Learning Cooperative Procedures

Andrew Garland and **Richard Alterman**
Computer Science Department
Brandeis University
Waltham, MA 02254
{aeg, alterman}@cs.brandeis.edu

Abstract

Learning in a dynamic, uncertain, multi-agent setting is a challenging task. In this work, we demonstrate techniques to convert noisy run-time activity into procedures useful for future problem-solving activities. The cooperative procedures created by this conversion process are stored in an agent resource called *collective memory*. We show how this memory enables agents to learn cooperative procedures beyond the scope of their first-principles planner. Finally, we give experimental results demonstrating that collective memory improves agent performance.

1 Introduction

Early models of procedural learning assumed actors were isolated, model-based thinkers. More recently, learning techniques have become more sophisticated as this assumption has been replaced with less restrictive ones. To date, however, there has been no thorough investigation of multiple, heterogeneous, situated agents who learn from the pragmatics of their domain rather than from a model. In this paper, we develop techniques that allow agents to improve their performance in a dynamic environment by using past run-time behavior to learn procedures to better coordinate their actions. These techniques are based upon a structure called *collective memory*.

This paper begins with general overviews of collective memory and MOVERS-WORLD, our test-bed domain. The heart of the paper is an exposition on how cooperative procedures are added to collective memory. The techniques which agents use to learn cooperative procedures from their run-time behavior are outlined and applied to a simple example. Reasoning from execution traces is a key aspect of the system because it enables agents to learn plans beyond the scope of the scratch planner. Finally, results are presented that clearly show the effectiveness of collective memory in improving community performance.

2 Overview of Collective Memory

Humans provide a natural model of pragmatic agents situated in a multi-agent world. (Cole & Engeström 1993) argues that the development of distributed cooperative behavior in people is shaped by the accumulated cultural-historical knowledge of the community. Our learning techniques are motivated by this argument and use a structure called collective memory to store the breadth of knowledge the community acquires through interacting with each other and the world during the course of solving sequences of distinct problems. In this paper, we assume a distributed (rather than centralized) memory model wherein each agent maintains only a memory of her own interactions; jointly, these memories represent the collective memory of the community.

The cornerstone of collective memory is a cooperative procedures case-base that augments the agents' first-order planner. In other words, this work extends single agent second-order planners (Alterman 1988; Hammond 1990; Veloso & Carbonell 1993) into multi-agent domains. The procedures stored in collective memory will be effective in future problem-solving episodes if they capture regularities in their domain of activity.

1. Give the community a set of goals to achieve.
2. Allocate the goals among the agents.
3. Until all agents have achieved their goals:
   (a) Active agents use collective memory to create or adapt a plan.
   (b) Agents with plans attempt an operator.
4. Add cooperative procedures to collective memory.

Figure 1: One cycle of community activity.

We consider the task environment for collective memory to be problem-solving by a community of multiple adaptive agents with differing roles and abilities who, initially, have very limited knowledge about each other and their domain. The activity of the community of agents is shown in Figure 1. In this paper, the phrases "solving a problem" and "problem-solving episode" both refer to a single pass through these four

steps, i.e. problem-solving is defined in terms of successfully executing actions (operators), not constructing plans. The performance of the community can be measured over successive problem-solving episodes in order to quantify the impact of collective memory. One measure of performance that we track is the number of times the agents must loop through step 3, which we call 'rounds.'

The focus of this paper is on the conversion of run-time activity into cooperative procedures, which takes place in step 4 above and is detailed fully in Section 4. The agents can use these procedures in lieu of plans generated from scratch during step 3a. All agents presently work on the entire set of community goals, but future research will investigate the benefits of using collective memory to more effectively allocate the goals in step 2.

3 MOVERS-WORLD

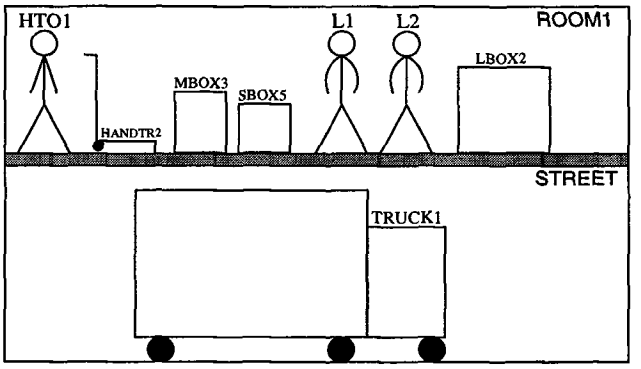

Figure 2: Sample MOVERS-WORLD problem.

