

Learning Hierarchical Performance Knowledge by Observation

Michael van Lent, John Laird

Artificial Intelligence Lab, University of Michigan, 1101 Beal Ave.,
Ann Arbor, MI 48109-2110

Abstract

Developing automated agents that intelligently perform complex real world tasks is time consuming and expensive. The most expensive part of developing these intelligent task performance agents involves extracting knowledge from human experts and encoding it into a form useable by automated agents. Machine learning from a sufficiently rich and focused knowledge source can significantly reduce the cost of developing intelligent performance agents by automating the knowledge acquisition and encoding process. Potential knowledge sources include instructions from human experts, experiments performed in the task environment and observation of an expert performing the task. Observation is particularly well suited to learning hierarchical performance knowledge for tasks that require realistic, human-like behavior. Our learning by observation system, called KnoMic (Knowledge Mimic), extracts knowledge from observations of an expert performing a task and generalizes this knowledge into rules that an agent can use to perform the same task. Learning performance knowledge by observation is more efficient than hand-coding the knowledge in a number of ways. Knowledge can be encoded directly from the expert without the need for a knowledge engineer to act as an intermediary. Also, the expert only needs to demonstrate the task rather than organize and communicate all the relevant information. This paper will describe the knowledge required for task performance, describe how this knowledge is learned by KnoMic, and report on our efforts to learn performance knowledge in the tactical air combat domain and the computer game Quake II.

Keywords: Supervised learning, Observation, Induction, Performance Tasks

Email address of contact author: vanlent@umich.edu

Phone number of contact author: 734-647-0969

Electronic version: Postscript

1 Introduction

Frequently the most time consuming and costly part of developing intelligent agents is the acquisition and hand coding of task performance knowledge into an appropriate knowledge format, such as production rules. The knowledge acquisition approach often used involves lengthy interviews with human experts, thousands of hours hand-programming and debugging performance knowledge and frequent demonstrations to allow the experts to verify the final agent behavior. The experts are required to organize and communicate their knowledge to the programmers and the programmers are required to learn the details of the task, effectively becoming experts themselves.

One approach to improving the efficiency of knowledge acquisition is to develop tools that the expert can use to encode knowledge. This reduces the need for a knowledge engineer to act as an intermediary between the expert and the encoded performance knowledge. However, the expert is still required to organize and describe the necessary knowledge and additionally must learn to use the knowledge acquisition tool. A second approach is to develop an unsupervised learning system that learns to perform the task without relying on an expert, perhaps through experimentation. Unfortunately learning to perform a task solely through unsupervised experiments in the task environment is generally more costly than the original knowledge acquisition. Furthermore, the resulting performance agents, while successful, may not perform the tasks as humans do which limits their usefulness in training tasks. A third approach is to develop a supervised learning system that uses an expert to provide a more focused source of knowledge, such as instruction or observation. Supervised learning differs from the knowledge acquisition tool approach described above in that the expert interacts with the system at the knowledge level rather than the implementation level. Supervised learning doesn't require the expert to learn a new tool or programming language and doesn't require a programmer to become an expert in the task. Furthermore the resulting performance knowledge is based on an expert's knowledge and allows different experts to generate different styles of performance. Finally, a well-chosen form of expert input can minimize the need for the expert to organize and communicate his or her knowledge.

A supervised learning system that takes observations of an expert performing a task and generates generalized task performance knowledge directly from these observations fulfills all of the requirements discussed above. Learning by observation requires experts to do nothing other than perform the task at which they are expert. In addition, the programmer isn't involved in encoding the knowledge and therefore doesn't need to be taught any task knowledge. An agent's performance, based on the encoded knowledge, will closely correspond to the expert's behavior and, if multiple experts are available, learning by observation can capture variations in their knowledge to provide multiple styles of performance.

The KnoMic (Knowledge Mimic) system learns the knowledge necessary for an intelligent agent to perform a task from observations of an expert performing the task and expert annotations indicating when goals have been achieved. Like design tasks and

analysis tasks, performance tasks represents a class of problems for machine learning. In a performance task the intelligent agent performs actions in a constantly changing environment to achieve or maintain a collection of goals. The agent constantly receives sensor inputs updating the state of the environment and uses the performance knowledge to select the current goals and the actions that will achieve those goals. For the research presented here performance knowledge is represented as production rules that implement operators, generate and maintain internal state features and test for goal achievement.

The observations used by KnoMic consist of a sequence of time steps with the sensor inputs from the environment, any actions taken by the expert, and annotations indicating goal achievement, all of which are recorded every time step. The system can learn online, using these observations as they are generated, or the observations can be stored and used to learn off line. The sensor inputs and actions are described in the description language for the domain. A programmer develops this domain description language, which should attempt to mimic the concepts used by the expert as closely as possible. KnoMic is designed to work in complex domains with real time behavior requirements, external actions, non-determinism and other complications. Thus the time steps can potentially be very short with lots of activity in some time steps and no activity in others.

