Autonomous Agents that Learn to Better Coordinate

Andrew Garland and Richard Alterman
Volen Center for Complex Systems
Brandeis University, Waltham Massachusetts 02454
{aeg,alterman}@cs.brandeis.edu

Brandeis University Technical Report CS-TR-03-237
July, 2003
To appear in *Autonomous Agents and Multi-Agent Systems*

Abstract

A fundamental difficulty faced by groups of agents that work together is how to efficiently coordinate their efforts. This *coordination problem* is both ubiquitous and challenging, especially in environments where autonomous agents are motivated by personal goals.

Previous AI research on coordination has developed techniques that allow agents to act efficiently from the outset based on common built-in knowledge or to learn to act efficiently when the agents are not autonomous. The research described in this paper builds on those efforts by developing distributed learning techniques that improve coordination among autonomous agents.

The techniques presented in this work encompass agents who are heterogeneous, who do not have complete built-in common knowledge, and who cannot coordinate solely by observation. An agent learns from her experiences so that her future behavior more accurately reflects what works (or does not work) in practice. Each agent stores past successes (both planned and unplanned) in their individual casebase. Entries in a casebase are represented as *coordinated procedures* and are organized around learned expectations about other agents.

It is a novel approach for individuals to learn procedures as a means for the group to coordinate more efficiently. Empirical results validate the utility of this approach. Whether or not the agents have initial expertise in solving coordination problems, the distributed learning of the individual agents significantly improves the overall performance of the community, including reducing planning and communication costs.

1 Introduction

Research on groups of agents that work together is a large and growing field. This field covers many diverse interests and motivations such as mobile robots, software agents, and smart objects. A fundamental difficulty faced by interacting agents is how to coordinate their efforts when they have overlapping or common objectives. This *coordination problem* [33] is both ubiquitous and challenging, especially in environments where agents have limited knowledge about, and control over, other agents and the world around them.

In human terms, the coordination of joint behaviors is so commonplace and constant that it is easy to overlook or trivialize its importance and the difficulty of achieving it. Coordination problems are regularly faced, and solved, in everyday settings such as conversations and shopping as well as in highly structured settings such as musical ensembles or athletic teams. In order to solve these problems, it is necessary to coordinate the expectations of the participants (about both intentions and actions).

This paper presents learning techniques that allow agents to solve coordination problems more efficiently. These techniques do not presume that agents have initial expertise about how to solve coordination problems, yet provide effective guidance for agents who do have extensive *a priori* knowledge. Using these techniques, at runtime, an agent acquires knowledge about the task environment and the capabilities of other agents. Thus, the agent's future behavior more accurately reflects what works (or does not work) in practice.

In order to recall practical solutions to coordination problems, agents learn *coordinated procedures* that are derived from past, possibly unplanned, successful joint behavior. A coordinated procedure encodes information about both domain actions and expectations of one agent about other agents. Learned procedures are extracted from execution traces, which are the result of multiple planning sessions occurring at various times during the activity.

It is a novel approach for individuals to learn procedures so that the group can coordinate more efficiently. Learning coordinated procedures extends single-agent case-based reasoning and procedural learning (e.g. [25, 14, 19, 38]) into multi-agent domains. However, multi-agent domains pose numerous technical challenges that are not addressed by prior work. In addition, using runtime episodes as the currency of memory differs from traditional procedural learning techniques, which are centered on plans or plan histories that are the output of a single planning session.

The agents in this work are motivated by personal goals and do not reason about group-wide objectives or seek to establish or maintain group-wide mental attitudes. Such agents are considered to be *autonomous*. Further, the model of joint activity presented in this work encompasses autonomous agents who are heterogeneous, who do not have complete built-in common knowledge, and who cannot coordinate solely by observation. As a result, this model is better suited

for open-world environments than previous research on either solving coordination problems efficiently from the outset [10, 7, 37] or learning to solve recurrent coordination problems more efficiently in the future [36, 17, 18, 31].

The next section provides a more detailed account of coordination problems. Section 3 presents an overview of our model of joint activity; Section 4 describes the techniques by which agents learn coordinated procedures; and Section 5 contains results, including examples of learned procedures. Following a discussion of related research, the paper concludes with a brief summary of the contributions of this work.

2 Coordination problems

Interacting agents face a coordination problem when, to achieve a shared goal, they must coordinate their individual actions. In order to solve coordination problems, agents must coordinate their expectations about both intentions and actions.

Expectations are coordinated by means of coordination devices. A coordination device is anything that enables the group to form a mutual belief about how the individual agents will interleave their individual actions. Notable coordination devices are explicit communication, precedence, (perceptual) prominence, and convention. Coordination cannot be guaranteed without coordination devices since the actions of an agent cannot always be reliably predicted by others. The kinds of coordination devices available to the agents constrain the set of possible solutions to coordination problems.

Coordination problems exist when agents have a common goal, even if each agent is able to independently achieve the goal. Solving even such simple problems may be non-trivial: the agents may not be aware of their mutual interests, the agents may not agree on which agent should achieve the goal, and the agents may not be able to determine whether their actions interfere with each other.

A more interesting coordination problem arises when agents agree to execute a joint action γ but some of the participants are not ready to attempt γ immediately. In this case, the group must work together to determine when to start attempting γ. Figure 1 shows a depiction of this situation for two agents, A and B, each of whom must execute different "set up" steps.

In many circumstances, the agents will not be able to correctly determine when to start γ based solely on observation and inference. This is the case when agents are not in the same location, when agents do not inform each other of what set up must be done, or when agents do not know how to recognize when the others are prepared to attempt γ. Note that a clock, even a global one, cannot be relied upon to solve such coordination problems when there is uncertainty about

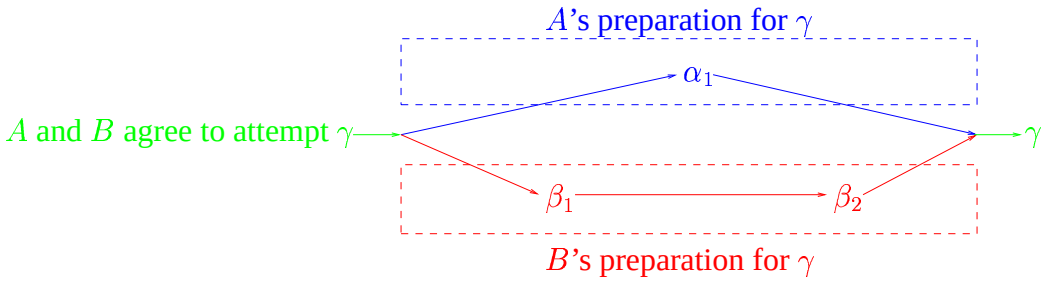

Figure 1: A and B must coordinate the beginning of joint action γ after completing their individual preparations for γ.

action durations, possibilities of interruptions and execution failures, or unpredictable delays due to external events.

Another common situation in which multiple agents must coordinate their efforts is when they have differing execution abilities. In many such cases, the solution will have to be "discovered" dynamically, through the interactions of the agents. Figure 2 shows an abstract description of this situation for two agents who want to change an object o from an initial state $S_1(o)$ to goal state $S_8(o)$. In this figure, possible states $S_k(o)$ are marked by ovals and individual actions by A and B are labeled $\alpha_i(o)$ and $\beta_j(o)$, respectively. (The o argument will be omitted unless needed for clarity.) The arrows indicate the required ordering of actions for the two agents to jointly solve the problem. Over the course of the solution, o "transitions" through eight different states. Only A knows how to transform S_1 to S_3. Only B knows how to get from S_3 to S_6. And only A knows how to change S_6 to S_8.

A cannot generate a plan to match this solution path on her own in the absence of planning knowledge about B's capabilities. Backward chaining can identify S_6 as a precursor to S_8 and forward-chaining can identify S_3 as a successor to S_1. However, A cannot distinguish the pair of states (S_3, S_6) that is relevant to this solution path from the many other pairs of states that are not relevant to any solution path.

To get a sense for how the complexity of coordination problems scales, suppose that A and B are working on two objects, o_1 and o_2, and that A is capable of an additional action α_5 that transforms S_2 into S_7. First of all, note that A is now unlikely to do α_2 of her own accord since α_5 enables A to transform either object efficiently by herself. However, efficiently solving both goals requires α_2! Namely, the most efficient solution possible requires that B asks for help with o_1, that A agrees to help by doing $\alpha_2(o_1)$, and that A correctly adapts her plan to do $\alpha_1(o_2), \alpha_5(o_2), \alpha_4(o_2)$ at the same time that B does $\beta_1(o_1), \beta_2(o_1), \beta_3(o_1)$. While none of these decisions is unlikely independently, neither agent possesses enough knowledge to reliably act in this way so the chance of them all occurring is less likely.

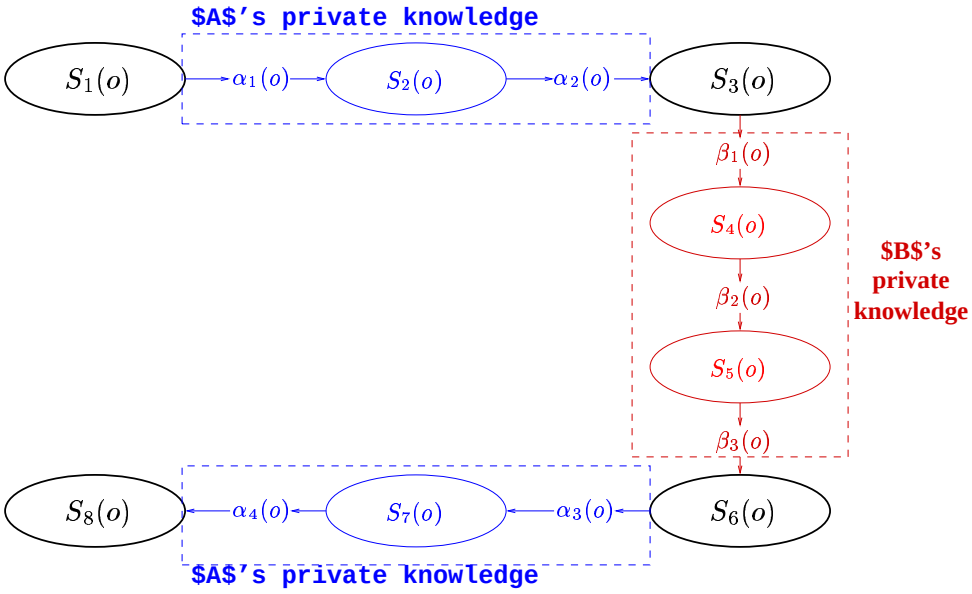

Figure 2: A and B can solve this problem by interleaving individual actions, yet neither agent has enough problem-solving knowledge to generate such a plan.

As the number of agents, goals, and possible solutions involved increase, there is an exponential increase in the number of decisions that must all be made "correctly" for the community to perform optimally. The memory-based learning techniques presented in Section 4 can lead the community to solve coordination problems more effi ciently by predisposing agents to approach familiar situations in compatible ways, which simplifi es decisions about whether to cooperate and how to adapt plans.