MOVERS-WORLD has several agents who are differentiated by their types: some are "lifters" and some are "hand-truck operators." A simple rendering of MOVERS-WORLD is given in Figure 2. Agents of the same type may have differing capabilities due to attributes such as strength. There are type-specific operators, about which only agents of that type have knowledge (such as LOAD-TOGETHER for lifters and STAND-HANDTR for hand-truck operators), and there are general purpose operators that all agents can perform (such as MOVE and SIGN). The agents in MOVERS-WORLD do not engage in any communication at planning time. Rather, they plan independently, act independently, and only communicate when necessary to establish cooperation or to maintain coordination.

It is exceedingly unlikely that agents will find efficient solutions without learning in this domain, since there is no mandated cooperation or coordination strategy. The agents have common top-level goals, but they all have their own point of view on how best to proceed. Through the use of collective memory, agents develop a "common viewpoint" on how best to achieve a solution. That is, an agent remembers successful patterns of cooperation in which she was involved, and can use them as a basis for future interactions. In novel situations, where no such common viewpoint has been established, there is no mechanism to force the agents to do the most sensible thing; in other words, agents might refuse to assist when they should, or agree to help when they should not.

Communication in MOVERS-WORLD

MOVERS-WORLD, and the techniques used to learn cooperative procedures by MOVERS-WORLD agents, is shaped by the fact that communication is the central mechanism for cooperation and coordination. The nature of communication in MOVERS-WORLD is quite different from standard distributed AI planner systems such as (Corkill 1979) because communication occurs at run-time rather than as part of the planning process. This is a consequence of our belief that communication takes roughly the same order of time as do primitive actions. Communication between agents in our system is similar to a telephone conversation. Agents do not have to be in the same location to engage in communication. Once a connection is established, communication is limited to a small number of request and response frames. Agents do not use communication to develop any sort of shared structure, such as a SharedPlan (Grosz & Kraus 1998).

Two agents are said to 'cooperate' if they act or work together to achieve some common purpose and they are 'coordinated' when their individual actions are appropriately ordered to support cooperative activity. In our framework, communication is the only mechanism whereby agents can determine if they are coordinated. In other words, there are no global structures (such as blackboards) for agents to use to determine if they happen to be working on the same goal and the agents do not have sufficiently powerful models of each other to use communication-free coordination strategies such as (Huber & Durfee 1995). Each MOVERS-WORLD problem-solving episode includes some goal(s) that can only be solved by coordinated agents, so communication is an essential part of the community activity required to solve problems.

Communication is used to attempt to establish cooperation when the set of agents working on a goal at a given time is inadequate. Cooperation is not guaranteed during communication since agents have their own decision-making strategies[1] and even if an agent is willing to cooperate she may be unable to do so. An agent who is unwilling or unable to assist can propose

[1] All agents presently use the same probabilistic, rational strategy based upon the expected number of top-level goals that alternative plans achieve. Despite their common strategy, agents will make different decisions because of their varying experiences interacting with the world.

an alternative cooperative activity which the original requester must now contemplate adopting.

Communication and a meta-planner operator jointly form the mechanism whereby coordination occurs in MOVERS-WORLD. When cooperation is first established during communication, the agents must determine how they will coordinate the cooperative activity. Sometimes, nothing need be done – for example, if two lifters are both adjacent to a ready-to-be-lifted box and both are ready to lift it. More often, though, the requester will have to idle for one or several rounds – for example, if the requestee lifter is not currently ready to lift the box. Another common situation involving idling occurs after a lifter agrees to load a box onto a hand-truck for a hand-truck operator. Idling is presently implemented as a busy-wait by pushing a meta-planner operator WAIT onto the plan. While the WAIT is at the top of her plan, an agent is waiting for one of two events to occur: communication indicating that joint action can occur (e.g., the other lifter now indicates she is ready to act) or the completion of her request (e.g. the box appears on the hand-truck). If an agent is idle too long, she will become frustrated and inquire about the status of her request.