In the next section two supervised learning systems which use observation as the primary knowledge source are described, behavioral cloning [1, 6] and the OBSERVER system [8, 9]. Section three describes the format of the performance knowledge generated. The KnoMic system, described in the fourth section, combines many of the strengths of the systems described in section two while overcoming some of their weaknesses. The fifth section provides examples and evaluation of the knowledge generated by KnoMic.

2 Related Research

The KnoMic system described in this paper is preceded by two significant research efforts in the area of learning by observation. Behavioral cloning uses pure induction from observation traces to build decision trees that decide which actions to take based on the sensor input. While behavioral cloning has been successfully applied to a complex, real world domain, the representation of the performance knowledge learned by behavioral cloning is limited. The OBSERVER system uses a version spaces type approach to learn STRIPS style operators [2]. OBSERVER learns many aspects of performance knowledge, including knowledge about the effects of operators, which is required for planning. However, OBSERVER makes assumptions about the domain that limits its effectiveness in complex, dynamic domains. KnoMic can be viewed as an attempt to combine the powerful knowledge representation of OBSERVER with the techniques used by behavioral cloning to learn in a complex domain.

2.1 Behavioral Cloning

Sammut, Hurst, Kedzier and Michie have used behavioral cloning to learn the knowledge necessary to fly a Cessna along a specific flight plan within the Silicon Graphics flight simulator. A human expert, whose behavior is to be cloned, flies a strictly defined flight plan 30 times. Recording the sensor inputs and the value of each control approximately 20 times a second creates an observation log of each flight. These observation logs form a large set of training examples with the sensor inputs as the attributes and the controls as the class values. Behavioral cloning uses this set of training examples to induce decision trees that classify the appropriate control values (or actions) based on the current sensor inputs. These decision trees can then be used to mimic the expert's behavior and fly the plane by setting each control to the value specified by applying the current sensor inputs to the decision tree.

Rather than learn a single decision tree for each control, the flight plan is divided into seven stages and a separate decision tree is learned for each stage. Since each stage has a different goal, such as takeoff and fly to an altitude of 2,000 feet, the expert's responses to the same sensor inputs differ according to the goal of the current stage and a separate decision tree is needed for each stage. Thus, splitting the observation logs into stages adds a goal directed aspect to behavioral cloning. Similarly, a single set of decision trees couldn't be learned from observations of multiple experts as different experts react to inputs in different ways to achieve the same goals.

The behavioral cloning system is able to learn 28 decision trees that effectively control the four controls over seven stages of the flight plan. Using these decision trees as performance knowledge an intelligent agent can take off, follow the defined flight plan and land in a manner very similar to the expert being cloned. One of the most impressive aspects of behavioral cloning is its effectiveness in a complex domain. The flight simulator domain is non-deterministic, includes a large set of complex sensors, exhibits standard and homeostatic goals, the observations contain noise and the nature of the domain requires rapid reactions to input. By using a number of small decision trees, behavior cloning splits the knowledge into many, rapidly processed rules which can be applied sufficiently quickly.

The major weakness of behavioral cloning is the limited usefulness of decision trees as a form of generalized performance knowledge. The decision trees learned are effective for a single flight plan but need to be relearned if the flight plan or the aircraft dynamics change. Because decision trees can't easily handle symbolic inputs, such as flight plans, it is impossible to learn generally useful decision trees that apply to multiple flight paths. Another weakness of decision trees is that they only encode reactive knowledge useful for execution. To be used in planning, task knowledge must be predictive allowing the outcome of potential plans to be evaluated. Decision trees contain no information about the effects of their actions and therefore are not appropriate for planning.

2.2 OBSERVER

The OBSERVER system learns STRIPS style operators using a learning method similar to version spaces. Xuemei Wang developed OBSERVER and applied it to a process-planning domain, in which the task is to generate a plan to produce machine parts meeting a set of specifications. For each operator OBSERVER creates and refines a most specific set of operator preconditions $S(Op)$, a most general set of operator preconditions $G(Op)$ and a set of operator effects $E(Op)$. OBSERVER initially learns $S(Op)$ and $E(Op)$ in a learning by observation phase. Then a second, unsupervised learning phase is used to improve $S(Op)$ and $E(Op)$ and to learn $G(Op)$. When $S(Op)$ and $G(Op)$ match then the operator's preconditions are assumed to be correct.

An advantage of the OBSERVER system over behavioral cloning is that the performance knowledge learned by OBSERVER uses a less limited form of representation. The STRIPS style operators, learned by OBSERVER, include complexities such as negated preconditions and conditional actions. While the operators are somewhat more difficult to learn, they represent most aspects of performance knowledge effectively and can deal with symbolic inputs. Additionally, the operator effects, which are learned as part of each operator, predict how applying the operator will change the current state. These effects provide the predictive knowledge necessary to evaluate the outcome of proposed plans.

