Inferring Task Structure From Data

Paul E. Utgoff
David Jensen
Victor Lesser

Department of Computer Science
140 Governor's Drive University of Massachusetts
Amherst, MA 01003 U.S.A.

Technical Report 00-54
November 9, 2000

Abstract

An algorithm is presented for fitting an expression composed of continuous and discontinuous primitive functions to real-valued data points. The data modeling problem comes from the need to infer task structure for making coordination decisions for multi-agent systems. The presence of discontinuous primitive functions requires a novel approach.

1 Introduction 1

2 Multi-Agent Task Structure 1

3 Task Structure Inference as Function Fitting 2

4 The BEFIT Fitting Algorithm 3

4.1 Generating Children 4

4.2 Node Ordering Heuristic 5

5 Illustrations of the BEFIT Algorithm 5

6 Summary 8

1 Introduction

Scientists and engineers often confront the problem of inferring structure from data. How does the value of one variable relate to the value of another? To the extent that such questions can be answered, one has knowledge of the structure of the domain. Such modeling problems, wherever they originate, are almost always of interest to researchers within the field of Machine Learning because to infer structure is to learn.

One such modeling problem comes from the study of multi-agent systems. For a group of agents to solve a problem as well as possible, one must consider how to coordinate the subtasks among the agents. Coordination involves deciding which agent will do which subtasks, and in what collective order the subtasks will be done. Assigning subtasks to agents is only one piece of the problem because the order in which each agent accomplishes its tasks can affect the ease or difficulty of problem solving for other agents. One can gather data about how the subtasks were assigned and ordered, and the global quality of the solution that resulted. From such data, one would like to infer the effects of various coordination choices.

One approach to multi-agent coordination is to create a wrapper around an agent which deals with how an agent's activities should be selected and ordered to achieve coherent activities with other agents. An example of such an approach is Generalized Partial Global Planning (GPGP) (Decker & Lesser, 1995). This approach requires an agent's wrapper to have knowledge about several aspects of an agent's possible activities, including the actions an agent can take (called *methods*) and:

1. ordering constraints among the methods,
2. behavioral characteristics of methods such as the probability distribution on their duration and the local quality of their results, and
3. how the choice of specific methods determines the global quality achieved.

This knowledge is used by a GPGP module in each agent to establish coordination commitments among agents about when certain tasks will be achieved and with what local quality. These commitments, together with local constraints such as deadlines and objective criteria, are used by a scheduling component in each GPGP module to provide the agent with ordering constraints on its activities that will result in coherent activities among agents.

A major shortcoming of such approaches to multiagent coordination is the engineering required to add the above knowledge to an agent, particularly if that agent is a pre-existing system that is being incorporated into a new multi-agent system. Thus, learning is a valuable capability in adapting preexisting agents for use in a multi-agent context. In past work (Jensen, Atighetchi, Vincent & Lesser, 1999), we have analyzed traces of agent behavior, and have been able to learn all these types of knowledge except the last — how the specific choice of methods and their associated local quality of methods determines the global quality of the solution. Filling this remaining gap would allow learning of all the knowledge necessary for a multi-agent system to make high-quality coordination decisions with GPGP.

Below, we describe in greater detail the problem of inferring task structure for multi-agent coordination. We then restate the problem in terms of a discontinuous function fitting task, and present a new algorithm for solving it.

2 Multi-Agent Task Structure

We adopt the existing TÆMS (Task Analysis, Environment Modeling, and Simulation) framework for multiagent coordination (Decker & Lesser, 1993) that has previously been used to represent the knowledge required by GPGP. TÆMS represents quantitative information about agent activities, including candidate actions and how those actions can be combined to achieve high-level tasks. A task is represented as a tree whose root is the overall task and whose leaves are methods. A method represents actual instantiated computations or actions that the agent can execute. Methods can be characterized by their duration and cost, and the results of methods can be characterized by their local *quality*. In short, a TÆMS task structure is a compact representation of a network of plan alternatives, and a method of calculating the performance characteristics (quality, duration, and cost) of those alternatives.