4 Adding Cooperative Procedures to Collective Memory

After the community of agents solves a problem, cooperative procedures are added to collective memory in order to improve the community performance in future problem-solving episodes. ADD-C-PROCS is the main procedure which an agent uses to update her procedural knowledge case-base with the information contained in her execution trace. This occurs in step 4 of Figure 1; pseudo-code is given in Figure 3. Briefly, the agent must summarize her run-time activities, both to remove unwanted data and to simplify further analysis. This summary is reorganized into chunks of operators that are associated with the goals they achieve. Then the operators in each chunk are modified to better prepare them for future re-use. Finally, the agent compares this chunk to current case-base entries to determine if the chunk should be added to the case-base.

Function ADD-C-PROCS(execution-trace)

```
summary = SUMMARIZE-TRACE(execution-trace)
foreach (chunk goals) in SPLIT-TRACE(summary)
  prepared = PREPARE-CHUNK(chunk)
  MODIFY-PKCB(prepared goals)
```

Figure 3: Algorithm to update collective memory.

In our model, the cooperative procedures stored in collective memory are culled primarily from execution traces. Others, such as (Carbonell 1983) and (Laird, Rosenbloom, & Newell 1986), have argued that an agent should store planning histories in memory. However, reusing plan derivations will not, in general, produce a sequence of actions to better solve a similar problem in the future. On the other hand, execution traces encapsulate the history of both planned and unplanned agent interactions with the domain. Consequently, cooperative procedures can be learned through the machinery of collective memory that were not developed in a single planning session.

Summarization

Agents abandon plans and switch top-level goals for a variety of reasons at run-time such as operator failure, incoming requests for cooperation, resource conflict, and the failure of other agents to abide by previous agreements. This means the execution trace may contain actions that should not be part of learned cooperative procedures. The goal of the summarization process is for agents to remove such actions, while preserving consequential ones.

SUMMARIZE-TRACE, whose pseudo-code is given in Figure 4, removes actions from the execution trace in four passes. The first pass does almost all of the summarization; the underlying justifications for the pruning criteria used during this pass are fairly self-evident. The third pass is really part of the preparation process of updating collective memory. However, removing planner-reconstructible operators at this point during summarization simplifies both the next pass of summarization, which removes communication related to those operators, and the reorganization which precedes preparation.

Function SUMMARIZE-TRACE(trace)

```
Pass 1: foreach event E with action A in trace
  Remove E from trace if:
    A is a no-op
    A is a failed attempted operator
    A is a refusal or a refused request
    A is an agreement or an agreed-to request,
       but the request was never accomplished
    A achieved only lowest criticality goals
    A is a WAIT
    A was undone by the subsequent operator
Pass 2: foreach event E in trace
  Remove E if A was intended to achieve a top-
    level goal which the agent did not achieve
Pass 3: foreach event E still in trace
  Remove other events still in the trace whose
    actions are planner-reconstructible from A
Pass 4: foreach event E still in trace
  Remove E if A is an agreement or an agreed-to
    request concerning a removed joint action
```

Figure 4: Algorithm to summarize traces.

The second summarization pass requires some explanation. This step is designed to weed-out operators that seemed to be useful at the time, but eventually were not. For example, suppose a hand-truck operator (HTO) gets a lifter (L1) to load a box onto a hand-truck, which HTO then pushes to the street and stands up. Suppose further that no lifter agreed to unload the box when it was at the street and HTO eventually pushed the hand-truck containing the box back to the box's original location. At this point, L1 unloads it from the hand-truck, carries it (back) to the street and loads it onto the truck. In this case, HTO's request did not help to achieve the goal and should be removed from the summary.

An example of summarization

The simplest plan that lifters learn that their first-principles planner does not construct is to load a box onto a hand-truck and then later unload it and load it onto a truck at the behest of a hand-truck operator. This chunk can get created from the following snippet of activity involving medium-sized box MBOX3.

First, a high-level description:

1. Hand-truck operator HTO asks lifter L1 to get MBOX3 onto hand-truck HANDTR2. L1 agrees and does so via lifting and loading the box. L1 next fails in an attempt to lift large box LBOX2 by herself, does nothing for a round since she has no plan, and lifts small box SBOX5 while HTO tilts, pushes to the street, and stands up HANDTR2.
2. HTO asks L1 to get MBOX3 onto truck TRUCK. L1 agrees, puts SBOX5 back down, moves to the street, unloads the box from the hand-truck and then loads it onto the truck.