3 Model of joint activity

The model of *joint activity* [6] in this work is designed to accommodate domains with any of the following attributes, characteristic of coordination problems found in many AI domains:

- The environment may contain a heterogeneous collection of agents.
- The goals of an agent are private knowledge unless she includes this information when communicating to another agent. Agents do not necessarily have the same set of goals.
- Agents make their own decisions about what order to work on their goals.
- Agents make their own decisions about whether or not to cooperate, based on their personal goals, current plan, and belief about the state of the world.

- Actions, both individual and joint, are not guaranteed to succeed.
- Agents may not posses enough commonality to coordinate exclusively by observation.
- Agents (implicitly) share the desire to solve coordination problems as efficiently as possible.
- Agents may not be designed with all relevant engineering information about the others, such as other agents' goal-ordering strategies, decision-making strategies, planning abilities, or execution abilities.

These attributes reflect a desire to support open systems, where agents perforce have incomplete knowledge about other agents and the task environment.[1] Given these attributes, even seemingly simple tasks contain imposing hurdles to efficient coordination. These hurdles lead to run-time failures, mistakes, false starts, etc. that the agents overcome through re-planning, communication, and adaptation. In the worst case, activity may not even be successfully completed if some actions are irreversible or the goals of the agents conflict.

Coordination problems are solved at runtime through communication, observation, and inference. Agents do not communicate at planning time; they plan independently, act independently, and only communicate when necessary to establish cooperation or to maintain coordination. Communication is the only mechanism whereby agents can determine if they are cooperating. 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. While observation and inference are sufficient, in some cases, to engineer the entrance to or exit from some coordination problems, the agents are not assumed to possess sufficiently detailed knowledge of each other in order to solve all coordination problems without communication.

Coordination is driven by *expectations* that are part of *coordinated procedures*. Coordinated procedures contain domain actions and expectations about other agents; expectations are non-primitive (i.e., not executable) actions that are represented just like domain actions.[2] Expectations are needed since the distribution of execution ability means that many plans of the agent will contain *impasse conditions*, i.e., conditions that must be true for the plan to succeed but the agent does not know how to make true on her own. For example, the following are impasse conditions for agent A: a domain predicate that A cannot achieve, the condition that B is ready, willing, and intending to attempt joint action γ as soon possible, or the condition that B makes a request to A. Expectations are a representation of the belief that another agent (the "enabler") will make the impasse condition true. However, the mere presence of an expectation in a coordinated procedure does not guarantee that the enabler will be willing to help.

[1] Clearly, this model does not support systems where agents are adversarial.

[2] The term "plan" will be used loosely in this paper to refer to a coordinated procedure.

Coordination points [3] are points in the activity where an actor cannot progress without assistance. At coordination points, an impasse condition is false and the corresponding expectation is at the top of the actor's plan. The expectation is said to be satisfied when the condition becomes true and is therefore no longer an impediment to the agent's forward progress. Since the agents are autonomous, there is no guarantee that the enabler will agree to try to satisfy the expectation. Colloquially, the terms "request" and "coordination point" will be used to refer to the impasse condition. In this sense, communication can be said to happen at and about coordination points.

Agents interleave planning and execution [1, 35, 9]. At planning time, an agent is content to construct a plan that contains expectations about impasse conditions; at run-time, if an expectation is not satisfied, the agent must "coordinate the expectation" by selecting a coordination mechanism to execute. The idea is that executing the mechanism will (eventually) lead the enabler to satisfy the expectation. Currently, the only implemented coordination mechanisms are to wait idly for the expectation to be satisfied or to initiate a conversation about the impasse condition with the enabler.[3]

Agents do not have to be in the same location to engage in communication. Communication between two agents is a peer-to-peer connection, similar to a telephone conversation, rather than a multi- or broad-cast. One agent calls another and either establishes a connection, gets a busy signal, or the other agent does not respond. Once a connection is established, communication is handled via frames — there are three initial request frame types and eighteen response frame types, twelve of which make an alternate request to the original caller. In most frames, the type and the request are the only slots that are filled. In practice, not all of the potential sequences of frame types are manifested; for the trials that form the basis of the learning curves presented in Section 5, there were 300,923 conversations comprising 94 different sequences. The communication language is simple enough that it could be mapped to a subset of KQML [24] or some comparable other agent language. Conversations do not occur concurrently with primitive actions.

Communication is the coordination mechanism *par excellence* to solve coordination problems since it allows agents to explicitly establish mutual beliefs. However, relying on communication has a cost, namely that the time spent conversing reduces the efficiency of the agents. Agents limit themselves to one request per conversation to keep down both communication and plan-merging costs.

When the first action of her coordinated procedure is a domain action, an agent will attempt it. When the first action is an expectation, the runtime behavior of the agent depends on several factors. First off, as a safeguard against prolonged inactivity, if an agent has waited "too long" (e.g., exceeds a pre-determined threshold) for the expectation to be satisfied, the agent will initiate

[3] Responding to a request is another coordination mechanism, but it is not a direct way to coordinate an expectation.

a discussion, possibly discovering that the other agent has malfunctioned or has opted out.[4] Otherwise, if the agent has an explicit ongoing agreement about the expectation, the agent will wait. If neither of these rules are applicable, there are three cases that describe how the agent will choose a coordination mechanism to coordinate the expectation.

In the first case, agent A expects agent B to be ready and willing to attempt joint action γ, where A and B are both participants of γ. Currently, the only way for A to coordinate this expectation is for A to initiate a dialog so that B can explicitly communicate her readiness; future work may include additional ways to signal or infer when an agent is ready for joint action. This kind of expectation is typically succeeded by γ itself, which can be attempted once the expectation is satisfied.

A second case arises when A expects B to make an impasse condition x true, where x is some domain predicate that A wants to hold but does not know how to make true. Once A believes that x is true (regardless of how x became true), A will usually execute an action that has x among its preconditions. Otherwise, unless A believes that B is already working toward making x true, A will initiate a conversation with B. A heuristic that is sometimes useful is for A to assume an *implicit agreement* already exists with B; this reduces communication in some cases and leads to unnecessary waiting in others.

The last case involves second-order expectations. Namely, two types of recursive expectations are created when transforming past experiences into a procedure to be stored in the casebase. One connotes that A expects to be asked to attempt a joint action and the other connotes that A expects to be asked to make a domain predicate true. For both types, A will wait for the request to occur rather than initiate negotiations. In principle, expectations can be about arbitrary expectations, so we have implemented a small subset of the possible recursive mental models (cf. [39]).

The response that an autonomous agent gives during communication depends, in part, on how her current goals and plan relate to the request. To be specific, the agent is more likely to agree to help if the request is compatible with one of the expectations in her own coordinated procedure. For example, an agent that receives an expected request will agree to help if possible. Cooperation is not guaranteed in most cases since agents make their own decisions. Even if they use the same decision-making policy, agents may make different decisions because of differences in their experiences, their assessment of the situation, and their current plan and goals. Also, even if an agent is willing to cooperate, she may be unable to construct a plan to do so; if an agreement is reached, each agent independently adapts her plan in order to achieve coordinated behavior (adaptation depends on the exact nature of the relationship between the request and the expectations in the current plan).

[4] As a pragmatic way to reduce communication, an agent may not notify others when she decides to opt-out of a prior agreement — this eliminates unnecessary communication when agents opt-out independently.

Agents are adaptive planners [2], so the current plan may be modified or abandoned during the course of activity as well as after making agreements. At any given point in time, an agent's knowledge of the external world is her perceivable environment and a map of the world that she constructs as she goes along. Adaptation substitutes role-bindings, adds actions, or removes actions (including expectations) when they are no longer needed given the agent's updated beliefs about the world. Plans are abandoned when they cannot be reconciled to the current circumstances, the associated top-level goal is achieved (presumably by another agent), or there is an execution failure. When an entire plan is completed or abandoned, the agent selects one or more of the unmet top-level goals to actively work on. A coordinated procedure is created for a set of goals by either selecting one from memory (see Section 4.1 for details) or creating one from scratch (using *expected discounted reward* as the planning metric; see Appendix A for details).

3.1 MOVERS-WORLD

In the test-bed domain, called MOVERS-WORLD, the task is to move boxes from a house onto a truck or *vice versa*. This domain has all of the attributes listed at the beginning of Section 3. In other words, MOVERS-WORLD is designed to make the problem of coordinating autonomous agents challenging.

Each agent is capable of attempting a subset of the primitive actions possible in MOVERS-WORLD. Normally, an agent has no knowledge about action types that she is not capable of attempting.[5] In order to simplify exposition, a MOVERS-WORLD agent will be labeled as a "lifter" (e.g., L1 and L2) or a "hand-truck operator" (e.g., HTO). A lifter can attempt to lift, carry, put down, load, or unload boxes, either individually or jointly with another agent capable of the same action type. A hand-truck operator can tilt, push, or stand up a hand-truck. Hand-truck operators are not capable of handling boxes directly so loading and unloading a hand-truck requires the cooperation of at least one lifter.

All agents are able to move and all agents use the same set of coordination mechanisms (initiating a dialog, responding to a request, and silently observing). Agents capable of the same action type may have differing abilities due to attributes such as strength. The duration of actions varies from 20 to 40 ticks (of a simulated clock); conversation durations vary depending upon the content of the dialog.

MOVERS-WORLD contains three types of objects: boxes, hand-trucks, and moving trucks. The domain features that determine whether attempted actions will be successfully executed are object size, object weight, agent strength, and object capacities. None of these are observable features of the environment. The static observable box features are height, width, depth and material.[6]

[5] Some experiments in Section 5 relax this assumption.

[6] The label associated with a box includes a S, M, L or XL solely to make it easier for the human reader.

For boxes, the size is derived from the volume of the box and the weight is a linear function of the size and material. Object capacities specify how many other objects and how much total weight the object can hold. Large and extra-large boxes require two agents to lift, and extra-large boxes are too unwieldy to be carried. Hand-trucks can hold one large or extra-large box or any combination of two smaller boxes.

While many of the primitive actions in MOVERS-WORLD have obvious BLOCKS-WORLD analogues, MOVERS-WORLD presents a far more challenging domain. The challenge is not in handling typical planning hurdles, such as deep search trees with large branching factors; rather, MOVERS-WORLD agents seek to efficiently coordinate their actions in an environment in which they have limited information about, and control over, other agents and the world.

```
Ticks  131 to  150: <STAND-HANDTR HANDTR3 PR24-STREET1> by HTO successful
Ticks  151 to  163: HTO and L1 converse
  "L1, would you help me achieve (ON PR24-MBOX0 TRUCK3)?"
  "HTO, I'll help, but you'll have to wait a bit."
Ticks  164 to  188: <MOVE PR24-STREET1> by L1 successful
Ticks  189 to  223: <UNLOAD PR24-MBOX0 HANDTR3> by L1 successful
Ticks  224 to  258: <LOAD PR24-MBOX0 TRUCK3> by L1 successful
```

Figure 3: Coordination based on first-principles plans.

Figure 3 shows an example involving coordination in unloading a box from a hand-truck and loading it onto the truck. The example is actual output of the system (with unrelated actions trimmed), but it is illustrative, not prototypical; empirically, agents agree to cooperate less than half of the time when they have different goals (as is the case in this figure). At tick 151, before conversing, an unsatisfied expectation is the first action in HTO's plan. Since there is no prior agreement about the expectation, HTO initiates a dialog; after establishing an explicit agreement, HTO waits for the expectation to be satisfied.