The OBSERVER system is described in the context of a design task that isn't as complex as the performance tasks demonstrated by behavioral cloning. One weakness of the STRIPS style operators is that they represent large, complex actions more easily than the highly reactive and fine-grained actions needed for the flight simulation task. Since the design task contains no noise or non-determinism, OBSERVER can and does generate knowledge based on a single observation. In more complex domains, this will result in incorrect or over specific operators. OBSERVER also doesn't use goal knowledge in its learning process. Goal knowledge could be very useful in learning operator effects as the goal provides a set of features the operators are trying to achieve.

3 Performance Knowledge Representation

The KnoMic system uses the Soar architecture [4] to execute the learned performance knowledge. Accordingly, the knowledge representation learned by KnoMic is a specialized form of Soar's production rule format. The rules implement a hierarchy of operators with higher level operators having multiple sub-operators representing steps in the completion of the high level operator. Each operator has a set of pre-conditions and a set of goal conditions that it achieves. An operator is selected when its pre-conditions are matched. Once selected, a series of conditional operator application rules and/or sub-operators perform actions to achieve the operator's goal conditions. When the goal conditions are met an operator specific goal-achieved feature is added to the system's internal state. Once an operator's goal feature appears on the state the operator is

unselected allowing another operator to be selected at the same level of the hierarchy. Non-persistent goal features are removed from the state when the goal conditions are no longer matched. Persistent goal features remain on the state independent of the status of the goal conditions until removed as the action of a later operator.

Operators can be classified as maintain operators, one-time operators or repeatable operators depending on the persistence of their goal features. Operators with non-persistent goal features are called maintain operators because whenever their goal conditions become untrue the goal feature is removed and the operator is reselected to re-achieve the goal conditions thus maintaining the goal conditions as true. While flying to a specific waypoint, a maintain operator might set the plane's heading to point in the waypoint's direction. Once the plane is flying at the waypoint the operator's goal is achieved and the operator is unselected. If the plane's heading deviates from that direction the goal achieved feature would be retracted and the operator would be reselected to re-achieve the goal. Thus the operator maintains the heading pointing towards the waypoint. One-time operators have persistent goal features that are never removed. Once these operators achieve their goal the goal achieved feature remains on the state indefinitely and the operator will never be reselected. Initialization operators that fire once at the beginning of a task are examples of one-time operators. Repeatable operators also have persistent goal features but, unlike one-time operators, these persistent features can be removed and the operator can be repeated. Examples of repeatable operators will be presented later in the context of the air combat domain.

Three types of rules implement operators. Operator selection rules test state features (internal and external) and select the next operator to be executed. Operator application rules generate actions, based on the current operator and other state feature tests, to be performed in the external environment. Goal feature generation rules create internal goal features representing that the operator's goal has been achieved. KnoMic learns each of these three types of production rules.

As with STRIPS operators, KnoMic operators contain operator preconditions and operator effects. Operator selection rules define an operator's pre-conditions. Common classes of pre-conditions include relevance conditions and goal achieved features from previous and current operators. Relevance conditions are the traditional state feature tests that assure that an operator will only be selected when its actions are relevant. Operator goal features test that the goal achieved feature of one or more other operators are present. These conditions assure that the later operators in a series of operators aren't selected until the earlier operators are finished. The current operator's goal feature condition is a negated test of the current operator's goal achieved feature. Once the operator has achieved its goal and the goal feature has been generated, the conditions of the operator will no longer match (due to the negated goal feature test) and the operator will be automatically be unselected. Unlike STRIPS, KnoMic operators may be applied over a series of time steps, making it possible for a higher level operator to remain in place while its sub-operators are selected. Each operator remains selected as long as the

conditions of its selection rule are matched. As the operator selection rules are generated it is ensured that they are mutually exclusive to avoid operator conflicts.

Operator application rules test internal and external state features and each issues a single action/value command to the environment. For example, the action/value pair (heading 90) would cause the plane being controlled to turn to a heading of 90 degrees from due north. Each application rule tests the name of a current operator, which associates each rule with an operator that must be selected for that application rule to fire. If multiple operators issue the same action command a separate application rule is required for each operator. This is similar to behavioral cloning's use of multiple decision trees and allows different operators to select the values of their actions differently according to their specific goal. In some cases the action's value is taken from one of the features tested in the rule's conditions. This type of application rule is said to "pass through" the value of a sensor input to an action command.