L1 records her behavior internally in each round of activity, including information about that round's active goals, cooperation agreements, state of the world, attempted action, reason for attempting the action, and result of the attempt. These internal structures contain too much data to show fully, but the gist of them is clear in the following description of L1's execution trace:

```
(<agreed to achieve (ON MBOX3 HANDTR2) for HTO>
 <executed (LIFT MBOX3)>
 <executed (LOAD MBOX3 HANDTR2)>
 <failed to (LIFT LBOX2)>
 <no-op>
 <executed (LIFT SBOX5)>
 <agreed to achieve (ON MBOX3 TRUCK1) for HTO>
 <executed (PUT-DOWN SBOX5)>
 <executed (MOVE STREET)>
 <executed (UNLOAD MBOX3 HANDTR2)>
 <executed (LOAD MBOX3 TRUCK1)>)
```

SUMMARIZE-TRACE reduces L1's activity to:

```
(<agreed to achieve (ON MBOX3 HANDTR2) for HTO>
 <executed (LOAD MBOX3 HANDTR2)>
 <agreed to achieve (ON MBOX3 TRUCK1) for HTO>
 <executed (LOAD MBOX3 TRUCK1)>)
```

L1 accomplished both agreements and the corresponding top-level goal (ON MBOX3 TRUCK1), so none of the summarization rules applied to them. Likewise, the two LOADs were left untouched by SUMMARIZE-TRACE.

The other actions in L1's execution trace were removed for varying reasons. The attempted LIFT of the large box was pruned just as any other failed operator would. (LIFT SBOX5) is omitted since it was undone by the subsequent PUT-DOWN operator, which was in turn dropped since it achieved HANDEMPTY, a lowest criticality goal. Lifting, and later unloading, MBOX3 is removed in the third pass of SUMMARIZE-TRACE since they are planner-reconstructible from the precondition (HOLDING MBOX3) of the LOAD operators following them.

From Summary To Memory

After the execution trace has been summarized, additional processing is required in order to add cooperative procedures to collective memory. ADD-C-PROCS (again see Figure 3) next calls function SPLIT-TRACE, which generates a list of operator chunks and the goals they achieve. PREPARE-CHUNK modifies each chunk to make a cooperative procedure that is suitable for reuse. The procedural knowledge casebase is then modified to include the novel procedures by MODIFY-PKCB. These three support routines are outlined in Figure 5.

Function SPLIT-TRACE(summary)

```
union ( Temporal-chunks (summary),
        Goals-oriented-chunks (summary) )
```

Function PREPARE-CHUNK(chunk)

```
foreach operator in chunk
  Consistently replace goal literals with
    the same, newly generated variable
  Convert agreement into WAIT-FOR
  Augment operator with additional
    planning binding information
```

Function MODIFY-PKCB(new goals)

```
foreach old in PKCB indexed by goals
  if (plan old) maps to (plan new)
     if (context old) maps to (context new)
       if GENERALIZED-CONTEXT(old new) exists
         set (context old) to generalization
       else if contexts differ
          add entry to PKCB
     else add entry to PKCB
     return
```

Figure 5: Support algorithms for ADD-C-PROCS.

SPLIT-TRACE. The purpose of this function is to identify chunks of operators; each chunk of operators must collectively accomplish a clearly-defined set of top-level goals. It currently identifies two kinds of chunks, but in the future, additional techniques may identify other types of chunks. SPLIT-TRACE takes a summarized execution trace as input and returns a list of operator-chunk/goals pairs as output.

Using the active goal information recorded in the trace events, SPLIT-TRACE considers a consecutive subsequence of the (chronologically-sorted) summary to be a chunk if the joint effects of the executed operators achieves the conjunction of the goals associated with them. SPLIT-TRACE combines these "temporal" chunks with "goal-directed" ones, which are determined by simply grouping together operators associated with identical goals.

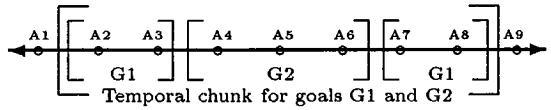

Figure 6: Schematic of a temporal chunk.

These two types of chunks are represented schematically in Figure 6. The axis represents a portion of an agent's activity timeline with markers indicating the execution of actions A1 through A9 (more actions may precede or follow these actions). If the agent, individually or in cooperation with other agents, accomplished goals G1 and G2 during the intervals marked on the timeline, there would be three chunks identified from this activity timeline. There would be a goal-directed chunk for G1 of (A2, A3, A7, A8), a goal-directed chunk for G2 of (A4, A5, A6) and a temporal chunk for both goals of (A2, A3, A4, A5, A6, A7, A8). This last chunk is the simplest type of chunk which is a temporal chunk that is not also a goal-directed chunk; a similar chunk forms the basis for agents to learn to interleave goals in the example in Section 5.