In Figure 4, L1 and HTO are using coordinated procedures retrieved from memory instead of first-principles plans. In this case, the lifter's plan contains an expectation that HTO will request a service. Although the two agents are working from compatible plans, L1 waits "too long"[7] for HTO to make the expected request. As it turns out, L1's inquiry comes in the same tick that HTO would have made the request, so the conversation proceeds smoothly. From an efficiency stand-point, this is no better than a first-principles solution; however, the outcome is more robust.

Figure 5 shows a scenario when agents are acting from incompatible plans recalled from memory. At tick 633, an expectation that L2 will get LBOX3 onto the truck is the first action in HTO's plan; HTO coordinates this expectation as if an implicit agreement existed. As a result, 160 ticks went by without any progress being made because the lifters did not see the situation in the same

[7] 160 ticks is the threshold for waiting "too long" for the runs in this paper.

```
Ticks  299 to  318: <STAND-HANDTR HANDTR3 PR36-STREET1> by HTO successful
Tick   317: L1 has waited too long for an expected request
Ticks  319 to  375: HTO and L1 converse
  "HTO, I was expecting a call about (ON PR36-MBOX7 TRUCK3)."
  "L1, would you help me achieve (ON PR36-MBOX7 TRUCK3)?"
  "HTO, I'll help, but you'll have to wait a bit."
Ticks  376 to  400: <MOVE PR36-STREET1> by L1 successful
Ticks  401 to  435: <UNLOAD PR36-MBOX7 HANDTR3> by L1 successful
Ticks  436 to  470: <LOAD PR36-MBOX7 TRUCK3> by L1 successful
```

Figure 4: Coordination based on compatible coordinated procedures recalled from memory.

```
Ticks  507 to  541: <LOAD-TOGETHER PR41-LBOX3 HANDTR3> by L1 and L2 successful
Ticks  613 to  632: <STAND-HANDTR HANDTR3 PR41-STREET1> by HTO successful
Tick   793: HTO has waited too long for an expectation to be satisfied
Ticks  794 to  859: HTO and L2 converse
  "L2, would you help me achieve (ON PR41-LBOX3 TRUCK3)?"
  "HTO, I'll help, but you'll have to wait a bit."
Ticks  879 to  908:<PUT-DOWN-TOGETHER PR41-XLBOX2 PR41-ROOM1> by L1 and L2 successful
Ticks  909 to  933: <MOVE PR41-STREET1> by L2 successful
Ticks  948 to 1026: L2 and L1 converse
  "L1, would you help me achieve (HOLDING-TOGETHER PR41-LBOX3)
       via (UNLOAD-TOGETHER PR41-LBOX3 HANDTR3)?"
  "L2, I'll help, but you'll have to wait a bit."
Tick  1020: HTO has waited too long for an expectation to be satisfied
Ticks 1027 to 1039: HTO and L2 converse
  "L2, I'm tired of waiting.  Are you still working on (ON PR41-LBOX3 TRUCK3)?"
  "HTO, I'm still working on it.  Chill out!"
Ticks 1027 to 1051: <MOVE PR41-STREET1> by L1 successful
Ticks 1073 to 1107: <UNLOAD-TOGETHER PR41-LBOX3 HANDTR3> by L1 and L2 successful
Ticks 1126 to 1160: <LOAD-TOGETHER PR41-LBOX3 TRUCK3> by L1 and L2 successful
```

Figure 5: Coordination based on incompatible coordinated procedures recalled from memory.

way that the hand-truck operator did. Except for the delay, everything proceeds as it would have by first principles (the joint actions by L1 and L2 starting in ticks 879, 1073, and 1126 are preceded by conversations that are not shown).

4 Learning to better coordinate

This section presents techniques for autonomous agents to improve upon their initial ability to solve coordination problems. The emphasis is on learning coordinated procedures from past successful joint endeavors. Learned procedures are culled primarily from execution traces and can encapsulate unplanned successes. In this work, agents individually apply memory-based techniques in order to use learned coordinated procedures to guide future behaviors.

Agents can learn to act more efficiently even if they do not learn coordinated procedures. Agents use and maintain tree structures, called *operator probability trees*, as a means of learn-

ing how likely individual actions are to succeed. Operator probabilities are used by the baseline planner, when making rational decisions about whether to cooperate, and when adapting coordinated procedures to fit the current setting.

Coordinated procedures and operator probability trees enable an agent to use past runtime experience to solve coordination problems more efficiently by allowing the agent to answer the following questions:

1. In the past, did I need to ask another agent to help me achieve similar goals?
2. When I asked another agent to help me achieve similar goals, was she able to help me?
3. In the past, how frequently was the other agent willing to help me when I asked?
4. Based on past experience, is the other agent likely to help me without being asked?
5. In past similar circumstances, did another agent ask me to help achieve similar goals?
6. If I expect another agent to ask me to help achieve similar goals, what has she asked me to do in the past?
7. If I expect another agent to ask me to do something, should I wait for her to ask me, should I do it without being asked, or should I contact the other agent to say that I expect her to ask me?

All learning is completely distributed; the agents learn on their own without communicating to each other. The two different learning structures are updated at different times. Coordinated procedures are learned only at the end of an activity, when the agent has time, and a better perspective, to reflect on the actions that contributed to her success. Once added to her memory, the learned procedures are available when creating plans in future activities. By contrast, operator probability trees are updated incrementally during the activity after each action (both primitive and communicative) is attempted.

4.1 Learning and retrieving coordinated procedures

Learning coordinated procedures enable individual agents to plan for new activities by recalling prior successful ones. Learned procedures are extracted primarily from execution traces, which encapsulate the history of both planned and unplanned agent interactions with the domain. Consequently, procedures are learned that were not developed in a single (or multiple) planning histories. In particular, unanticipated requests and responses that occur during communication allow

the agents to acquire coordination knowledge about other agents. Thus, some coordinated procedures stored in memory can represent unplanned successes that are potentially more efficient than first-principles plans.

An item in an agent's execution trace is an object that records a "snapshot" of one of the agent's runtime events. The snapshot includes a value that summarizes the event (e.g., successful action, delayed by resource conflict, initiated a dialog, failed to initiate a dialog, idle since no current plan) as well as the start and end times for the event. It also includes pointers to all of the related objects that the agent was using to guide her behavior at that time: action attempted, current goals and plan, world state at start of event, world state at end of event, current agreements, and all request and response frames exchanged.

Execution traces include records of executed coordination mechanisms (e.g., initiating a dialog, responding to a request, or silently observing). Those mechanisms that represent past successful joint achievement of coordination points are converted into expectations that are part of future plans. While some of these expectations could be regenerated upon re-use (by plan adaptation), the expectation is explicitly represented in order to codify who satisfied the expectation. This process adds second-order expectations (i.e., expectations about expectations) that are not present in first-principles plans.

Acting from shared past experience can lead agents to coordinate more efficiently. For example, when an agent receives a familiar request, she can retrieve a plan which anticipates subsequent requests rather than merely creating a plan to satisfy that single request. Recalling coordinated procedures from memory can be effective since remembering points of coordination predisposes the individual agent to anticipate the actions and requests of other agents. When different agents anticipate the same points of coordination, they can communicate more effectively for three reasons:

1. They will not waste time discussing alternatives that will prove to be unproductive.

2. They will not waste time negotiating over two workable alternatives.

3. In some cases, they can eliminate communication entirely.

Unfortunately, recalled procedures will only be effective if agents retrieve them in the right circumstances; in this paper, organizing memory around expectations about other agents and surface features of the environment propagates "compatible viewpoints" on which past activities to recall.

Nevertheless, acting from past experience does not guarantee that agents will coordinate more efficiently, regardless of how memory is organized. First of all, the coordination problems may not have enough regularity. Second, agents will sometimes assess the same situation in incompatible manners. This reflects both differences in experience between agents and the open-endedness of

interpretation in general. In general, global difficulties can arise when individuals use local criteria to determine the best case to retrieve [29]. In this paper, the agents discover, and redress, such mismatches during communication.

Figure 6 presents the two highest level algorithms involving acquiring and using coordinated procedures. As shown by UPDATECPCB, an individual agent updates her casebase of coordinated procedures in two steps, IDENTIFYCOORDINATEDPROCEDURES and STORECOORDINATEDPROCEDURE, each of which are the subject of a subsection. The last subsection talks about RETRIEVEFROMCPCB, which provides the preferred return value of an agent's planning routine, simply called PLAN. (If planning time is not a concern, the agent can compute both plans and select the better one.) Before getting into the technical details, the first subsection outlines how an agent can learn a novel procedure.

UPDATECPCB $(executionTrace, CPCB) \equiv$
  forall $\langle segment, goals \rangle$ **in** IDENTIFYCOORDINATEDPROCEDURES$(executionTrace)$
    $CPCB \leftarrow$ STORECOORDINATEDPROCEDURE$(segment, goals, CPCB)$

PLAN $(goals, state, CPCB) \equiv$
  $plan \leftarrow$ RETRIEVEFROMCPCB$(goals, state, CPCB)$
  if $plan$ = **nil**
    return PLANFROMSCRATCH$(goals, state)$
  else return $plan$

Figure 6: High-level algorithms for learning and generating plans.

4.1.1 Example of a learned coordinated procedure

The simplest plan that lifters learn that their first-principles planner does not construct is to load a box onto a hand-truck and later unload it and load it onto a truck at the behest of a hand-truck operator. This is not generated from first-principles when the lifters do not have built-in knowledge of the actions available to the hand-truck operator. This coordinated procedure can be learned from the following snippet of activity involving a medium-sized box.

First, a high-level description:

1. Hand-truck operator HTO asks lifter L1 to get box 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, spends a tick in observation 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 at each tick, including information about that tick's active goals, cooperation agreements, state of the world, attempted action, reason for attempting the action, and result of the attempt. Only the gist of these internal structures is shown in the following listing of the relevant portion of L1's execution trace:

```
<agreed to achieve (ON MBOX3 HANDTR2) for HTO>
<executed <LIFT MBOX3>>
<executed <LOAD MBOX3 HANDTR2>>
<failed to <LIFT LBOX2>>
<observed>
<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>>
```

This trace is "cleaned" by removing the failed primitive action and the time spent in observation. Then the trace is segmented into groups of actions that are related by the goals toward which they contribute. For the medium sized box, the trace segment is:

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

Summarization of this grouping then removes any action that is planner-reconstructible:

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

Next, each past agreement is converted into a second-order expectation that represents that the lifter expects to be asked for help again in the future. Finally, to prepare this group of actions for retrieval in the future, literals such as MBOX3 must be replaced by newly generated variables. Variables are typed and are usually given a machine generated label such as ?T1524; for readability, the variable label has been replaced with the type. The procedure stored in memory is:

```
<EXPECT ?AGENT (EXPECT L1 (ON ?BOX ?HANDTR))>
<LOAD ?BOX ?HANDTR>
<EXPECT ?AGENT (EXPECT L1 (ON ?BOX ?TRUCK))>
<LOAD ?BOX ?TRUCK>
```

4.1.2 Identifying coordinated procedures

The first half of updating the memory of coordinated procedures is to reflect upon the activity that just ended in order to identify sets of actions that collectively accomplished a useful purpose. A key part of this process in dynamic multi-agent environments is to remove inefficiencies in order to prevent reifying sub-optimal solutions to coordination problems. Figure 7 presents pseudo-code describing the procedure IDENTIFYCOORDINATEDPROCEDURES; the basic idea behind CLEAN and SEGMENT will be given before delving into the details of REMOVEINEFFICIENCIES.