Goal feature generation rules test a current operator's goal conditions and create either persistent or non-persistent goal features. The non-persistent rules add a goal achieved feature to the state when the goal conditions are satisfied and remove the feature if inputs change such that the goal conditions are no longer satisfied. Persistent rules add a goal achieved feature to the state which remains in place even after the goal conditions are no longer matched. Persistent goal features can only be removed as the action of a later operator. Goal achieved features frequently represent relationships between sensor inputs that are important in the domain. For example, a takeoff operator's goal conditions might test that the plane's current altitude is equal to the flight plan's cruising altitude. The goal achieved feature of this operator could then be tested by other rules rather than relearning the relationship between altitude and cruising altitude each time. Also, goal features are likely to represent concepts familiar to the expert making the learned rules easier to understand.

KnoMic combines a form of knowledge representation similar to OBSERVER's, but more reactive, with the ability to learn effectively in complex performance tasks found in behavioral cloning. The specialized form of Soar operators KnoMic uses is somewhat similar to the STRIPS style operators used in OBSERVER. These operators combine the performance knowledge necessary for an intelligent agent to perform a task in a complex, dynamic environment with predictive knowledge that could be used to plan new approaches to task performance. One major advantage of using Soar operators is the potential to use KnoMic in conjunction with other learning systems which generate Soar operators including INSTRUCTO-Soar [3] which learns from expert instruction and IMPROV [5] which learns through unsupervised experimentation.

4 KnoMic

The KnoMic system makes use of many of the techniques used by behavioral cloning to learn effectively in the complex flight simulator environment. The operators are expressed as separate selection and application rules, the matching of which can be

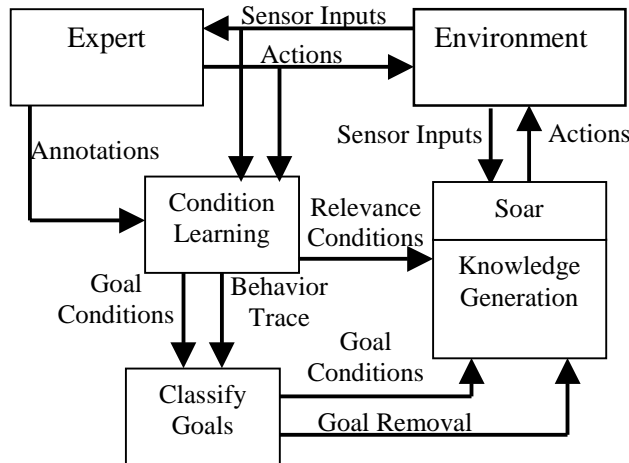


Figure 1: The KnoMic system including the Expert/Environment loop, the Condition Learning, Goal Classification and Knowledge Generation components and the Soar Architecture.

efficiently calculated for quick reactivity. KnoMic learns from multiple observations allowing incorrect rules to be corrected or even replaced if necessary. Additionally, KnoMic makes extensive use of goal knowledge to focus the learning process. Unlike behavioral cloning's fixed goals, KnoMic learns goals and subgoals dynamically based on annotations provided by the expert as part of the observation log.

The KnoMic system consists of three components that observe the interactions between the expert and the environment (or simulator) and communicate among themselves to generate performance knowledge (see Figure 1). The Soar architecture then acts as the brains of the intelligent agent using the learned knowledge to perform the task. The condition learning component compares the expert system's behavior to the current performance knowledge and generalizes operator and goal conditions in situations where the performance knowledge doesn't match the expert's behavior. The goal classification component uses the learned goal conditions and the expert's behavior to classify each goal feature as persistent or non-persistent and to build any necessary goal removal operators. The conditions and goal classifications are passed to the knowledge generation component, which simply formats the conditions into Soar production rules. An intelligent agent can then use the Soar architecture to perform the task using this new performance knowledge. In this section KnoMic is described in the context of a flight simulation domain similar to the domains used in behavioral cloning and TacAir-Soar, a 5,200 rule, hand coded intelligent agent used in air combat training exercises [7].

4.1 Expert-Environment Loop

An expert's interaction with the environment while performing a task can be viewed as a communication loop. The environment sends information to the expert in the form

of instruments (altitude dial, air speed indication...) displayed on the computer screen, the view from the cockpit window and perhaps sounds from a radio. The expert reacts to this information by sending actions to the environment in the form of changes to aircraft controls (flight stick, throttle...) or keypresses and mouse movements that represent aircraft controls in a simulation. The environment reacts to these actions by modifying the information sent to the expert to reflect the changes these actions make in the current situation of the plane in the environment.

For KnoMic, the information passed between the expert and the environment is translated into a symbolic form. Thus, where the expert perceives a needle pointing to 320 on the airspeed indicator, KnoMic perceives a `vehicle.speed` sensor feature with a value of 320. For each relevant sensor displayed for the expert, KnoMic receives a corresponding sensor input feature. Similarly, for each action the expert can take, KnoMic is provided with a symbolic action/value command such as `air_speed=300`. Creating the symbolic sensor inputs and actions requires a programmer but the programmer only needs a limited amount of task knowledge to create the interface.