PREPARE-CHUNK. The operators of a chunk may need to be modified to make them better suited for future re-use. PREPARE-CHUNK makes purely representational preparations as well as more substantial changes designed to improve cooperative plans, such as the formation of macrops (Fikes, Hart, & Nilsson 1972). In other words, when possible, the chunk is made more general and more readily adapted in the future.

The representational preparations are straightforward. Obviously, the operator literals must be replaced by new variables. Also, a WAIT-FOR operator is introduced whenever an agent agreed to a request. WAIT-FOR operators are functionally equivalent to the coordinating operator WAIT already discussed, but there is a semantic difference that is relevant during communication. WAIT implies an agreement **already** exists but WAIT-FOR only implies an agent **expects** an agreement to be established.

The third pass of SUMMARIZE-TRACE provides the benefit of making the incoming chunks more general and should be considered a means of improving the cooperative plans. It improves them because removing operators that achieve subordinate goals makes the chunk more easily adapted in the future (at the cost of regenerating the original operator if it is needed again).

The most complicated portion of the preparation phase involves potentially augmenting operators to macro-operators by including additional binding information, in the form of new roles and preconditions. This additional information is identified via an annotation process, similar to that of (Kambhampati & Hendler 1992), which records the reasons why operator variables took on their final ground value.

Operator augmentation is better understood with a simple motivating example than with a long explanation. HTO's execution trace from the previous example is:

```
(<asked L1 to (ON MBOX3 HANDTR2)>
 <waited for (ON MBOX3 HANDTR2)>
 <waited for (ON MBOX3 HANDTR2)>
 <executed (TILT-HANDTR HANDTR2)>
 <executed (PUSH-HANDTR HANDTR2 STREET)>
 <executed (STAND-HANDTR HANDTR2 STREET)>
 <asked L1 to (ON MBOX3 TRUCK1)>
 <waited for (ON MBOX3 TRUCK1)>
 <waited for (ON MBOX3 TRUCK1)>
 <waited for (ON MBOX3 TRUCK1)>
 <waited for (ON MBOX3 TRUCK1)>)
```

And after summarization and literal replacement:

```
((SIGN ?AGENT (ON ?BOX ?HANDTR))
 (STAND-HANDTR ?HANDTR ?FINAL-LOCATION)
 (SIGN ?AGENT (ON ?BOX ?TRUCK)))
```

The SIGN operator (which triggers communication) does not normally contain a ?HANDTR role or binding information regarding it. Nonetheless, HTO needs some way to determine an appropriate instantiation for ?HANDTR before the first SIGN operator can be re-used. This can be accomplished if the operator is augmented with the pair of preconditions (AT ?HANDTR ?STARTING-LOCATION) and (AT ?BOX ?STARTING-LOCATION).

MODIFY-PKCB. The final step in updating the procedural knowledge case-base is MODIFY-PKCB. A prepared chunk and the top-level goals it achieves form a potentially new cooperative procedure. This procedure is compared to other entries in the procedural knowledge case-base. The chunk is added if there is no matching entry or the matching entry cannot be generalized.

It should be noted that indexing at both storage and retrieval time is based on observable characteristics of the top-level goals being achieved. In a multi-agent setting, standard CBR techniques (Kolodner 1993) can

lead to global difficulties when individuals use local criteria to determine the best case to retrieve, as discussed in (NagendraPrasad, Lesser, & Lander 1995). In MOVERS-WORLD, any such inconsistencies are resolved in the exact same manner as when the plans were generated from first principles. In other words, when an agent needs to request cooperation or to coordinate her actions with a cooperating agent, she communicates. If the requestee's actions are being guided by a plan based upon the same shared experience, there will be no difficulties. Otherwise, the requestee will have to decide whether to abandon her current plan in favor of the request or to suggest alternative cooperative behavior (based upon either a different plan from the case-base or a plan generated from scratch).

5 Learning to Interleave Goals

As mentioned previously, using execution traces as a basis for future cooperative plans allows the agents to learn plans beyond the scope of the first-principles planner (or a traditional second-order planner based on it). Another example shows how lifter L2 learns to interleave two goals. In this example, HTO and L1 will act as in the first example, with the box in question now extra large box XLBOX1 rather than MBOX3. The following are L2's plans and actions, starting four steps before HTO makes her first request to L1.