IDENTIFYCOORDINATEDPROCEDURES $(executionTrace) \equiv$
  $cleanTrace \leftarrow$ CLEAN$(executionTrace)$
  $CPs \leftarrow$ LIST$()$
  forall $\langle cleanSegment, goals \rangle$ **in** SEGMENT$(cleanTrace)$
    $segment \leftarrow$ REMOVEINEFFICIENCIES$(cleanSegment, goals)$
    $CPs \leftarrow$ PUSH$(\langle segment, goals \rangle, CPs)$
  return CPs

Figure 7: Algorithm to identify coordinated procedures.

CLEAN removes patently unproductive behavior from execution traces to simplify further analysis. Failed primitive actions, refused requests, and time spent observing constitute the bulk of the trace entries removed during cleaning. The cleaned trace is next restructured to facilitate the learning steps that follow, which rely upon being given a context of a set of goals being achieved and the actions that contributed to achieving the goals. SEGMENT provides this context by identifying sequences of actions that are logically grouped together.

Currently, two logical relations are used to identify segments: all actions that accomplish the same goal (called "goal-groupings") and all actions that occur during a time interval (called "time-groupings"). Since the agents are goal-directed, only time-groupings that subsume goal-groupings are included. A possibility for future work is to group together actions that took place in the same location.

Execution trace segments contain inefficient solutions to coordination problems when activity unfolds in ways that were not anticipated by the agent. REMOVEINEFFICIENCIES is a crucial part of the process to learn coordinated procedures since it prevents reifying poor solutions to coordination problems. For example, if a lifter loads and unloads the same box from the same hand-truck multiple times at the same location (due to unexpected runtime failures or changes in intentions), the lifter will boil that down to loading the box onto the hand-truck just once.

Analyzing trace segments is more difficult than analyzing the output of a first principles planner. One difficulty is that actions can be effective (or ineffective) for reasons other than their intended effects.[8] Another difficulty is that trace segments may be incomplete from a planning sense.[9] The net impact of these two difficulties is that the identification of inefficient actions may depend on predicates whose truth values *do not change* as a result of any of the actions.[10]

Removing inefficiencies occurs in two passes. In the first pass, inefficient primitive actions are removed from the execution trace segment; in the second, superfluous coordination mechanisms are removed. Inefficient primitive actions, from the standpoint of this work, are defined as sets of actions whose cumulative effect makes no *relevant* changes to the state of the world.

Figure 8 gives pseudo-code for determining if a sequence of primitive actions $\alpha_{1,}..., \alpha_n$ is inefficient. ISINEFFICIENTSEQUENCE builds upon methods for determining the set of relevant literals of an action and the set of predicates that are relevant to a set of literals. The default definition for RELEVANTLITERALS(α) is the set of non-location role-fillers for α (location literals are removed since co-presence is not, by itself, typically meaningful). RELEVANTPREDICATES$(literals, state)$ is the transitive closure of the union of all predicates that are true in $state$ and whose arguments intersect $literals$. The transitive closure is needed because when a relevant predicate is added to the union, its non-location arguments are added to the set of relevant literals.

ISINEFFICIENTSEQUENCE $(\langle \alpha_1, \ldots, \alpha_n \rangle) \equiv$
  $literals \leftarrow \bigcup_i$ RELEVANTLITERALS(α_i)
  $initialPreds \leftarrow$ RELEVANTPREDICATES$(literals,$ state immediately before $\alpha_1)$
  $finalPreds \leftarrow$ RELEVANTPREDICATES$(literals,$ state after before $\alpha_n)$
  if $initialPreds = finalPreds$
    return true
  else return false

Figure 8: Algorithm to determine if a sequence of primitive actions is inefficient.

Any primitive actions in the agent's segment that are part of an inefficient sequence will be removed from the segment. Exhaustive search of the possible inefficient action sequences requires checking 2^n subsets in the worst case, so it is impractical for traces with many actions. The implemented system tests contiguous pairs, which only requires $O(n)$ checks.

[8] For example, a lifter may load a box onto a hand-truck in order to free up her hands, but it serendipitously enables the hand-truck operator to get the box out to the street.

[9] Consider a segment to move a single box onto the truck involving loading the box onto, and later unloading it from, a hand-truck. If the lifter arrived at the street by carrying some other box, then the CARRY is not part of this segment (but may be part of a segment that achieves both boxes). Thus, the LOAD and the UNLOAD of the hand-truck are not separated by a MOVE as they would be if the sequence of actions were generated in a single planning session.

[10] The fact that the LOAD and UNLOAD take place in different locations is the only relevant difference between the pair of actions in the previous footnote and a similar pair that truly did reverse each other. Thus, in the absence of being able to both observe and reason about the hand-truck operator's actions, the lifter must compare the two locations, even though neither the LOAD nor the UNLOAD change the location of any items.

Superfluous coordination mechanisms (including communication) are identified based on the primitive actions still in the segment after the first pass. Superfluous mechanisms about joint action are easy to identify because of the close coupling that exists between the expectation and the joint action itself. If a related joint action (i.e., about the same coordination point) does not appear in the segment after the coordination mechanism, the mechanism is removed. If there are multiple mechanisms related to and preceding the same joint action, only the last mechanism is kept.

Determining whether a coordination mechanism about a request for service is superfluous is more complex. The basic idea is that the requestee must achieve the request and that the achievement of the request must enable future action by the requester. If it is a request for the agent, then there must exist an action in the segment that achieved the request and occurred after the request was made, and the request must have been the first one the agent received during this attempt at achieving the request. If it is a request made by the agent, then this request must have been the first one the agent made since the request was last true and, for sub-goal requests, there must be an action remaining in the agent's segment that started during the next time period in which the request was true.

In a dynamic multi-agent setting, removing all inefficient actions requires that an agent can infer when another agent (or an external force) undid the effect of some of her actions. However, in many domains this is not possible — an agent may not even have an accurate model of her own actions much less of other agents; also, she may not be aware of (i.e. have observed or been told about) all of the actions of other agents. Consequently, some inefficient actions may not be removed.

4.1.3 Adding coordinated procedures to the casebase

The second procedure called when updating the casebase of coordinated procedures is STORECOORDINATEDPROCEDURE, which converts an execution trace segment into a entry in memory. This conversion involves summarizing and optimizing the coordinated procedure, and determining appropriate storage indices. Indexing at storage time is primarily based on the top-level goals being achieved and expected coordination points (requests); indexing based on expected requests facilitates retrieving coordinated procedures during conversations.

A casebase entry contains two pieces of information: a context and a coordinated procedure. The stored context for a casebase entry is a set containing one record for each variable that must be bound in order to instantiate the goals, expected requests, and procedure. Each record includes the role the variable played in the coordinated procedure (i.e., the variables are typed), the observable features of the past bound values, and a set of relevant predicates. Relevant predicates are determined from the indexing state in the same manner as REMOVEINEFFICIENCIES.

Before presenting pseudo-code for STORECOORDINATEDPROCEDURE and getting into the details of how coordinated procedures are indexed in memory, we will describe the summarization and optimization criteria. The combined effect of these criteria is to remove an action if it is either planner-reconstructible or arguably unnecessary for agents acting from shared experience. The motivation for removing actions at this stage is to simplify the stored procedure and increase the frequency with which the retrieval context is generalized.

An important characteristic of summarization is that it keeps (1) coordination mechanisms for agreements and (2) actions that precede a shift between goals. The summarization principle is to remove steps that are in the planning support of another action in the coordinated procedure. Summarization does not remove any actions that cannot be regenerated by the planner in the correct position relative to the actions that are kept. Removing reconstructible actions can be effective since the particular time at which subordinate goals were achieved at runtime may be misleading and it makes the stored plan more easily adapted in the future (at the cost of regenerating the original action if it is needed again). Requests for joint action require special handling to ensure that the initiator only keeps the joint action and the assistant only remembers to expect a request.

The two optimization criteria are based on the (optimistic) belief that requests for service can be coordinated by implicit agreement when agents are acting from learned procedures.[11] To that end, agreements to satisfy requests for service are removed rather than converted into expectations. This criterion allows, for example, a lifter to learn to load the hand-truck without being explicitly told to do so. The second criterion is described below.

Figure 9 contains pseudo-code for STORECOORDINATEDPROCEDURE, which determines storage indices at the same time that the summarization and optimization criteria are applied (these criteria are encapsulated into the KEEP function). The indices for a new learned procedure are determined in a "roll-back" fashion. When an action is removed from the trace, the time at which it started is added to a list of indexing times; if it was an agreement, it is added to a list of expected requests that are also used for indexing. If an action is kept, these indexing lists are reset. Thus, a side effect of removing actions is that the procedure may be indexed differently in memory.

After passing over the trace and keeping the essential steps, the coordinated procedure and indexing information are passed into ADDCASEBASEENTRY. ADDCASEBASEENTRY is a complex procedure, which makes representational changes to the procedure and maintains the structure of the underlying casebase. Only two pieces of that procedure merit mention. The principle representational changes of ADDCASEBASEENTRY are to convert each agreement into an expectation for a request to be made and to consistently replace domain literals in the memory (including the procedure and the indexing and contextual information) by newly created variables. When representational changes are being made, the second optimization criterion — dual to the one described

[11] The criteria do not relate to requests for joint action, which require more fine-grained coordination.

```
STORECOORDINATEDPROCEDURE (segment, goals, CPCB) ≡
  procedure ← LIST()
  times ← LIST()
  expectedRequests ← LIST()
  forall step in REVERSE(segment)
    if KEEP(step, procedure)
      procedure ← PUSH(step, procedure)
      times ← LIST()
      expectedRequests ← LIST()
    times ← PUSH(STARTTIME(step), times)
    if step is an agreement
      expectedRequests ← PUSH(step, expectedRequests)
  if HEAD(procedure) is not a member of expectedRequests
    CPCB ← ADDCASEBASEENTRY(procedure, goals, null, times, CPCB)
  forall r in expectedRequests
    rTimes ← REMOVELATERTHAN(times, STARTTIME(r))
    CPCB ← ADDCASEBASEENTRY(procedure, goals, r, rTimes, CPCB)
  return CPCB
```

Figure 9: Algorithm to store a coordinated procedure, possibly under multiple indices.

above — is applied. This criterion is to mark expectations that led an agent to make a request for service so that the agent will prefer to coordinate the expectation in the learned procedure by implicit agreement. This allows the hand-truck operator to learn to expect a lifter to load the hand-truck without being asked to do so.

In terms of the organization of memory, a key division is that STORECOORDINATEDPROCEDURE only computes the times that correspond to states of the world that should be used as indexing states; ADDCASEBASEENTRY converts those states into an entry's contextual information. A casebase entry is generalized when a new case is indexed under the same goals and expected requests, the new and old procedures contain the same sequence of action types, and a one-to-one mapping can be constructed between the two contexts.[12] When generalizing previous context records, the list of relevant predicates shrinks to include only predicates present in both the new and the old case, and the range of seen values for each observable feature expands to include both the new and the old case.