As the expert performs the task, recording any changes in the sensor inputs and any action commands issued by the expert creates a trace of the expert's behavior. For a rapidly changing environment, such as a flight simulator, the inputs and actions need to be recorded multiple times a second. As mentioned previously, the expert annotates the observation log with comments describing the operators and sub-operators currently active from an operator hierarchy. For example, an expert might report "Top level operator is execute mission; sub-operator of execute mission is fly racetrack pattern; sub-operator of fly racetrack pattern is fly to waypoint." Generally, annotating the observation trace, which can either be done as the task is being performed or afterwards, is easier if the expert has developed the operator hierarchy ahead of time. All that the expert needs to provide in the annotations are markers denoting the selection of operators. KnoMic needs no information about the operators other than their hierarchy and when they are selected.

4.2 Condition Learning

For each operator, the condition learning component must select the sensor input/value tests that must be true for the operator to be appropriate in the current situation and applicable to the current goals. To successfully perform realistic, complex tasks operators must be able to test for the absence of input/value pairs (negated tests), inputs with ranges of values and pairs of inputs with equal values. The condition learning component must find the correct selection and goal conditions for each operator while excluding irrelevant conditions. A simple specific-to-general learning algorithm is used to learn both selection and goal conditions, which initially creates a most specific set of conditions and generalizes by removing irrelevant conditions based on further observation traces. The expert's annotations allow KnoMic to compare the conditions under which the expert selects an operator to the performance knowledge's operator

proposal conditions. If the performance knowledge doesn't have any conditions for the operator yet then an initial set of maximally specific conditions (all current sensor input values) are assigned to the operator. If the expert selects an operator in a situation where the performance knowledge's operator proposal conditions aren't matched, then the proposal conditions are generalized to match in the current situation. If the performance knowledge selects an operator when the expert didn't select the same operator, then the proposal conditions have been overgeneralized. In most cases the cause of the overgeneralization is an operator with disjunctive proposal conditions. Although KnoMic can't currently learn disjunctive operator proposal conditions, a plan for correcting overgeneralization by splitting the proposal conditions into two sets of disjunctive conditions is being implemented.

Sensor inputs can also be used to represent static domain constants such as flight plan parameters (`cruising_altitude = 2,000` feet). KnoMic will detect relationships between dynamic sensor inputs and static sensor inputs (called parameters) that use the same units and create operator proposal conditions representing the relationship between the two values. Thus, if one of the proposal conditions for the fly-towards-destination operator is that the plane's altitude is 2,000 feet and a parameter input `cruising_altitude` has a value of 2,000 feet, then KnoMic will specify `altitude = cruising_altitude` as a selection condition of fly-towards-destination and remove the `altitude = 2,000` condition. Then, rather than relearning the performance knowledge to change the flight plan's cruising altitude, simply changing the `cruising_altitude` parameter will allow the existing performance knowledge to fly at the new altitude.

An operator's goal conditions represent the input/value conditions that the operator's actions and sub-operators will achieve if that operator is selected. Goal conditions are learned in a similar fashion to operator proposal conditions. The first time the expert's annotations indicate that an operator has been unselected, KnoMic collects the sensor inputs that have recently changed and initializes the operator's goal conditions to be these changes. When subsequent observations of the operator being unselected are found, the goal conditions are generalized to include only those sensor input changes that always immediately proceed the operator being unselected. Thus, if the plane's altitude reaches 2,000 feet immediately before the Takeoff operator is unselected in each observation, one of the goal conditions for Takeoff will be that the altitude is 2,000 feet. If the `cruising_altitude` parameter mentioned above is present, then the condition will test that the current altitude equals the cruising altitude. Goal conditions can also test that parameters just became less than or greater than sensor inputs. Perhaps in the first observation, by coincidence, the wind speed sensor changes to 8 knots just as Takeoff is unselected. In this case `wind_speed = 8` will also be a goal condition of Takeoff after the first observation. It is unlikely that the wind speed will change at just the right time twice and this extra condition will therefore be removed when the goal of Takeoff is generalized based on the second observation of the expert. Some sensor inputs might change rapidly enough that they will always have recently changed when an operator is unselected. KnoMic detects these frequently changing inputs and doesn't include them

as goal conditions unless they are part of a condition that tests the sensor's relationship with a parameter.