Internal nodes in a TÆMS task structure represent subtasks composed of other tasks and methods. Internal nodes also specify functions that indicate precisely how quality will accrue depending on which methods are executed and when. These quality accumulation functions (QAFs) define how the results of constituent subtasks and methods combine to determine the global quality of the overall task. For example, consider a task with three subtasks, A, B and C, as its

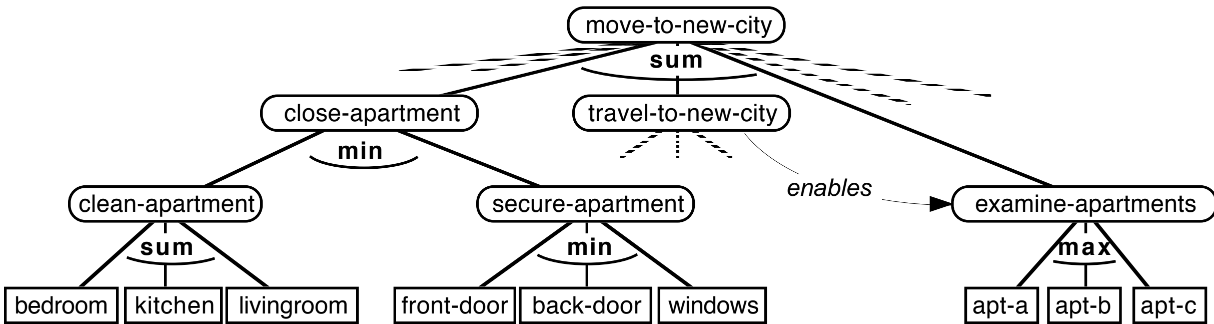

Figure 1. Example Task Structure

descendants. The quality of these subtasks can be accrued in several ways to yield the global quality. Consider three scenarios:

- Suppose that the task is to clean a house and the subtasks are clean-bedroom, clean-kitchen, clean-living-room. In this case, the more that gets done, the better. The quality of this process could be measured by the total quality accrued from all three tasks — the *sum* of the individual qualities.
- Suppose the task is to select an apartment, with three methods examine-apartment-A, examine-apartment-B, examine-apartment-C. Only a single apartment will be selected, and the final quality is the quality of the best apartment — the *maximum* of the individual qualities.
- Suppose the task is to secure a house, with subtasks secure-windows, secure-front-door, secure-back-door. All subtasks must be accomplished, and the final quality depends on the least secure entry point — the *minimum* of the individual qualities.

In these three situations, the task structures are similar, but accurate representation of the task requires different QAFs: SUM, MAX, and MIN.

TÆMS represents other important characteristics of tasks, in addition to the hierarchical task structure and the QAFs. First, interrelationships between tasks or methods indicate where the execution of one method will affect the quality or duration of another method. For example, executing method A may enable execution of method B. In other words, B cannot be successfully performed before A is successfully completed. Second, each method is characterized by frequency distributions indicating possible values of performance parameters such as quality, duration, and cost.

Figure 1 shows a fragment of a TÆMS task structure for moving to a new city. The task structure incorporates the elements discussed above (e.g., selection of an apartment) into a larger structure, illustrating the hierarchical nature of TÆMS, as well as the incorporation of other elements such as interrelationships among tasks. For example, the task structure shows an “enables” relationship between the tasks “travel to new city” and “examine apartments” — indicating that one must be physically present in a new city to examine apartments. While one might argue with specific aspects of a particular task structure (e.g., should the the “close apartment” task have a min or sum as its QAF?), TÆMS can be used to express a wide variety of such structures.

3 Task Structure Inference as Function Fitting

To induce the TÆMS task structure from a collection of examples, it is sufficient to fit the examples with a function composed of the QAF primitives *MIN*, *MAX*, and *SUM*. Each example indicates which methods were executed, the order in which the methods were scheduled, the local quality obtained for each method, and the global quality achieved. One can think of each local method as a variable, and the global quality as the target value of the function for those variables. The goal is to induce a single function composed from the primitives that produces the correct target value for all of the examples.

Although the order of the methods (variables) has an impact on the global quality achieved, the global quality (function value) is computed as a composition of the primitives over the local qualities (variable values). Each of the primitive

functions is well defined and is applied solely to particular observed values. Other information does not bear on the computation of a primitive function. For example, *MIN(A,B)* evaluates based solely on the values of A and B, not other information. For this induction task, the ordering information can be ignored safely.

This function fitting task differs from more familiar forms of function approximation in two important ways. First, an exact fit can be found. Even when there is variation or noise in the local quality measurements, and even when interaction effects due to ordering are present, the final combination of these values into the global quality is a function composed of the primitive functions *MIN*, *MAX*, and *SUM*. Because there is no error in the computation of these primitives, the global quality is computed exactly from the given local qualities.