4.1.4 Retrieving coordinated procedures from the casebase

When an agent is planning (either to achieve an unmet top-level goal or to accommodate a request from another agent), the agent consults her casebase of coordinated procedures. The basic idea is that the agent retrieves the most time-efficient procedure whose goals unifies with the current

[12] On average, procedures are indexed under eight different contexts when the agent is experienced.

planning goal, whose relevant literals can be filled, and whose stored context is sufficiently similar to the current setting.[13] Figure 10 gives pseudo-code for RETRIEVEFROMCPCB, the routine that implements retrieval.

```
RETRIEVEFROMCPCB (goals, state, CPCB) ≡
  highSimilarity ← 0.5
  plans ← LIST()
  forall ⟨case, maps⟩ in MATCHABLEGOALS(goals, CPCB)
    predicates ← LIST()
    forall literalRecord in CONTEXT(case)
      maps ← MATCHFEATURES(OBSERVABLEFEATURES(literalRecord), maps, state)
      if maps = nil then consider next case
      predicates ← predicates ∪ RELEVANTPREDICATES(literalRecord)
    ⟨similarity, bestMap⟩ ← COMPUTEMOSTSIMILAR(predicates, maps, state)
    if similarity ≥ highSimilarity
      plan ← APPLYMAP(PROCEDURE(case), bestMap)
      plan ← ADAPTPLAN(plan, state)
      if maps = nil then consider next case
      if similarity > highSimilarity
        highSimilarity ← similarity
        plans ← LIST()
      plans ← PUSH(plan, plans)
  if plans is empty
    return nil
  else return PREFERREDPLAN(plans)
```

Figure 10: Algorithm to retrieve casebase entries from memory.

The first step is to retrieve all entries whose index can be unified with the current planning goal, which may be a request. A byproduct of this unification performed by MATCHABLEGOALS is that each returned case will be associated with a set of maps that indicates possible ways to (partially) instantiate the data in the case.

For each matching casebase entry, the contextual information is then checked to see if each relevant literal has at least one match in the current setting based on observable features.[14] After each check, the set of maps associated with the case is updated by MATCHFEATURES to include the possible mappings for each literal. Next, COMPUTEMOSTSIMILAR determines which map produces a context that is most similar to the current surroundings. If the resulting similarity score meets or exceeds the highest similarity (that starts at an arbitrary threshold of 0.50), the coordinated

[13] This occurs approximately 20% of the time when the agent is experienced.

[14] A conservative, domain-independent criterion for matching is currently used: all of the observable features for a filler from the current setting must fall in the range of previously seen values for the literal.

procedure stored in the entry is checked to see if it can be adapted to the current state of the world after applying the map. If so, the current highest similarity is updated.

Finally, an entry is selected from the entries with maximal similarity by PREFERREDPLAN. In principle, this selection can be made by the number of storages for the entry, expected discounted reward, or any other preference criteria that are well-suited to the domain. Currently, plans that have the highest expected discounted reward are preferred and any remaining ties are broken randomly.

The implemented retrieval method incorporates two biases that are not shown in the pseudocode. The first is that MATCHABLEGOALS prefers casebase entries that achieve more unmet top-level goals. In most cases, this bias merely speeds retrieval (since fewer cases are identified as matches) rather than changes the return value (since the evaluation of the plan to achieve more goals is likely to be higher). The second is that MATCHFEATURES prefers the original filler for a literal over other possible fillers. This is mainly relevant for agent literals since the agent that assisted in the past is frequently present in the current setting.

In the experiments conducted so far, the number of entries in the casebase was never large enough to make storage or retrieval a bottleneck (the sum of the sizes of the casebases barely surpassed 100). In general, though, a more refined indexing scheme, such as indexing by differences [21, 22] or based upon the footprint metric [38] or based upon causal analysis [16] might be needed to keep retrieval times manageable. Further refinements such as distinguishing between short and long term memory or selectively "forgetting" past cases [34] may also be fruitful. Another deficiency in the current implementation is that the entries are not organized in a way that facilitates retrieving constituent parts of the coordinated procedure, cf. multicases [40] or snippets [20].

4.2 Learning operator probabilities

Agents acquire some coordination knowledge independently of learning coordinated procedures — agents learn more accurate estimates of the probability of success of possible actions. Agents use probabilities when planning from scratch, when deciding whether to cooperate, and when adapting coordinated procedures to match the current problem setting. The accuracy of the probability estimates, therefore, can have a large impact on how efficiently agents coordinate.

To get a feel for how accurate probability estimates can help an agent coordinate more efficiently, consider a common scenario a lifter faces. A lifter can lift some boxes alone, but not all of them. If a lifter can lift the box alone, it is more likely to be productive to do so because they do not have to spend time asking for assistance (and there is no guarantee the assistance will be given). On the other hand, if there is little chance that the lifter can handle the box on her own, it is a waste of time (and energy) to make the attempt. Ideally, individuals should be able to recognize which of these two possibilities is more likely and plan accordingly.

Initially, probabilities are set to 50%. This steers an inexperienced lifter to try to lift boxes alone because a plan to do so will be shorter (because there is no communication needed) and thus have a higher expected discounted reward. So, at the beginning of a problem, lifter L1 would try to pick up a large box LBOX1 by herself. L1 would fail; the next time she wants to lift LBOX1, L1 will decide to ask for the help of another agent. This decision is based upon L1's experience interacting with that particular box. However, L1's experience interacting with a particular box will not prevent her from attempting (and failing) to lift other large boxes alone.

Operator probability trees constitute the resource that enables an agent to increase the accuracy of her probability estimates by generalizing past interactions. Operator probability trees are based upon COBWEB [11] trees; slight changes to the classification algorithm are needed to output an estimate for successfully executing an action. In this paper, successfully executing an action means that after the attempt has been completed, all of the anticipated effects have occurred.

The successes or failure of an attempted action is stored in a COBWEB tree, along with all of the observable features of the various role fillers for the action. Actions might fail for features that are not explicitly represented (q.v. the *qualification problem* [27]) and COBWEB can handle such noisy data. Also, COBWEB trees can be updated incrementally, which allows the agents to learn during the course of their activity. The probability of success for a yet-to-be-attempted action is based on the similarity of the action to the node returned by COBWEB classification as well as the data stored in that node [12]. Throughout this paper, the phrase "learning operator probabilities" means all agents are maintaining operator probability trees and using them to estimate operator probabilities.

Returning to the MOVERS-WORLD scenario, if the observable characteristics (e.g., height, width, depth, material) of another box LBOX2 are identical to those of LBOX1, L1 will not attempt to lift LBOX2 alone. If the features are not identical and there are other experiences stored in the tree, L1's behavior depends on which experience the classification algorithm considers the best match with the current action (as well as how similar that experience is).

Operator probabilities could be maintained by a different machine learning technique than COBWEB. The features of past attempted actions, together with their outcome, form the training set; a potential future action is assigned a probability that the attribute "success" will be true given the action's other features. From this viewpoint, another on-line learning technique that can be used for prediction and is tolerant of noise could be substituted for COBWEB. An area of future research is to generalize the predicted attribute to predict a member of a set of possible operator effects. This would support actions that have conditional effects. In the extreme, it might be possible to learn both the members that make up the set and the probabilities associated with each member.

5 Results and analysis

This paper claims that groups of autonomous agents can solve coordination problems more efficiently when individuals independently learn from past experiences. This section supports that claim with an example of a learned procedure that is more efficient than first-principles plans and with rigorous empirical studies for an implemented test-bed. The empirical results demonstrate that learning substantially benefits groups whose members initially have minimal knowledge about how to solve coordination problems. Additional results show that groups with extensive initial knowledge also significantly improve their performance by learning coordinated procedures.

5.1 Learning more efficient procedures

This subsection shows how learning from an execution trace makes it possible to learn plans that are more efficient that those of her first-principles planner (or a traditional second-order planner based on it). In this example, lifter L2 learns to interleave working with the hand-truck with carrying a box to the street on her own. This is beyond the scope of the baseline planner when L2 is not designed with knowledge of the actions available to the hand-truck operator. The description of the activity given below has been organized to focus on L2's plans and actions.

1. L2 creates a first-principles 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 action (and hence rest of the plan) fails.

2. L2 creates a plan to get SBOX4 onto the truck. The plan consists of putting down XLBOX1 with L1's help and then lifting, carrying and loading SBOX4 onto the truck by herself. L2 is delayed in asking for L1's assistance because HTO calls L1 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 SBOX4 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 baseline planner. Nonetheless, L2 will store a single coordinated procedure with only three actions in it (showing the original literals instead of new variables for clarity):

```
<LOAD-TOGETHER XLBOX1 HANDTR2>
<LOAD SBOX4 TRUCK1>
<EXPECT L1 (EXPECT L2 (LOAD-TOGETHER XLBOX1 TRUCK1))>
```

In this run, L1 also learned a procedure to interleave two goals since L1 carried MBOX0 to the truck while L2 was handling SBOX4. Figure 11 shows a summary of how L1's behavior is converted into procedures in memory. In this figure, L1 is creating three casebase entries; two of them will be indexed with an expected request, one will not. The performance that led to learning this procedure was exceptional (33% fewer ticks than average) and was chosen for illustrative purposes; execution traces are normally much more chaotic.

```
    Time         Action                                                  Outcome

       1  <agreed to <LIFT-TOGETHER XLBOX1> with L2>                    Summarized
      21  <executed <LIFT-TOGETHER XLBOX1>>                             Summarized
      51  <agreed to <CARRY-TOGETHER XLBOX1 STREET1> with L2>           Cleaned
      71  <failed to <CARRY-TOGETHER XLBOX1 STREET1>>                   Cleaned
      91  <agreed to (ON XLBOX1 HANDTR3) for HTO>                       Optimized
     105  <L2 agreed to <LOAD-TOGETHER XLBOX1 HANDTR3>>                 Summarized
     126  <executed <LOAD-TOGETHER XLBOX1 HANDTR3>>                     Kept
     161  <executed <LIFT MBOX0>>                                       Summarized
     191  <executed <CARRY MBOX0 STREET1>>                              Summarized
     231  <executed <LOAD MBOX0 TRUCK3>>                                Kept
     266  <agreed to (ON XLBOX1 TRUCK3) for HTO>                        Optimized
     279  <L2 agreed to <UNLOAD-TOGETHER XLBOX1 HANDTR3>>               Summarized
     300  <executed <UNLOAD-TOGETHER XLBOX1 HANDTR3>>                   Summarized
     335  <L2 agreed to <LOAD-TOGETHER XLBOX1 TRUCK3>>                  Summarized
     353  <executed <LOAD-TOGETHER XLBOX1 TRUCK3>>                      Kept

 Casebase MEM75
Procedure: <LOAD-TOGETHER ?L1-177 ?L1-174>
           <LOAD ?L1-175 ?L1-176>
           <LOAD-TOGETHER ?L1-177 ?L1-176>
Top-level goals:  ((ON ?L1-177 ?L1-176)(ON ?L1-175 ?L1-176))
Request: NIL
State indices based on ticks 1, 21, 91, 105, 126

 Casebase MEM76 derived from MEM75
Request: ((LIFT-TOGETHER ?L1-177)) by ?L1-180
State indices based on tick 1

 Casebase MEM77 derived from MEM75
Request: ((ON ?L1-177 ?L1-174)) by ?L1-173
State indices based on ticks 1, 21, 91
```

Figure 11: Sample memories

5.2 Empirical results

The testbed system solves sequences of MOVERS-WORLD problems. 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. An area for future work is to investigate the performance of the group if agents in the group change over time or some agents learn at difference rates than others.