4.3 Goal Classification

The goal classification component needs to learn the conditions under which each operator's goal feature should be removed from internal memory. In simple cases the goal features are never removed or removed as soon as the goal conditions are no longer true. In more complex cases a new operator, with new selection conditions, needs to be learned to remove the goal feature. Goal features are classified by examining how the expert reacts when an operator's goal conditions become true or change from true to false. If the expert immediately reselects an operator every time that operator's goal conditions are no longer true, then the expert is applying the operator to maintain the truth of the goal conditions. In this case the operator is classified as a maintain operator and its goal feature is marked as non-persistent. If, once the expert has achieved an operator's goal, the sensor inputs change to make the goal conditions untrue but the operator isn't reselected, then the operator either only needs to be executed once (a one time operator) or is only reselected under certain conditions (a repeatable operator). In either case the goal feature for that operator is marked as non-persistent. If the operator's is reselected at a later time, the operator is classified as repeatable, and the situations in which the operator is reselected are used to learn the proposal conditions of a goal feature removal operator.

4.4 Soar Architecture

The Soar architecture uses the sensor inputs and the performance knowledge to generate actions that perform the task. At a simple level, the Soar architecture compares the performance knowledge rules to the current symbolic sensor inputs. At the top level of the hierarchy, the operator with matching selection conditions is selected. Once an operator is selected, its application rules and/or sub-operators are selected and applied until the goal conditions are achieved. Once the goal is achieved, a goal feature is generated and the operator is unselected so a new operator can be selected and applied. A full description of the Soar architecture [4] is not necessary to understand how the performance knowledge learned by KnoMic is used. Once the performance knowledge is correctly learned, an intelligent agent, using Soar, can replace the human expert and perform the task in the style of the human.

5 Experimental Results

The KnoMic system, as described above, is fully implemented and has been connected to the ModSAF simulator used in the TacAir-Soar project and to the computer game Quake II. With the ModSAF simulator, eighty seven values are converted to

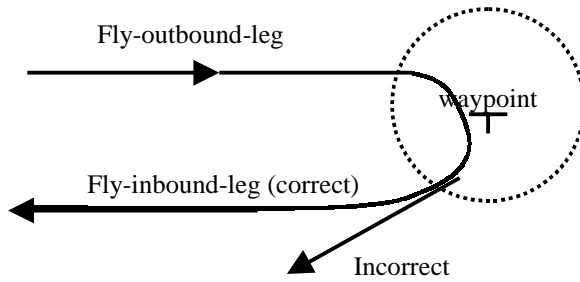


Figure 2: The agent flies the outbound leg towards the waypoint and then turns to fly the inbound leg (searching for inbound enemies) once the waypoint range is less than 3000 meters.

symbolic sensor inputs including the plane's position and velocity, radar information, mission parameters, waypoints and a waypoint computer. Twenty output actions are available including flight controls, radar controls weapon controls and various miscellaneous controls. The interface with Quake II is somewhat less complex implementing 7 symbolic sensor inputs and 5 output actions.

The sensor inputs and output actions used by KnoMic were originally developed as part of the TacAir-Soar project and have been incorporated unchanged. A Human Interface Panel has been implemented and connected to the system so that KnoMic can learn task knowledge based on observations of a human expert controlling a plane in ModSAF. KnoMic can also learn by observing hand coded intelligent agents as they control the plane or play Quake II. The results presented in this section are based on experiments with KnoMic learning from the hand coded TacAir-Soar agents. This is particularly interesting as the rules learned by KnoMic can be compared to the hand coded rules. Performance knowledge has also been successfully learned in Quake II although those experiments are not presented here.

Human experts will be less precise in their performance of the task, which will cause more variations between the observations. In some cases this additional noise may help the system generalize out irrelevant conditions but in general, learning from observations of human experts is more difficult. Variations in human behavior such as delays between receiving input and taking actions and performance errors will introduce a large amount of noise into the observation traces. However, early experiments with the racetrack portion of the air combat task have shown that KnoMic can learn correct, functional rules based on observations of humans.

5.1 The Air Combat Task

The knowledge learned in the experiments with the ModSAF simulator attempts to perform the air combat task based on observations of the micro TacAir-Soar agent. The micro TacAir-Soar agent is a reduced version of the full TacAir-Soar agent and is used in

Conditions Learned per Observation

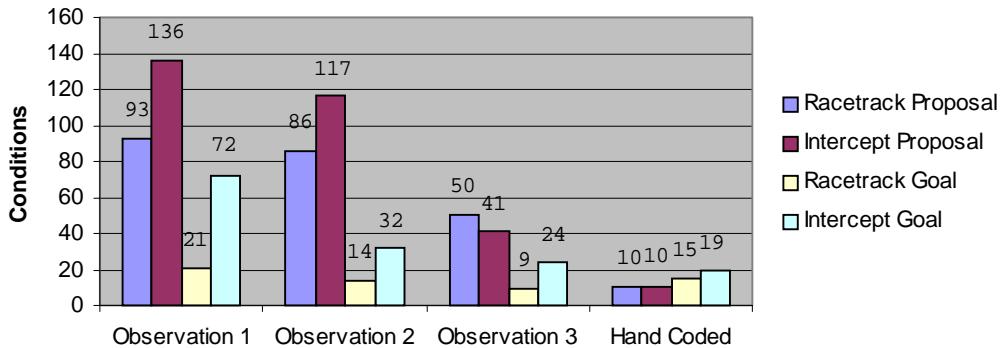


Figure 3: The results of KnoMic’s learning over a series of three observations of the TacAir-Soar agent performing the racetrack task. The fourth data set represents the hand-coded