The second important difference is that the primitive functions *MIN* and *MAX* have discontinuous derivatives. Were it not for this, one might consider adapting standard parametric approaches such as gradient descent (Duda & Hart, 1973; Press, Flannery, Teukolsky & Vetterling, 1988) that iteratively reduce error. We require a function that is composed of the primitives as deeply as is needed to fit the data. The two primitives *MIN* and *MAX* have the special property of *selecting* one argument as the function value. For example, *MIN(A,B)* selects A when $A<B$, and excludes B. This notion of value selection is important in the discussion below.

A naive approach would be to search the space of expressions over *MIN*, *MAX*, and *SUM* in an uninformed manner for an expression that evaluates correctly for all the examples. One might try simply to be correct for the largest number of examples, but that is disappointing for a problem in which an exact fit is known to exist. This will not work however because, as is often the case, an uninformed search is intractable.

Consider the size of the space of expressions, which is finite. Throughout the discussion, we consider just the binary form of the primitive functions. For example, *MIN(A,B,C)* would be represented as one of *MIN(MIN(A,B),C)*, *MIN(MIN(A,C),B)*, or *MIN(MIN(B,C),A)*. There are more possibilities because all three primitive functions are symmetric. However, one can generate expressions in a manner that does not duplicate the various nested binary expressions that are equivalent to a single n-ary primitive function. Thus we can work with the binary forms without affecting the complexity.

Assume that one builds an expression tree by repeatedly joining a pair of disconnected expressions by a primitive function. Initially, every local method is represented by the base form of an expression tree, which is a leaf. A leaf is marked as such, and also records how to access the appropriate value for that leaf from an example. For n local methods, there are initially n disconnected leaves. By repeatedly joining two disconnected expression trees with a primitive function, the initial set of disconnected leaves eventually becomes a single connected expression tree.

How can we know which pair of disconnected expression trees to connect, and by which of the primitive functions *MIN*, *MAX*, or *SUM*? For an uninformed search, one must consider all the possibilities. For n local methods, there are initially $n(n-1)/2$ possible pairs, and 3 possible primitive functions for each of those, making $3n(n-1)/2$ possible new expressions. For each of these, there are now $n-1$ expressions, making $3n(n-1)/2 \cdot 3(n-1)(n-2)/2$ possible new expressions. Continuing this expansion for the n factors shows that the number of possible expressions is $O(n^n)$. An informed search is essential in order to navigate this space at all productively.

4 The BEFIT Fitting Algorithm

The BEFIT (Best-first Expression FITter) algorithm solves this function fitting task with a combination of best-first search and constraint satisfaction. One can control growth of the search tree by early refutation of every candidate expression that must be incorrect for at least one example. Among those candidate expressions that are correct for all the examples, a measure of evidentiary support guides the search to consider the more promising candidates earlier rather than later.

As described above, BEFIT initializes the root node as the list of disconnected leaf expressions. A leaf simply specifies which local method's quality measure should be indexed in an example. A goal node is one in which the list of disconnected expressions is of length one, which is equivalent to having one completely connected expression. Because all expressions listed in a node are individually correct for all the examples, a list of one expression is guaranteed to be a correct solution. The best-first search removes the most promising node from the open list. If it is not a solution, then its potentially correct children are generated, as described in Section 4.1, and inserted into the open list according to the metric described in Section 4.2.

Global	87.75
A	20.09
D	20.09
F	20.09
L	20.09
M	7.39
Q	20.09
T	148.41
W	20.09
Z	20.09

Figure 2. A Data Point

4.1 Generating Children

We would like to generate only those children that could be on the path to a solution expression. If a node has k disconnected expressions, then there are $3k(k-1)/2$ candidates to consider. A great many of these candidates need not be generated as child nodes because they would be incorrect for at least one example. No descendant of an incorrect node can ever be correct for all the examples.

Consider the data example shown in Figure 2. The global quality measure (function value) is indicated by 'Global', and each of the local method values (variables) is indicated by a single letter. Suppose that the node being expanded contains nine disconnected expressions, each one being a leaf expression. There are 108 candidate children for the node. For this example, it happens that there are 35 combinations of the expression values that sum to the target global quality. Among these possibilities *for this example*, expression T is always excluded, and expression M is always included. This permits three different ways of eliminating candidate children.

First, because T is always excluded, eliminate every candidate *MAX(T,x)* for which $T>x$. Such a candidate must be wrong, and all of its descendants would therefore also be wrong. Also eliminate every candidate *MIN(T,x)* for which $T<x$. Second, because M is always included, eliminate every candidate *MAX(M,x)* for which $M<x$. Also, eliminate every candidate *MIN(M,x)* for which $M>x$. Finally, because T is always excluded and M is always included, eliminate *SUM(T,M)*. These steps eliminate nine candidates when considering just this one example. By considering all the data, one achieves significant pruning of candidate children. Let's state this approach algorithmically.