1. L2 creates a typical plan to get XLBOX1 onto the truck: to lift, carry and load the box jointly with L1. L1 agrees to lift the box together and they do so. L1 then agrees to carry the box to the street. However, XLBOX1 is too large to carry, even jointly, and the operator (and hence rest of the plan) fails.
2. L2 creates a plan to get SBOX5 onto the truck. The plan consists of putting down XLBOX1 with L1's help and then lifting, carrying and loading SBOX5 onto the truck by herself. L2 is delayed in asking for L1's assistance because HTO calls first with a request to put XLBOX1 onto HANDTR2. L1 agrees to help HTO.
3. When L2 does ask L1 to help achieve HANDEMPTY via putting XLBOX1 down together, L1 replies that she would rather load the box together onto the hand-truck. L2's planner adapts her current plan by replacing the PUT-DOWN-TOGETHER with the appropriate LOAD-TOGETHER. The agents then load XLBOX1 onto the hand-truck.

   L2 continues on with her plan and loads SBOX5 onto the truck. Meanwhile, HTO has pushed the hand-truck to the street and L1 has agreed to get XLBOX1 onto the truck.
4. L2 constructs a plan to get large box LBOX2 onto the truck and moves back to ROOM1.
5. L2 is interrupted before attempting to lift LBOX2 by a request from L1 to help unload XLBOX1 from the hand-truck. L2 constructs the plan of moving to the street and then unloading XLBOX1. The agents do so.
6. L1 asks L2 to load XLBOX1 onto the truck and they do.

The sequence of actions L2 undertakes corresponds to six different calls to the planner. Nonetheless, L2 can extract a single cooperative procedure from her execution traces by the machinery of collective memory (showing the original literals instead of new variables for clarity):

```
((WAIT-FOR L1 (ON XLBOX1 HANDTR2))
 (LOAD-TOGETHER XLBOX1 HANDTR2)
 (LOAD SBOX5 TRUCK1)
 (WAIT-FOR L1 (ON XLBOX1 TRUCK1))
 (LOAD-TOGETHER XLBOX1 TRUCK1))
```

Interleaving loading and unloading a box onto the hand-truck with loading a small box onto the truck, as in this example, is the most useful procedure lifters learn in the current version of the system. Its utility arises from low idle time and wide applicability.

6 Results and Analysis

Our collective memory test-bed system solves randomly generated MOVERS-WORLD problems, in isolation or in sequences. Individual problems are constructed by randomly selecting subsets from the pool of permanent MOVERS-WORLD objects (agents, hand-trucks, and trucks) that will be active for that problem. Then a random group of boxes and locations is constructed and a list of goals involving them is generated.

A database was created of 60 problems, whose goals were always to move all boxes to the truck. The number of boxes was uniformly distributed between 3 and 5. The average number of rounds required to solve these problems without using collective memory to learn was 42.6, with a standard deviation of 0.64.

It should be noted that there are many sources of randomness that can effect the number of rounds of activity it takes a community to solve any given problem. The experiments underlying the data presented here are designed to control as many sources of randomness as possible in order to make comparisons meaningful. For example, it is not feasible to determine learning curves by running the system on all possible permutations of the database problems, so the system is run on four predetermined groups of sequences. Each group of sequences is balanced in the following way: each of the database problems occurs once as the first problem of some sequence in the group, once as the second of a different sequence in the group, et cetera.

There are presently two components of collective memory. Besides the procedural knowledge case-base described in this paper, agents can use *operator probabilities trees* to estimate the probability of success for operators an agent may attempt. Accurate estimates improve agent performance since the agents' first-principles planner is a hierarchical planner (Sacerdoti 1974) which uses probabilities to guide search, including role-binding selections (c.f. (Kushmerick, Hanks, & Weld 1995)). Operator probabilities trees are incrementally updated as the agent interacts with

the domain and this enables the planner to improve during the course of solving a problem. Results will be presented verifying this intra-problem learning, but the reader is directed to (Garland & Alterman 1996) for more technical details on operator probability trees.