research and teaching situations. To perform the air combat task, TacAir-Soar takes off and flies to a specified waypoint. Once the waypoint is reached the plane flies a patrol pattern, called a racetrack, until an enemy plane is detected on radar. When the enemy plane is detected the agent engages the enemy plane, selects and fires missiles and, once the threat is eliminated, returns to flying the racetrack pattern. The hand coded micro TacAir-Soar agent uses approximately 175 production rules to perform the air combat task.

The racetrack pattern involves flying at a specific heading (the racetrack heading parameter) until a specific distance (the racetrack distance parameter) away from the specified waypoint is reached and then flying back to the waypoint. The racetrack waypoint, heading and distance are specified as mission parameters (static sensor inputs). The operator annotations, which are based on the hand-coded operator hierarchy, divide the task into five top level operators, init-agent, create-agent, wait-to-start-vehicle, init-plane and execute-mission. The execute-mission operator has two sub-operators, racetrack and intercept. The racetrack operator has three sub-operators, fly-to-racetrack, fly-inbound-leg and fly-outbound-leg while under intercept are 11 operators in a four level hierarchy.

One aspect of the racetrack pattern demonstrates well why persistent features are necessary. Consider the case where the KnoMic agent is flying towards the waypoint and must change heading to the racetrack direction parameter once the plane is close enough to the waypoint (see Figure 2). What should happen in this situation is that the fly-outbound-leg operator achieves it’s goal (i.e. the waypoint range is less than the waypoint achieved range parameter), generates a goal feature and terminates. Fly-outbound-leg’s goal feature is a pre-condition of fly-inbound-leg so fly-inbound-leg is selected and an

application rule fires immediately to turn the plane to the racetrack direction parameter. If fly-outbound-leg's goal feature is non-persistent, then once the waypoint range again becomes greater than the achieved parameter the goal feature is removed and fly-inbound-leg is unselected since its pre-conditions are no longer satisfied. The agent effectively forgets it is flying the inbound leg and never completes its turn to fly in the racetrack direction. However, if the goal feature is made persistent then it is not removed when the waypoint range becomes greater than the achieved parameter and the turn is completed correctly. Note that fly-outbound-leg's persistent goal feature will have to be removed by a goal feature removal operator before fly-outbound-leg can be reselected.

5.2 Performance knowledge Learned for Racetrack

Figure 3 displays the results of KnoMic's learning over three observations of the racetrack portion of the air combat task. Generally after three or four observations KnoMic has generated correct rules for flying the racetrack pattern and has three quarters of the intercept part of the task correct. Since KnoMic's learning algorithm is specific-to-general, the rules learned are initially over specific (too many conditions) and as subsequent observations are made the rules are generalized by removing irrelevant conditions. After the first observation, conditions have been learned for the 91 rules that implement the 22 operators required for the air combat task. Altogether, the rules have 229 operator selection conditions and 93 goal conditions. After the second observation 26 selection conditions and 20 goal conditions have been removed through generalization. The starting conditions for the third observation are slightly different than the first two allowing for greater generalization and 112 selection conditions and 13 goal conditions are removed. After three observations the knowledge for performing the racetrack portion of the task is functional and the knowledge for the intercept portion is about 75% working. While the performance knowledge is functional, it does contain a number of extraneous conditions which are not present in the hand-coded productions. Almost all of these productions are unnecessary but not incorrect in that they will never cause the learned agent to behave differently than the hand-coded agent. Many of the rules learned by KnoMic bear a striking similarity to the hand-coded rules in the TacAir-Soar system. Approximately 75 percent of the rules learned by KnoMic have recognizable hand-coded counterparts. The similarity between the performance knowledge from KnoMic and hand-coded knowledge suggests that programmers and experts will easily understand knowledge generated by KnoMic if it ever needs to be examined.