Enumerate the 2^k possible ways to include or exclude each of the k expressions such that summing just the included values produces the target global quality measure. This notion of selection follows from our earlier observation that the primitives *MIN* and *MAX* act as selectors. We refer to each combination of include/exclude choices an *interpretation*. If there is no interpretation under which a candidate expression holds (evaluates correctly) for this example, then that expression is refuted and removed as a candidate child.

Some care must be taken in refuting candidates. For example, suppose that an expression that will be part of a correct solution is *MAX(A,B)*. When $A>B$ and B is included in every interpretation, then *MAX* is refuted. Note that *MAX* is not refuted when B is excluded in any interpretation, regardless of its numerical relationship to A. This is because *MAX(A,B)* may itself be nested somewhere below a selecting primitive *MIN* or *MAX*. If *MAX(A,B)* need not be selected, then the relationship of A and B is irrelevant and cannot serve as a refutation.

This pruning of necessarily incorrect candidate expressions provides a drastic reduction in the number of children that are produced and potentially considered further. However, the need to enumerate 2^k possible interpretations is worrisome because it consists of an exponential search. On present-day machines, k much beyond 20 becomes expensive. Fortunately, for the time-being, the largest TÆMS task structures are smaller than this.

The BEFIT algorithm does not typically need to generate all 2^k possible interpretations. The space of possible interpretations is itself searched and pruned safely. Think of the most general possible interpretation as including all k of the expression evaluations. Compute its sum, and package this most general inclusion mask and its sum as the root node of this subsearch. Now conduct a depth-first search, where at each step an included variable becomes excluded, updating the node's sum by a single subtraction. If the sum has dropped below the global quality measure in the example, discard it, as further exclusions can never help. Otherwise, place it onto the head of the open list of the subsearch. If the sum matches that of the example's global quality measure, then the mask in the node indicates an

interpretation. When the open list is exhausted, all possible interpretations (sum of included values matches global measure) will have been considered. It is trivial to generate the mask children uniquely so that there is no duplication of effort.

4.2 Node Ordering Heuristic

Generating only the potentially correct children of a node provides valuable guidance away from many fruitless search avenues. For the children that are generated, additional guidance is needed. This takes the form of a heuristic function of a node. Because a node contains a list of disconnected expressions, one would expect that a good heuristic function would measure the promise of the set of expressions at the node. However, of the many heuristic functions we have tried, the best one to date measures (explained below) just the expression created most recently. Recall that a new expression is created by joining two existing expressions by a primitive function.

The measure computed for the most recently created expression at a node is the total number of interpretations with positive support across all the examples. Positive support is present when the primitive function must evaluate correctly for its arguments. As noted above, it may be the case for some interpretations that both argument values are excluded, in which case their values are irrelevant. Such interpretations are not counted as positive support.

A greater number of interpretations with positive support are presumed to provide greater freedom downstream in the search. This measure alone is not quite enough because shorter expression lists tend to have fewer total interpretations. To compensate, the composite heuristic is to order nodes on the open list first by the number of expressions (fewer is better), and second by the positive support count (higher is better). This renders BEFIT's best-first search as a heuristically guided depth-first search, but we shall continue to characterize it simply as best-first search.

5 Illustrations of the BEFIT Algorithm

The algorithm has been tested on artificial data that was generated by a simulator available in the TÆMS project. Three data sets are discussed here. All use the same task structure, but they vary in the kinds of interaction effects. The first problem (p5) has no interactions of any kind. The second problem (p6) contains one 'enable' and one 'facilitate' interaction, and the third problem (p7) contains one 'disable' and one 'hinder' interaction. In each of the three cases, the BEFIT finds the task structure from the data. It does not identify the interaction effects because these are considered to be apart from the task structure. We expect that it will not be difficult to identify these interactions, but this has not yet been done. Presence of interactions alters various local qualities, and hence the global quality.

Below is a trace of the BEFIT algorithm as applied to the p5 data set. Other than two notes about cpu consumption, each block summarizes information about the node just expanded. 'Support' is the measure of positive support as described above in Section 4.2. 'Open' is the number of nodes on the open list of the best-first search. 'Expressions' is the number of disconnected expressions at the node. Recall that disconnected expressions are combined repeatedly in an effort to produce just one connected expressions.