Experiments in the testbed take into account the possible influence of sampling bias and ordering effects. A test suite was created so that different runs of the system are solving problems

of equal difficulty. The test suite was composed of 60 problems; the goal of each problem was to move all boxes to the truck, where the number of boxes was uniformly distributed between 3 and 5. Each problem included exactly one extra-large box (whose observable features vary). To limit the influence of ordering effects, each learning curve shown is the average of running the system on ten predetermined groups of sequences. (It is not feasible to determine learning curves by running the system on all possible permutations of the test problems.) Each group is composed of 60 sequences such that each of the test problems occurs once as the first problem-solving episode of some sequence in the group, once as the second of a different sequence in the group, *et cetera*. In sum, each datum point reported is the average of 600 trials.[15]

There are many ways to measure the runtime performance of the agent community, such as the number of primitive actions attempted and the number conversations that occur. The best overall measure of community effort, however, is the number of ticks of a simulated clock that transpire during the course of the community solving the problem. This measure includes both action and communication effort, in addition to time when the agents are idle for one reason or another.

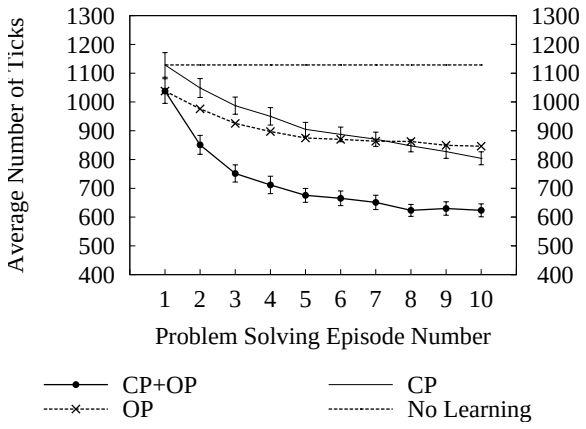

Figure 12: Overall effort

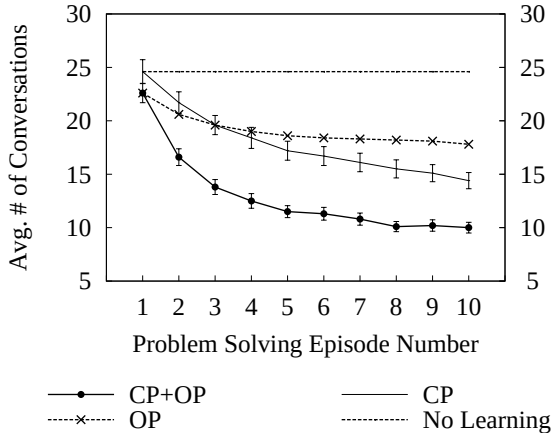

Figure 13: Communicative effort

Figure 12 shows that learning reduces the overall amount of effort, measured in ticks, required by the community to solve problems. Learning curves are shown for when agents are learning just coordinated procedures ("CP"), just operator probabilities ("OP"), or both ("CP+OP"). Also, a baseline ("No Learning") curve is included for comparison. On curves for runs where the agents learned coordinated procedures, 99% confidence intervals are given.

Both OP and CP lead to statistically significant improvements on their own. OP shows significant intra-problem learning, reducing the average number of ticks for solving the first problem of a sequence from 1128.8 to 1038.1. This number continues to decrease until it levels off at 846.3,

[15]Random seeds and decisions are recorded for each of these 600 trials and re-used across experiments to minimize the impact of randomness.

but learning after the fifth problem is not significant with 99% confidence. Initially, the CP curve falls more slowly, but it eventually undercuts the OP curve beginning with the eighth episode. The combined effect of learning coordinated procedures and operator probabilities has substantially more impact than either alone. After two problem problem-solving episodes, the community outperforms the best performance at any point in the sequence for either CP or OP. Improvement is significant until the seventh episode.

Figure 13 shows that acting from past experience in MOVERS-WORLD reduces communication. The average number of conversations required to solve these problems without learning was 24.6. Learning coordinated procedures leads to statistically significant improvement; the average number of conversations drops fairly steadily, ending at 14.4, a 41.5% improvement over the baseline system. OP is not as effective in reducing communication, dipping below 18 talks per problem for only the tenth problem. As with the number of ticks, the number of dialogs is much lower when the agents are learning both learning structures. By the third problem, fewer conversations occur (13.8) than at any time in the sequence for either OP or CP. The final value of 10.0 is a reduction of 14.6 conversations, slightly less than the combined reductions of OP (6.8) and CP (10.2).

Learning leads to similar statistical improvement in the number of ticks spent conversing, the number of attempted actions, and the number of ticks spent acting. Furthermore, these results hold across a wide spectrum of possible goal-selection strategies, cooperation strategies, and communication costs [12]. An analysis of how learning improves runtime performance identifies these advantages of learning coordinated procedures [12]:

1. Recalling shared past experience leads to compatible viewpoints on how to solve problems.
2. Optimizations lead to more efficient coordination.
3. Coordinated procedures implicitly acquire additional planning knowledge (e.g., lifters learn plans involving the hand-truck).
4. Learned goal-selection, coordination, and planning knowledge are more useful as a unit, when retrieved from the casebase, than when accessed separately.

Compatible viewpoints improve group performance by improving the quality of agents' goal selection and simplifying their decisions about whether to cooperate. Having compatible viewpoints means more than just working on the same goals at the same time — agents who are concurrently working on a shared goal must approach the goal in related ways. Determining whether agents have compatible viewpoints is determined during conversations; the listener is considered to have a compatible viewpoint if the listener's plan already includes a coordination mechanism related to the request.

When agents have compatible viewpoints, cooperation decisions require less reasoning. Many times a decision is trivial, since the listener was planning on helping anyway. In most other cases, the response of the listener is still a logical consequence of the relationship between the request and her plan. In the remaining cases, an agent is faced with a "hard" choice: she constructs a plan to satisfy the request and evaluates whether this plan is more likely to be productive than her current plan (by considering expected discounted reward).

Figure 14 measures how frequently two conversing agents have compatible viewpoints. In the first problem-solving episode, the listener has a related plan just 25.4% of the time. By the last problem-solving episode, neither learning just operator probabilities nor learning just coordinated procedures dramatically increases the frequency that agents have compatible viewpoints. However, the synergy between the two types of learning is clearly evident since the percentage of time that the listener has a compatible plan rises to 65.8%.

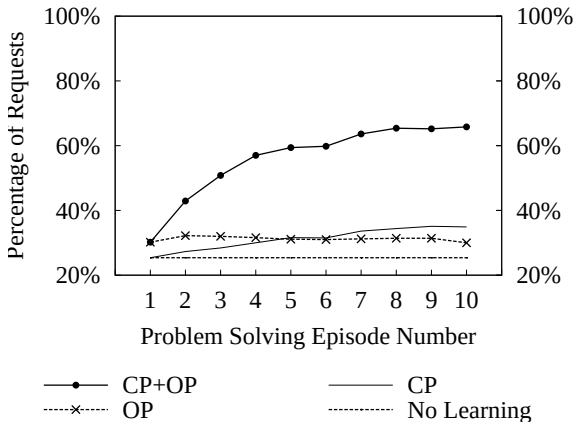

Figure 14: Related requests

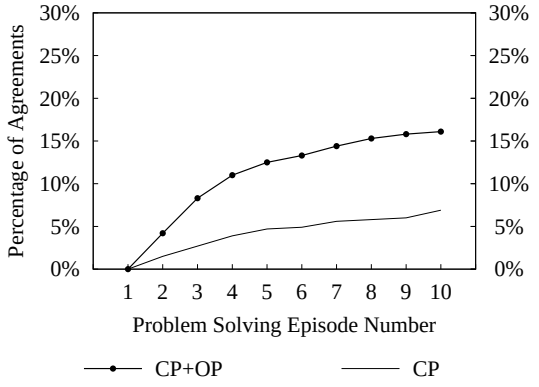

Figure 15: Requests avoided

Optimized procedures may contain expectations that are coordinated by *implicit agreement*. An implicit agreement occurs when an agent would normally explicitly make a request, but, based on past experience, will instead prefer to avoid communication. Measuring when communication is avoided involves comparing satisfied implicit agreements (SIA) and explicit agreements made during communication (EA). The percentage of conversations avoided is $SIA/(SIA+EA)$ since each time an implicit agreement is silently satisfied, the agents have, in effect, had a conversation in which the request was made and agreed-to. Figure 15 shows that the percentage of explicit conversations that were avoided rises steadily, exceeding 16% by the last problem-solving episode.

The disadvantage of coordinating by implicit agreement, which is not shown in a figure, is that sometimes the other agent will not be working from a compatible plan and the agent has to make an explicit request anyway. In this case, the period of time spent waiting was completely for naught. This occurred less than 0.8 times per problem-solving episode, on average, for all of the runs reported in this paper.

5.3 The benefits of coordinated procedures for experts

The results presented so far show that agents with minimal initial knowledge about how to solve coordination problems can benefit greatly from learning coordinated procedures. This subsection will show that learning coordinated procedures significantly improves community performance even for agents that are engineered with more than minimal knowledge. In order to address this issue, the experiments in this subsection isolate the advantages of learning coordinated procedures from the advantages of having accurate probability estimates. To this end, agents learn operator probabilities in all of the experiments. Thus, variations in the performance of the community in the last episode for different runs of the system are due to the coordinated procedures learned and/or the initial ability of agents to solve coordination problems.

	Procedures Learned?	
Built-in Ability	No	Yes
Minimal	846.3	623.5
Expert	687.4	594.1

Table 1: Comparison of the amount of community effort required for the tenth problem-solving episode. Agents learn operator probabilities in all of these runs.

Table 1 presents a tabular summary of the results of four runs of the system. To compare these numbers with the results in the previous subsection, "Minimal, Yes" in this table corresponds to CP+OP and "Minimal, No" corresponds to OP. The initial performance of the community without any learning is 1128.8. Agents with expert built-in coordination abilities have additional domain-dependent coordination knowledge. Expert agents are built with planning knowledge of other agents, coordinate all requests for service by implicit agreement, and use a hand-crafted heuristic for goal selection. Expert knowledge is designed to avoid conversations more frequently and, when discussion does occur, to have related requests more often.

The data in the "No" column of Table 1 indicate that improving the initial coordination ability of agents who do not learn coordinated procedures substantially reduces the number of ticks. Scanning across each row shows that, regardless of the initial coordination ability of the agents, learning basic coordinated procedures leads to significant reductions in the number of ticks.

The learning curves in Figure 16, plotted with their 99% confidence intervals, show the community runtime effort for three runs. Comparing curves MY and EN reveals that learning coordinated procedures is more effective than initial expertise by the second problem-solving episode. Another interesting result, shown by the MN and EN curves, is that learning operator probabilities alone does not overcome a disparity in the initial ability to solve coordination problems.