5.3 Efficiency of Learning by Observation vs. Hand-coding

Although the performance knowledge learned for the racetrack portion and most of the intercept portion of the task is correct and functional, KnoMic is only useful if it generates the knowledge more efficiently than a programmer does. To test this we

Task Knowledge Generation

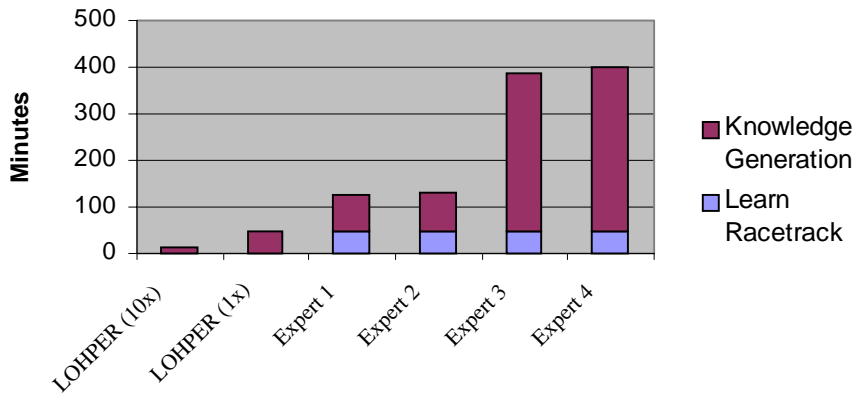


Figure 4: Time KnoMic and four experts spent generating the performance knowledge necessary for the racetrack task. The 10x KnoMic learned from observations run at 10 times

compared the time required for KnoMic to generate the performance knowledge for the racetrack portion of the task to the time taken by each of four programmers. The ModSAF simulator can be run up to ten times faster than real time, which decreases the time KnoMic spends observing and doesn't affect the knowledge generated. As shown in figure 4, when run at ten times real time KnoMic requires 3 minutes to start the simulator, 2 minutes for each observation, 2 minutes to generate the task knowledge and 2 minutes to demonstrate the final behavior for a total time of 15 minutes. If run at real time this increases to approximately 50 minutes. By comparison, the fastest human programmer required 50 minutes to learn the racetrack task and 75 minutes to program the performance knowledge. The slowest human had programmed half the required performance knowledge after 400 minutes. This experiment demonstrates that KnoMic is likely to generate performance knowledge more efficiently than that generated by human programmers.

6 Future Research

The current focus of research with the KnoMic system is to correct and expand learning algorithm so that the entire air combat task is correctly learned. A number of the operators in the intercept portion of the task set internal timers and are unselected when the timers expire. Adding timer sensors and actions to the observation trace allows KnoMic to learn these operators but it is unclear how KnoMic might detect equivalent delays by a human expert. While KnoMic can correctly learn conjunctive selection

conditions, conjunctive goal conditions can't currently be learned. Due to the bias towards using recently changed inputs as goal conditions KnoMic only learns the last condition in the conjunction to become true. Possible solutions include relaxing the bias and including a small amount of domain knowledge that suggests combinations of inputs that are likely to be tested together.

To date KnoMic has been only applied to observational traces taken from human experts, as opposed to using hand-coded intelligent agents as experts. Variations in human behavior such as delays between receiving input and taking actions and performance errors will probably introduce a large amount of noise into the observation traces. Although the few experiments that have been performed are promising, it is likely that KnoMic will need to be made much more noise tolerant before it can learn from human experts. One possibility is to integrate other systems that use different supervised knowledge sources such as instruction or unsupervised learning like experimentation.

Acknowledgement

This research was supported under contract N61339-97-K-008 from the Defense Advanced Research Projects Agency (ARPA).

References

- [1] Bain, M. and Sammut, C (1995). A framework for behavioral cloning. in S. Muggleton, K. Furakawa, and D. Michie (Eds.), *Machine Intelligence 15*. Oxford University Press.
- [2] Fikes, R.E. and Nilsson, N.J. (1971). STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3,4):189-208.
- [3] Huffman, S.B. *Instructable autonomous agents*. PhD thesis, University of Michigan, Dept. of Electrical Engineering and Computer Science, 1994.
- [4] Laird, J. E., Newell, A. and Rosenbloom, P.S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence* 33:1-64.
- [5] Pearson, D.J. *Learning Procedural Planning Knowledge in Complex Environments*. PhD thesis, University of Michigan, Dept. of Electrical Engineering and Computer Science, 1996.
- [6] Sammut, C., Hurst, S., Kedzier, D. and Michie, D. (1992). Learning to Fly. in D. Sleeman (Ed.) *Proceedings of the Ninth International Conference on Machine Learning*, Aberdeen: Morgan Kaufmann, 385-393.

[7] Tambe, M., Johnson, W. L., Jones R. M., Koss, F., Laird, J. E., Rosenbloom, P. S. and Schwamb, K. (1995). Intelligent agents for interactive simulation environments. *AI Magazine* 16(1):15-39.

[8] Wang, X. (1996). Learning Planning Operators by Observation and Practice. PhD thesis, Carnegie Mellon University, Computer Science Dept., 1996.

[9] Wang, X. (1995). Learning by Observation and Practice: An Incremental Approach for Planning Operator Acquisition. in Proceedings of the 12th International Conference on Machine Learning.