The next lines, until but not including 'Generated ...' simply list the disconnected expressions at the node. The leaf expressions, each corresponding to a single local method, are all listed on just one line. Each local method (variable) is indicated by a single capital letter. Each of the non-leaf expressions is then listed on a separate line. Each primitive function is implemented as a function of two arguments, but the expression printer causes it to to appear as a function of any number of arguments wherever possible. For example, an expression *F(F(x,y),z)* would print as *F(x,y,z).* The final line shows how many potentially correct children were generated from the number of candidate children.

```
1.28 cpu seconds to load 1185 examples

Root: Open: 1 Expressions: 15
  Z W V U T S R Q M L K G F D A
Generated 30 of 315 possible successors

Support: 54530 Open: 30 Expressions: 14
  A D F G L M Q S T U V W Z
  max(R,K)
Generated 24 of 273 possible successors

Support: 40775 Open: 53 Expressions: 13
  Z W V U S Q L G F D A
  min(M,T)
```

```
    max(R,K)
Generated 17 of 234 possible successors

Support: 37928 Open: 69 Expressions: 12
    A D F G L Q S U W Z
    max(V,R,K)
    min(M,T)
Generated 19 of 198 possible successors

Support: 21869 Open: 87 Expressions: 11
    Z W S Q L G F D A
    sum(max(V,R,K),U)
    min(M,T)
Generated 16 of 165 possible successors

Support: 11497 Open: 102 Expressions: 10
    D F G L Q S W Z
    min(M,T,A)
    sum(max(V,R,K),U)
Generated 18 of 135 possible successors

Support: 7412 Open: 119 Expressions: 9
    Z W S Q L G F D
    sum(min(M,T,A),max(V,R,K),U)
Generated 15 of 108 possible successors

Support: 2954 Open: 133 Expressions: 8
    D F G L Q W Z
    sum(min(M,T,A),max(V,R,K),U,S)
Generated 13 of 84 possible successors

Support: 1039 Open: 145 Expressions: 7
    Z W Q G D
    sum(F,L)
    sum(min(M,T,A),max(V,R,K),U,S)
Generated 11 of 63 possible successors

Support: 962 Open: 155 Expressions: 6
    D Q W Z
    sum(G,min(M,T,A),max(V,R,K),U,S)
    sum(F,L)
Generated 10 of 45 possible successors

Support: 356 Open: 164 Expressions: 5
    Z Q D
    sum(W,F,L)
    sum(G,min(M,T,A),max(V,R,K),U,S)
Generated 10 of 30 possible successors

Support: 962 Open: 173 Expressions: 4
    D Q Z
    max(sum(W,F,L),sum(G,min(M,T,A),max(V,R,K),U,S))
Generated 9 of 18 possible successors

Support: 11 Open: 181 Expressions: 3
    Z
    sum(D,Q)
    max(sum(W,F,L),sum(G,min(M,T,A),max(V,R,K),U,S))
Generated 3 of 9 possible successors

Support: 11 Open: 183 Expressions: 2
    sum(D,Q,Z)
    max(sum(W,F,L),sum(G,min(M,T,A),max(V,R,K),U,S))
Generated 0 of 3 possible successors

Support: 0 Open: 182 Expressions: 2
```

```
   min(sum(D,Q),Z)
   max(sum(W,F,L),sum(G,min(M,T,A),max(V,R,K),U,S))
Generated 0 of 3 possible successors

Support: 0 Open: 181 Expressions: 2
   max(sum(D,Q),Z)
   max(sum(W,F,L),sum(G,min(M,T,A),max(V,R,K),U,S))
Generated 0 of 3 possible successors

Support: 11 Open: 180 Expressions: 3
   Q
   sum(D,Z)
   max(sum(W,F,L),sum(G,min(M,T,A),max(V,R,K),U,S))
Generated 3 of 9 possible successors

Support: 11 Open: 182 Expressions: 2
   sum(D,Z,Q)
   max(sum(W,F,L),sum(G,min(M,T,A),max(V,R,K),U,S))
Generated 0 of 3 possible successors

Support: 0 Open: 181 Expressions: 2
   min(sum(D,Z),Q)
   max(sum(W,F,L),sum(G,min(M,T,A),max(V,R,K),U,S))
Generated 0 of 3 possible successors

Support: 0 Open: 180 Expressions: 2
   max(sum(D,Z),Q)
   max(sum(W,F,L),sum(G,min(M,T,A),max(V,R,K),U,S))
Generated 0 of 3 possible successors

Support: 11 Open: 179 Expressions: 3
   D
   sum(Q,Z)
   max(sum(W,F,L),sum(G,min(M,T,A),max(V,R,K),U,S))
Generated 3 of 9 possible successors

Support: 11 Open: 181 Expressions: 2
   sum(Q,Z,D)
   max(sum(W,F,L),sum(G,min(M,T,A),max(V,R,K),U,S))
Generated 0 of 3 possible successors

Support: 0 Open: 180 Expressions: 2
   min(sum(Q,Z),D)
   max(sum(W,F,L),sum(G,min(M,T,A),max(V,R,K),U,S))
Generated 0 of 3 possible successors

Support: 0 Open: 179 Expressions: 2
   max(sum(Q,Z),D)
   max(sum(W,F,L),sum(G,min(M,T,A),max(V,R,K),U,S))
Generated 0 of 3 possible successors

Support: 0 Open: 178 Expressions: 3
   Z
   min(D,Q)
   max(sum(W,F,L),sum(G,min(M,T,A),max(V,R,K),U,S))
Generated 3 of 9 possible successors

Support: 0 Open: 180 Expressions: 2
   min(D,Q,Z)
   max(sum(W,F,L),sum(G,min(M,T,A),max(V,R,K),U,S))
Generated 1 of 3 possible successors

Solution: Open: 180 Expressions: 1
   max(min(D,Q,Z),sum(W,F,L),sum(G,min(M,T,A),max(V,R,K),U,S))

75.12 cpu seconds to solve
```