Case-based reasoning has been previously demonstrated to be an effective technique for reducing CPU usage during planning in a standard distributed AI planner in (Sugawara 1995). In our domain though, CPU usage is an inappropriate measure of community performance because we are primarily interested in improving the community's run-time behavior, not the speed at which they plan (or retrieve plans from memory). However, since primitive actions and communication are simulated, roughly 90% of the CPU usage for our system is devoted to agent planning. The statistic we consider best suited to our domain is the number of rounds (defined in Section 2), which views primitive action and communication as taking the same order of time as a single planning session.

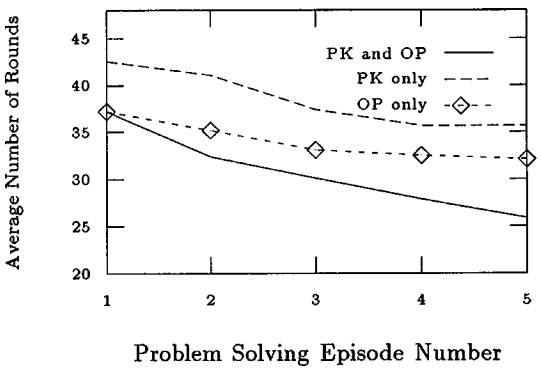

Figure 7: Overall community performance improves.

Figure 7 shows how the two collective memory structures improve community performance. Both the operator probabilities tree (abbreviated OP) and the procedural knowledge case-base (abbreviated PK) lead to statistically significant improvements. OP shows substantial intra-problem learning, reducing the average number of rounds for solving the first problem of a sequence to 37.3. This number continues to go down, albeit slowly, reaching 32.0 for the fifth problem. PK has less dramatic improvement, dropping to 35.7 for the fifth problem. Using both structures produces much better results than either of the two alone, as the average number of rounds drops steadily, ending at 25.9, which is close to a 40% improvement over no learning. Standard deviations for these points ranged from 0.56 to 0.90.

In some domains (e.g. robot agents), the time it will take an agent to attempt primitive actions will far exceed the amount of time the agent spends planning or communicating. And in other domains (e.g. web-based applications), the communication costs will dwarf the

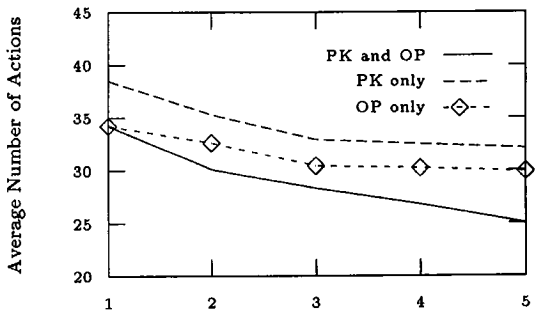

Figure 8: Community primitive action decreases.

costs of planning or acting. Figure 8 shows that the use of collective memory in MOVERS-WORLD reduces the number of primitive actions needed to solve problems. Figure 9 shows that the use of collective memory in MOVERS-WORLD reduces run-time communication.

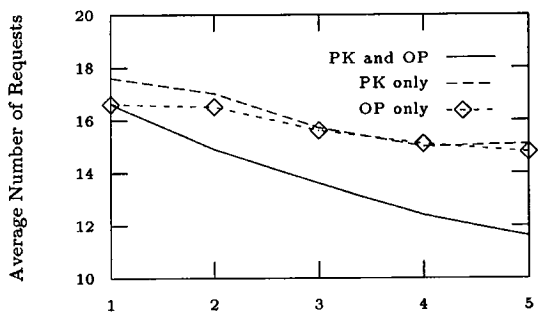

Figure 9: Community communication decreases.

The improved run-time performance of the community is a direct result of the fact that the planner produces plans that are either more efficient (e.g. interleaves goals) or more likely to be successful or both. The question must be asked whether this improvement in plan quality comes at a high price in increased planner effort due to increased match costs and/or increased branching factors.[2] Using collective memory in MOVERS-WORLD, there is no high price to pay. There is a performance decline in the amount of effort the planner requires in order to instantiate local role-binding variables (see Figure 10). However, this is more than compensated by a dramatic reduction in the number of planning search nodes expanded during calls to the planner as shown in Figure 11.

[2]There are other concerns regarding case-bases, principally increased retrieval time, but there are techniques to handle this such as (Minton 1990; Smyth & Keane 1995).

