FFNN that in turn controls the robot. The graph can be used for other control functions such as observation, next state prediction, and solution space reduction.

This learning method fulfills the original goals in that it allows a robot to function without a-priori knowledge embedded in databases, maps, models, or rule systems. All that is required is a description of the goal state in terms of sensor/action values. It allows the system to avoid undue physical trials of solutions, by learning from previously experienced situations, and by using techniques such as sensor mirroring. It shares some similarities with Q Learning, but is different in that the table built in Q-Learning does not preserve information on the sequence of sensor/action relationships to one another. The graph-based technique organizes the situations experienced in the recent memory into a data structure that can work in conjunction with a FFNN. Besides providing an effective means to learn, the graph will allow features not usually associated with Q-Learning, including tracking of progress towards goals, prediction of sensor values based on actions, selection of alternate routes, and prediction of non-experienced states.

6 References

Aho, A., Hopcraft, J., & Ullman, J. (1974). *The Design and Analysis of Computer Algorithms*, Addison Wesley, 67–69.

Dijkstra, E. W. (1959). A Note on Two Problems in Connection with Graphs. *Numerische Math., 1*, 269-271.

Floyd, R. W. (1962). Algorithm 97: Shortest Path. *Communications of the ACM, 5* (6): 345.

Grefenstette, J. J. and Schultz, A. C. (1994). An evolutionary approach to learning in robots, *Machine Learning Workshop on Robot Learning*, New Brunswick, NJ

Hagan, Bemuth, & Beale (1996). *Neural Network Design*, Campus Publishing Service, University of Colorado, page 7.2.

Kushleyev, Vohra (2004), "Sound Localization," University of Maryland,

McDowell, P, (2005). Biologically Inspired Learning System. Ph.D. dissertation., chapter 4, Louisiana State University.

McDowell, P., Chen, J. & Bourgeois, B. (2002). UUV Teams, Control From A Biological Perspective, *Proceedings of the Oceans 2002 MTS/IEEE Conference*, Biloxi MS, 331-337;

Mitchell, T. *Machine Learning*, McGraw-Hill, 1997, page 105

Mitchell, T. *Machine Learning*. McGraw-Hill, 1997, pp 367-386.

Parker, G. B. (2002). Punctuated Anytime Learning for Hexapod Gait Generation. *Proceedings of the 2002 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems*, EPFL, Lausanne, Switzerland.

Reeve, R. and Webb, B. (2003). New neural circuits for robot phonotaxis, *Philosophical Transcations of the Royal Society*, 2245-2266

Rucci, M, Wray, J., & Edelman, G. M. (2000). Robust localization of auditory and visual targets in a robotic barn owl. *Robots and Autonomous System*, *30*, 181-193

Shah, Vinah (2003), Practical Implementation of the Sound Localization Algorithm on a Robot, Rensselaer Polytechnic Institute, Research Experience for Undergraduates

Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, Vol. 38, No. 3, 58 - 68