There are several points to notice. First, the number of potentially correct children is typically very much less than the number of candidate children. Pruning in this manner eliminates many worthless paths. Sometimes, no children are generated at all, indicating a failed path in the best-first search. Second, as the number of disconnected expressions falls, so too does the support. Third, this task structure (see 'Solution" at end of trace) includes nontrivial nestings: *MAX(SUM(),SUM())* and *SUM(MIN(),MAX())*. Finally, the search is quite efficient, making only a few mis-steps when the values of the support heuristic become low. We suspect that more data would help; one needs data examples that will include positive support for the various cases.

The BEFIT algorithm was also applied to problems p6 and p7. It found the same task structure for each, as it should have. However, whereas BEFIT took 27 steps to solve problem p5, it took 15 steps to solve problem p6 and 23 steps to solve p7. There are 15 local methods, so it must take at least 15 steps to produce a solution, so p6 was solved without any mis-steps. We suspect that the presence of interactions reduces significantly the number of combinations of local method values that could produce the total quality value, which helps rather than hurts the search efficiency. For problem p7, 23 steps is somewhat better than the 27 steps for p5. The number of data examples was 1185 for p5, 1012 for p6, and 520 for p7. It may be that fewer examples for p7 had a detrimental effect. We will be pursuing these performance issues to understand them better.

6 Summary

We have produced a new data fitting algorithm BEFIT that produces an expression composed of continuous or discontinuous primitive functions. BEFIT navigates expression space efficiently both because it can refute and prune possible expressions very effectively, and because a simple measure of evidentiary support ranks non-refuted expressions well. The data modeling problem originated from the desire to understand how coordination choices for multiple agents affects the overall quality of the solution produced by those agents. By using knowledge of the domain, it was possible to fashion a learning algorithm that has advanced the technology for multi-agent coordination.

Acknowledgments

This material is based on work supported by the National Science Foundation under Grants IRI-9711239 and IIS-9812755. Regis Vincent chose the task structures and generated the data tuples for them. David Stracuzzi and Margaret Connell provided helpful comments.

References

Decker, K. S., & Lesser, V. R. (1993). Quantitative modeling of complex environments. *International Journal of Intelligent Systems in Accounting, Finance, and Management, 2*, 215-234.

Decker, K., & Lesser, V. (1995). Designing a family of coordination algorithms. *Proceedings of the First International Conference on Multiagent Systems (ICMAS-95)* (pp. 73-80). San Francisco: AAAI Press.

Duda, R. O., & Hart, P. E. (1973). *Pattern classification and scene analysis*. New York: Wiley & Sons.

Jensen, D., Atighetchi, M., Vincent, R., & Lesser, V. (1999). Learning quantitative knowledge for multiagent coordination. *Proceedings of the Sixteenth National Conference on Artificial Intelligence* (pp. 24-31).

Press, W. H., Flannery, B. P., Teukolsky, S. A., & Vetterling, W. T. (1988). *Numerical recipies in C: The art of scientific computing*. New York: Cambridge University Press.