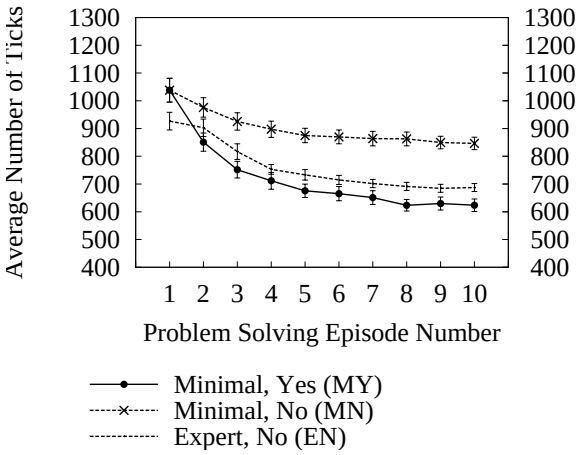

Figure 16: Comparing runtime effort

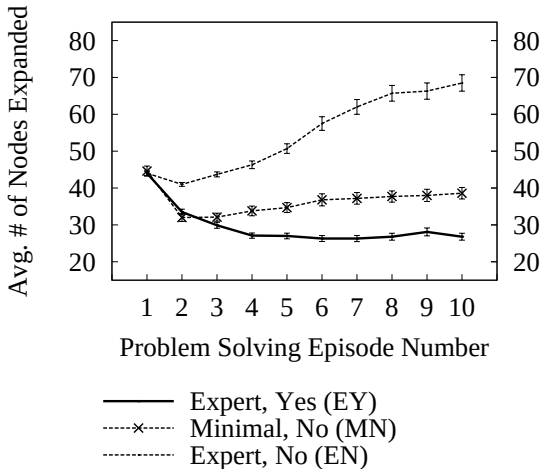

Figure 17: Comparing planner effort

Another signifi cant advantage of learning coordinated procedures is in controlling the amount of planner search. Figure 17 depicts planning effort, as measured by the average number of planning nodes expanded per call to the planner. The disparity between curves MN and EN provides evidence that initial expertise comes with a price — the increased amount of planning information can lead to a large increase in planner search. However, learning and acting from past experience controls planner search so effectively that experts who learn coordinated procedures (curve EY) expanded fewer planning nodes than agents with minimal initial knowledge who do not learn them (curve MN). CPU time, which includes the time agents spend maintaining their casebases, is also reduced signifi cantly by learning coordinated procedures.

Overall, the results clearly demonstrate that learned coordinated procedures are an effective resource for agents to better coordinate their runtime activities. Learning coordinated procedures benefi ts agents regardless of the initial ability to solve coordination problems. There is leverage in storing and retrieving plans based on the surface features of the environment, rather than having access to similar information that is accessed at separate times. Finally, the techniques are very effective in preventing increased planner search.

6 Discussion and related research

An alternative way to defi ne a solution to a coordination problem might be to specify a set of mental attitudes to be held by the group of agents. Three notable possibilities would be to say that a group of agents has solved a coordination problem once the agents' beliefs, desires, and intentions satisfy the defi nitions of a *shared plan* [13], *joint intentions* [26], or a *shared cooperative activity* [4]. This approach, however, would overlook the many sources of uncertainty that might prevent

such a group of agents from actually solving coordination problems. Incomplete or inaccurate domain models, incomplete or incorrect knowledge of the state of the world, exogenous events, the introduction of new agents or the failure of current ones are some of the practical reasons that mental attitudes alone cannot guarantee success. Coordination problems can only be solved by acting.

The generalized partial global planning (GPGP) architecture [7] is an alternative framework that supports the model of agent interactions studied in this paper. A *partial global plan* [10] is a general structure for representing coordinated activity in terms of goals, actions, interactions, and relationships. In the partial global planning architecture (PGP), agents rely on a meta-level organization that specifies the coordination roles of each agent in order to control how they exchange and reason about their possibly different partial global plans. GPGP is a domain-independent coordination technique that extends PGP by communicating more abstract information and separating the process of coordination from individual planning. A key to GPGP is that coordination relationships are abstractly defined, so coordination mechanisms can be invoked based on the detection of certain features in the environment or task. Coordination mechanisms are tailored *a priori* to create a fixed coordination strategy. Once created, the strategy is used across all problem instances in the environment, all agents in the community have common knowledge of the coordination strategy, and the agents exchange meta-level information throughout the activity to coordinate effectively.

In our work, agents do not exchange the meta-level information that forms the heart of the GPGP "augmented goals". Namely, agents do not communicate about the current projected result, the physical resources needed, timing information, the capabilities of an agent, or any measures of probability or certainty. Another differentiating feature is that our framework does not rely on pre-determined high-quality coordination strategies; instead, agents can start with simple strategies and learn more advanced ones that reflect what works in the environment.

COLLAGE [31] leverages case-based reasoning techniques to improve coordination in the GPGP framework. COLLAGE builds upon the basic GPGP approach by allowing the agents to tailor the coordination strategy for each problem, based on the results of training runs. There are three key differences between this work and COLLAGE. First, COLLAGE learns to choose a subset of the pre-defined coordination mechanisms rather than learning ways to coordinate that were not pre-defined. Second, learning in COLLAGE happens during a separate training phase rather than on-line. Finally, all agents select the same strategy because each agent communicates her local viewpoint to all other agents to form a consistent global viewpoint, and each agent records the same training data. In our work, learning is effective without a global viewpoint even when local viewpoints differ or when agents have varying amounts of training.

Case-based reasoning has been used to change runtime behavior in other systems with multiple agents. In particular, Haynes & Sen [17] keep a record of past execution-time conflicts in

a communication-free domain to supplement control strategies in order to reduce conflicts in the future. Ohko, Hiraki, and Anzai [30] learn to allocate tasks better among a community communicating via a contract net by storing the outcome of past bid and task announcements.

Coordinated procedures are culled primarily from execution traces. Others [5, 38, 25] have argued that an agent should store planning histories in memory. However, reusing a plan derivation will not produce a sequence of actions to solve a similar problem better if the derivation was deficient (due to the agent's incomplete knowledge). On the other hand, execution traces encapsulate the history of both planned and unplanned agent interactions with the domain. Consequently, procedures are learned that were not developed in a single (or multiple) planning histories.

Thus, storing coordinated procedures emphasizes learning from (unplanned) successes, whereas previous approaches have emphasized learning from failures. For example, Hammond [15] focused on learning to anticipate and avoid problems through the explanation of past execution failures in a framework that is exclusively case-based. In that model, runtime failures are not repaired at the time; instead, the explanation knowledge is stored so that plans generated later will execute without failure.

In this paper, the casebase contains procedures that are outside the scope of plans that can be generated using first-principles search control knowledge. Thus, one could say that the casebase implicitly contains learned search control knowledge. However, the techniques presented in this paper are not explanation-based learning (EBL) [28, 8]. As with reusing plan derivations, EBL is traditionally geared toward planning efficiency rather than knowledge acquisition. While one might imagine a variant of EBL that would change the output of the baseline planner based on runtime experience, our techniques would be more general since we do not assume that there is a known domain theory that can be operationalized. Suguwara & Lesser [36] present EBL techniques to learn coordination rules based on past failures in order to select and prioritize future diagnostic activities in a homogeneous multi-agent network monitoring system.

7 Conclusion

Solving coordination problems efficiently is a fundamental difficulty faced by groups of interacting agents. The most basic hurdle to efficient coordination is when agents are not aware of their common interests; another is overcoming a distribution of either execution ability or problem-solving knowledge. Even when agents have common knowledge about goals and planning, there may be ambiguity about the order on which to work on the goals. Finally, agents may have different preferences among alternative solutions when there is uncertainty about the outcome of actions or when agents have different beliefs about the world.

Using the techniques presented in this paper, an agent learns from her experiences and the group moves from satisficing solutions to coordination problems toward optimal ones. In order to recall practical solutions to coordination problems, agents learn coordinated procedures from execution traces and store them into a casebase that is organized around expectations about other agents. Agents also learn better estimates for how likely individual actions are to succeed in order to improve the quality of decisions when planning, communicating, and adapting plans.

It is a novel approach for individuals to learn procedures as a means for the group to coordinate more efficiently. In addition, this work is distinct from prior procedural learning research in several ways. Analyzing execution trace segments is more difficult than analyzing the output of a first principles planner because trace segments may be noisy, inefficient, and incomplete. Unanticipated requests and responses that occur during communication allow an agent to acquire coordination knowledge about other agents, and constitute the building blocks of learned coordinated procedures. Indexing casebase entries based on expected requests facilitates retrieving coordinated procedures during conversations.

The learning techniques do not presume that agents have initial expertise about how to solve coordination problems. Further, the model of joint activity presented in this work encompasses autonomous agents who are heterogeneous, who do not have complete built-in common knowledge, and who cannot coordinate solely by observation. As a result, this model is better suited for open-world environments than previous research on coordination in multi-agent systems.

Overall, the results clearly demonstrate that the learning techniques enable agents to better coordinate their activities. Learning coordinated procedures reduces action effort, communicative effort, and planning effort, whether or not the agents have initial expertise in solving coordination problems. Finally, there is leverage in storing and retrieving learned plans as a unit, rather than having access to similar information that is accessed at separate times.

A Expected discounted reward

This appendix describes the *expected discounted reward* of a plan, which is a measure of how time-efficient the plan is. The expected discounted reward for a plan is used when planning from scratch, when making rational cooperation decisions, and when selecting among coordinated procedures in memory.

The formulae in Figure 18 determine $EDR(P)$, which is the weighted average discounted reward for a plan P that is comprised of n actions, $\alpha_1, ..., \alpha_n$, to achieve goals G given an initial planning state S_0. ($MinReward$ is set to 0.01 when called by the baseline planner so that $EDR(P) > 0$ for a partial plan P.)

$$
\begin{aligned}
Prob_i &= Prob_{i-1} * Probability(\alpha_i \text{ will succeed}), \qquad Prob_0 = 1 \\
Dur_i &= Dur_{i-1} + Duration(\alpha_i, \text{ if } \alpha_i \text{ succeeds}), \qquad Dur_0 = 0 \\
S_i &= \text{Expected state after executing } \alpha_1, ..., \alpha_i \\
Reward_i &= Max(MinReward, \text{ the number of goals in } G \text{ that are true in } S_i) \\
EDR(P) &= \frac{Reward_n * Prob_n}{Dur_n} + \sum_{i=1}^{n} \frac{Reward_{i-1} * Prob_{i-1} * Probability(\alpha_i \text{ will fail})}{Dur_{i-1} + Duration(\alpha_i, \text{ if } \alpha_i \text{ fails})}
\end{aligned}
$$

Figure 18: Formulae to compute the expected discounted reward of a plan P.

An agent can create a plan from scratch using a given set of actions in a hierarchical fashion (cf. [32, 23]). The action descriptions are traditional; each action type is associated with a set of preconditions (that include type checks on the action's arguments) and anticipated effects. The search heuristic is based upon an agent's current probability estimates, so the agent produces plans that are more likely to succeed during the course of action. For any given planning session, the probabilities can be treated as fixed and the search algorithm is a version of best-first search. The baseline planner maintains a list of planning objects that are sorted according to estimates of the quality of the final plans that will be derived from the objects.