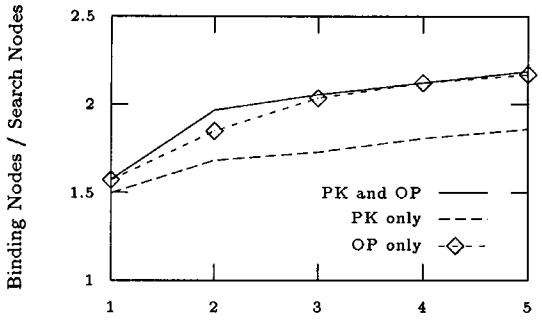

Figure 10: Agents search more to find role bindings.

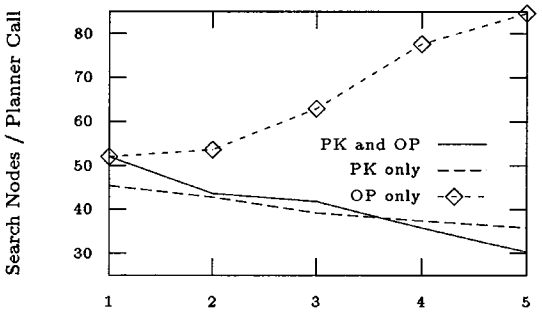

Figure 11: PK reduces planning search.

7 Concluding Remarks

This paper has shown the significant benefits that collective memory provides to a community of agents in a problem-solving setting. We presented illustrative examples of how cooperative procedures that are beyond the scope of the scratch planner can be learned and stored into collective memory. Empirical results confirmed the usefulness of collective memory in an implemented test-bed system. These results showed that cooperative procedures and another collective memory structure, operator probabilities trees, each independently lead to reductions in the amount of time a community takes to solve randomly generated problems. Furthermore, the two structures are more effective together than either alone, showing that the structures facilitate non-overlapping aspects of learning.

Acknowledgments

This work was supported in part by the Office of Naval Research under grant number N00014-96-1-0440.

References

Alterman, R. 1988. Adaptive planning. *Cognitive Science* 12:393–421.

Carbonell, J. 1983. Derivational analogy and its role in problem solving. In *Proc. Third National Conference on Artificial Intelligence.*

Cole, M., and Engeström, Y. 1993. A cultural-historical approach to distributed cognitition. In Salomon, G., ed., *Distributed Cognitions.* Cambridge University Press. 1–46.

Corkill, D. D. 1979. Hierarchical planning in a distributed environment. In *Proc. Sixth International Joint Conference on Artificial Intelligence*, 168–175.

Fikes, R. E.; Hart, P. E.; and Nilsson, N. J. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3:251–288.

Garland, A., and Alterman, R. 1996. Multiagent learning through collective memory. In *1996 AAAI Spring Symposium on Adaptation, Coevolution and Learning in Multiagent Systems*, 33–38.

Grosz, B. J., and Kraus, S. 1998. The evolution of SharedPlans. In Rao, A., and Woolridge, M., eds., *To Appear in Foundations and Theories of Rational Agency.*

Hammond, K. J. 1990. Case-based planning: A framework for planning from experience. *Cognitive Science* 14:385–443.

Huber, M. J., and Durfee, E. H. 1995. Deciding when to commit to action during observation-based coordination. In *Proc. First International Conference on Multiagent Systems*, 163–170.

Kambhampati, S., and Hendler, J. A. 1992. Control of refitting during plan reuse. *Artificial Intelligence* 55:193–258.

Kolodner, J. L. 1993. *Case-Based Reasoning.* San Mateo, CA: Morgan Kaufmann Publishers.

Kushmerick, N.; Hanks, S.; and Weld, D. 1995. An algorithm for probabilistic planning. *Artificial Intelligence* 76.

Laird, J. E.; Rosenbloom, P. S.; and Newell, A. 1986. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning* 1:11–46.

Minton, S. 1990. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence* 42:363–392.

NagendraPrasad, M. V.; Lesser, V. R.; and Lander, S. 1995. Retrieval and reasoning in distributed case bases. Technical Report CS TR 95-27, University of Massachusetts.

Sacerdoti, E. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5:115–135.

Smyth, B., and Keane, M. T. 1995. Remembering to forget. In *Proc. Fourteenth International Joint Conference on Artificial Intelligence*, 377–382.

Sugawara, T. 1995. Reusing past plans in distributed planning. In *Proc. First International Conference on Multiagent Systems*, 360–367.

Veloso, M., and Carbonell, J. 1993. Derivational analogy in prodigy: Automating case acquisition, storage, and utilization. *Machine Learning* 10:249–278.