For each planning object O, $EDR(O)$ is an (under-)estimate of the expected discounted reward of the final plan to be produced by O. Since planning is hierarchical, each precondition of an action is associated with a planning level at which it should be made true. Each planning object O is associated with a planning level $level(O)$, a partial plan $Plan(O)$ to achieve a set of goal predicates G, a goal-stack, an expected discounted reward $EDR(O)$, a state $State(O)$, and a parent planning object $Parent(O)$. The plan and goal-stack may contain both actions and goals, including subgoals that are preconditions of actions. There are three ways in which it differs from the expected discounted reward of a plan. One difference is that $Plan(O)$ may contain unmet preconditions (with criticality levels $< level(O)$) that will eventually force an action to be added to the plan. Another is that there may be actions in the goal-stack that will eventually become part of the plan (actions are put in the goal-stack when a planning object "descends" to the next lowest planning level). Accounting for eventual additions to the plan makes the expected discounted reward for O more accurate. The third difference is that to encourage — rather than legislate — planning objects to be explored in a hierarchical manner, the expected discounted reward is divided by the current planning level of O.

Figure 19 provides the formulae to compute $EDR(O)$, which is derived from the expected discounted reward of O's plan. When computing $EDR(Plan(O))$, the formula to compute Dur_i from Figure 18 is replaced by the one in Figure 19 (so α_i refers to an action in $Plan(O)$, not the

$$Unmet_i = \text{the number of unmet preconditions in } Plan(O) \text{ that precede } \alpha_i$$

$$Dur_i = Dur_{i-1} + Duration(\alpha_i, \text{ if } \alpha_i \text{ succeeds}) + MaxDuration * Unmet_i$$

$$EDR(O) = \frac{EDR(Plan(O)) + \frac{Max(1, Reward_n) * Prob_n * \prod_{\alpha} Prob(\alpha \text{ will succeed})}{Dur_n + \sum_{\alpha} Duration(\alpha, \text{ if } \alpha \text{ succeeds})}}{level(O)}$$

Figure 19: Heuristic to estimate expected discounted reward for a planning object O.

goal-stack) . This treats unmet preconditions in $Plan(O)$ as actions with probability of success 1.0 and duration of $MaxDuration$ (which is 40 for MOVERS-WORLD). $Prob_n$ and $Reward_n$ refer to the partial results generated when computing $EDR(Plan(O))$, where n is the length of $Plan(O)$. The acts α included in the product and sum in the equation for $EDR(O)$ are the actions in the goal-stack, not $Plan(O)$. The reward for the second term in the numerator is at least 1 because a returned plan will accomplish at least one top-level goal.

The baseline planner starts with a queue containing a single planning object that has level $MaxLevel$ (4 for MOVERS-WORLD), an empty plan, and a goal-stack containing only the top-level planning goal(s). In each loop, if the queue is empty, no plan has been found and planning terminates. Otherwise, the planning object O with the highest expected discounted reward is removed from the queue. If the goal-stack of O is empty and $level(O)$ is 1, the loop is terminated and the plan in O is returned as the result. If the goal-stack of O is empty and $level(O)$ is greater than 1, a new object is created, and added to the queue, that has a goal-stack equal to O's plan, an empty plan, and whose criticality level is $level(O) - 1$. If the top of the goal-stack is an action and all of its preconditions of criticality level greater than or equal to $level(O)$ are true in $State(O)$, it is moved to the plan. Otherwise, the unmet preconditions are inserted onto the goal-stack. If the top of the goal-stack is a predicate that is true or whose criticality level is less than $level(O)$, it is moved to the plan. Otherwise, a set of children planning objects are created and added to the queue; one child is created for each action in the domain that may achieve the unmet predicate.

At planning time, an agent is content to construct a plan containing expectations about impasse conditions. When an unmet precondition is an impasse condition, then the action added to the plan is an expectation about the condition. The role-fi ller of the enabler is determined in the same manner as other unbound role-fi llers — by a subordinate search that seeks a bound action with the highest expected discounted reward. This is not an anytime planning algorithm, so the agent must complete her search for a plan before acting; if no plan is found, the agent will either do nothing or respond to a request during the next tick.

References

[1] Alterman, R.: 1986, 'An Adaptive Planner'. In: *Proc. Fifth National Conference on Artificial Intelligence*. pp. 65–69. Reprinted in *Readings in Planning*; Morgan Kaufmann, publishers; 1990.

[2] Alterman, R.: 1988, 'Adaptive Planning'. *Cognitive Science* **12**, 393–421.

[3] Alterman, R. and A. Garland: 2001, 'Convention in Joint Activity'. *Cognitive Science* **25**(4), 611–657.

[4] Bratman, M. E.: 1999, 'Shared Cooperative Activity'. In: *Faces of Intention*. Cambridge University Press, pp. 93–108.

[5] Carbonell, J.: 1983, 'Derivational analogy and its role in problem solving'. In: *Proc. Third National Conference on Artificial Intelligence*. pp. 64–69.

[6] Clark, H. H.: 1996, *Using Language*. Cambridge University Press.

[7] Decker, K. S. and V. R. Lesser: 1992, 'Generalized Partial Global Planning'. *Intl. Journal of Intelligent and Cooperative Information Systems* **1**(2), 319–346.

[8] DeJong, G. and R. Mooney: 1986, 'Explanation-based Learning: An Alternative View'. *Machine Learning* **1**(2), 145–176.

[9] desJardins, M. E., E. H. Durfee, C. L. Ortiz, and M. J. Wolverton: 1999, 'A Survey of Research in Distributed, Continual Planning'. *AI Magazine* **20**(4), 13–22.

[10] Durfee, E. H. and V. R. Lesser: 1991, 'Partial global planning: A coordination framework for distributed hypothesis formation'. *IEEE Transactions on Systems, Man, and Cybernetics* **21**(5), 1167–1183.

[11] Fisher, D. H.: 1987, 'Knowledge Acquisition via Incremental Conceptual Clustering'. *Machine Learning* **2**, 139–172.

[12] Garland, A.: 2000, 'Learning to Better Coordinate in Joint Activities'. Ph.D. thesis, Brandeis University.

[13] Grosz, B. and C. Sidner: 1990, 'Plans for discourse'. In: P. R. Cohen, J. Morgan, and M. E. Pollack (eds.): *Intentions in Communication*. Bradford Books, pp. 417–444.

[14] Hammond, K. J.: 1986a, 'CHEF: A Model of Case-Based Planning'. In: *Proc. Fifth National Conference on Artificial Intelligence*. pp. 267–271.

[15] Hammond, K. J.: 1986b, 'Learning to Anticipate and Avoid Planning Problems through the Explanation of Failures'. In: *Proc. Fifth National Conference on Artificial Intelligence*. pp. 556–560.

[16] Hammond, K. J.: 1990, 'Case-Based Planning: A Framework for Planning from Experience'. *Cognitive Science* **14**, 385–443.

[17] Haynes, T. and S. Sen: 1998, 'Learning Cases to Resolve Conflicts and Improve Group Behavior'. *Intl. Journal of Human-Computer Studies* **48**, 31–49.

[18] Ho, F. and M. Kamel: 1998, 'Learning Coordination Strategies for Cooperative Multiagent Systems'. *Machine Learning* **33**(2-3), 155–177.

[19] Kambhampati, S. and J. A. Hendler: 1992, 'Control of Refitting During Plan Reuse'. *Artificial Intelligence* **55**, 193–258.

[20] Kolodner, J.: 1988, 'Retrieving events from a case memory: A parallel implementation'. In: J. Kolodner (ed.): *Case-based Reasoning Workshop*. San Mateo, CA.

[21] Kolodner, J. L.: 1983a, 'Maintaining Organization in a Dynamic Long-Term Memory'. *Cognitive Science* **7**, 243–280.

[22] Kolodner, J. L.: 1983b, 'Reconstructive memory: A computer model'. *Cognitive Science* **7**, 281–328.

[23] Kushmerick, N., S. Hanks, and D. Weld: 1995, 'An Algorithm for Probabilistic Planning'. *Artificial Intelligence* **76**.

[24] Labrou, Y. and T. Finin: 1997, 'Proposal for a new KQML Specification'. Technical Report CS-97-03, University of Maryland Baltimore County.

[25] Laird, J. E., P. S. Rosenbloom, and A. Newell: 1986, 'Chunking in SOAR: The Anatomy of a General Learning Mechanism'. *Machine Learning* **1**, 11–46.

[26] Levesque, H. J., P. R. Cohen, and J. H. T. Nunes: 1990, 'On Acting Together'. In: *Proc. Eighth National Conference on Artificial Intelligence*. pp. 94–99.

[27] McCarthy, J.: 1977, 'Epistemological problems in Articial Intelligence'. In: *Proc. Fifth Intl. Joint Conference on Artificial Intelligence*.

[28] Mitchell, T., R. Keller, and S. Kedar-Cabelli: 1986, 'Explanation-based generalization: A unifying view'. *Machine Learning* **1**, 47–80.

[29] NagendraPrasad, M. V., V. R. Lesser, and S. Lander: 1995, 'Retrieval and Reasoning in Distributed Case Bases'. Technical Report CS TR 95-27, University of Massachusetts.

[30] Ohko, T., K. Hiraki, and Y. Anzai: 1996, 'Learning to Reduce Communication Costs in Task Negotiation among Multiple Autonomous Mobile Robots'. In: G. Weiß and S. Sen (eds.): *Adaptation and Learning in Multi-Agent Systems*, Lecture Notes in Artificial Intelligence. Berlin: Springer-Verlag, pp. 177–190.

[31] Prasad, M. V. N. and V. R. Lesser: 1999, 'Learning Situation-Specific Coordination in Cooperative Multi-agent Systems'. *Autonomous Agents and Multi-Agent Systems* **2**, 173–207.

[32] Sacerdoti, E. D.: 1975, 'The Nonlinear Nature of Plans'. In: *Proc. Fourth Intl. Joint Conference on Artificial Intelligence*. pp. 206–214.

[33] Schelling, T. C.: 1963, *The strategy of conflict*. New York, NY: Oxford University Press. First published in 1960.

[34] Smyth, B. and M. T. Keane: 1995, 'Remembering To Forget'. In: *Proc. Fourteenth Intl. Joint Conference on Artificial Intelligence*. pp. 377–382.

[35] Suchman, L. A.: 1987, *Plans and Situated Actions*. Cambridge: Cambridge University Press.

[36] Sugawara, T. and V. Lesser: 1998, 'Learning to Improve Coordinated Actions in Cooperative Distributed Problem-Solving Environments'. *Machine Learning* **33**(2-3), 129–153.

[37] Tambe, M.: 1997, 'Towards Flexible Teamwork'. *Journal of Artificial Intelligence Research* **7**, 83–124.

[38] Veloso, M. and J. Carbonell: 1993, 'Derivational Analogy in PRODIGY: Automating Case acquisition, storage, and utilization'. *Machine Learning* **10**, 249–278.

[39] Vidal, J. M. and E. H. Durfee: 1995, 'Recursive Agent Modeling Using Limited Rationality'. In: *Proc. First Intl. Conference on Multiagent Systems*. pp. 376–383.

[40] Zito-Wolf, R. and R. Alterman: 1992, 'Multicases: A case-based representation for procedural knowledge'. In: *Proc. Fourteenth Annual Conference of the Cognitive Science Society*. pp. 331–336.