Planning for Interactions among Autonomous Agents

Tsz-Chiu Au, Ugur Kuter, and Dana Nau

University of Maryland, College Park, MD 20742, USA

Abstract. AI planning research has traditionally focused on offline planning for static single-agent environments. In environments where an agent needs to plan its interactions with other autonomous agents, planning is much more complicated, because the actions of the other agents can induce a combinatorial explosion in the number of contingencies that the planner will need to consider. This paper discusses several ways to alleviate the combinatorial explosion, and illustrates their use in several different kinds of multi-agent planning domains.

1 Introduction

AI planning research has traditionally focused on offline planning for static single-agent environments. In environments where an agent needs to plan its interactions with other autonomous agents, planning is much more complex computationally: the actions of the other agents can induce a combinatorial explosion in the number of contingencies that the planner will need to consider, making both the search space and the solution size exponentially larger.

This paper discusses several techniques for reducing the computational complexity of planning interactions with other agents. These include:

- Partitioning states into equivalence classes, so that planning can be done over these equivalence classes rather than the individual states. In some cases this can greatly reduce both the size of the search space and the size of the solution.
- Pruning unpromising parts of the search space, to avoid searching them. This can reduce complexity by reducing how much of the search space is actually searched.
- Online planning, i.e., interleaving planning and execution. This can enable the planner to avoid planning for contingencies that do not arise during plan execution.

Each of these techniques has strengths and drawbacks. To illustrate these, the paper includes case-studies of two different multi-agent planning domains: the Hunter-and-Prey domain, and a noisy version of the Iterated Prisoner's Dilemma.

2 Background

This section very briefly describes some relevant concepts from AI planning. For a much more detailed description, see [1].

2.1 AI Planning in General

Figure 1 shows a conceptual model of AI planning. The three components include (1) the *planner*, (2) the *plan-execution agent*, and (3) the *world* Σ in which the plans are to be executed.

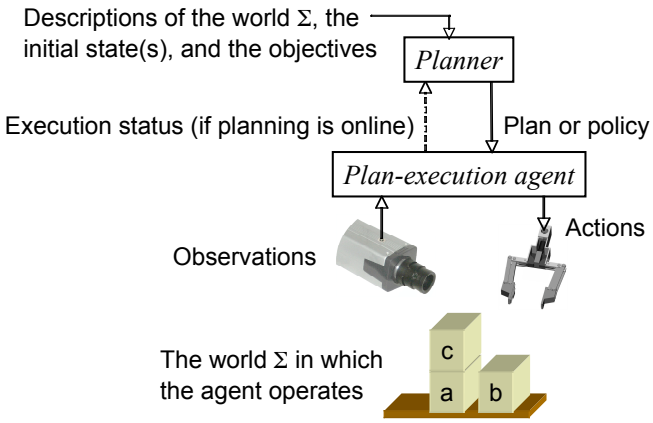

Fig. 1. Conceptual model of AI planning

The planner's input includes descriptions of Σ, the *initial* state(s) that Σ might be in before the plan-execution agent performs any actions, and the desired objectives (e.g., to reach a set of states that satisfies a given *goal condition*, or to perform a specified task, or a set of states that the world should be kept in or kept out of, or a partially ordered set of states that we might want the world to go through). If the planning is being done online (i.e., if planning and plan execution are going on at the same time), the planner's input will also include feedback about the current execution status of the plan or policy.

The planner's output consists of either a *plan* (a linear sequence of actions for the agent to perform) or a *policy* (a set of state-action pairs with at most one action for each state).

2.2 Classical Planning

Historically, most AI planning research has focused on *classical* planning problems. A classical planning problem is one that satisfies a very restrictive set of assumptions:

1. **State-transition model.** The world is a finite state-transition system, i.e., a triple $\Sigma = (S, A, \gamma)$, where S is a finite set of states, A is a finite set of actions, $\gamma : S \times A \rightarrow 2^S$ is a state-transition function. If $\gamma(s, a) \neq \emptyset$ then we say that a is *applicable* to s or *executable* in s.
2. **Full observability.** Σ's current state is always completely knowable.
3. **Determinism.** For every s and a, $|\gamma(s, a)| \leq 1$. In other words, if a is applicable to s, then there is exactly one possible outcome, namely the state in $\gamma(s, a)$. Furthermore, there is exactly one *initial state* s_0 that will be Σ's current state before plan-execution begins.
4. **Single agency.** The plan-execution agent is the only agent capable of making any changes in the world. If it were not for this agent's actions, the world would be static.
5. **Achievement goals and sequential plans.** The planner's objective is to produce a plan (i.e., a linearly ordered finite sequence of actions) that puts Σ into any one of some finite set of states S_g.
6. **Implicit time.** Actions have no duration; they are instantaneous state transitions.
7. **Offline planning.** The planner produces a complete plan for the given initial and goal states prior to any execution of its plan by the plan-execution agent.

In multi-agent systems, Assumption 4 does not hold, and several of the other assumptions may not necessarily hold. Sections 3 and 4 describe two generalizations of classical planning that can be used to represent certain kinds of multi-agent planning problems.

2.3 Classical Representation

A classical planning problem is conventionally represented as a triple $P = (O, s_0, g)$, where:

- s_0 and g, the *initial state* and *goal condition*, are sets of ground atoms in some first-order language L.
- O is a set of *planning operators*, each of which represents a class of actions that the plan-execution agent may perform. An operator is conventionally represented as a triple

$$o = (\text{head}(o), \text{precond}(o), \text{effects}(o)),$$

  where precond(o) is a collection of literals called *preconditions*, effects(o) is a collection of literals called *effects*, and head(o) is a syntactic expression of the form $name(x_1, \ldots, x_n)$, where *name* is a symbol called o's *name*, and $x_1, \ldots, x_n$ are all of the variables that appear anywhere in precond(o) or effects(o). We will let effects$^+$(o) be the set of all non-negated atoms in effects(o), and effects$^-$(o) be the set of all atoms whose negations are in effects(o).

A *state* is any set s of ground atoms of L. An atom l is *true* in s if $l \in s$; otherwise l is *false* in s. The set of goal states is $S_g = \{s : s \text{ is a state and } g \text{ is true in } s\}$.

An *action* is any ground instance of a planning operator. An action a is *applicable* to a state s if a's preconditions are true in s, i.e., if $l \in s$ for every positive literal $l \in \text{precond}(a)$ and $l \notin s$ for every negated literal $\neg l \in \text{precond}(a)$. If a is applicable to s, then the *result* of applying it is the state $\gamma(s, a)$ produced by removing from s all negated atoms in effects(a), and adding all non-negated atoms in effects(a). Formally,

$$\gamma(s, a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a).$$

A *plan* is a linear sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$. The plan π is *executable* in a state s_0 if there is a sequence of states $\langle s_0, s_1, \ldots, s_n \rangle$ such that for $i = 1, \ldots, n$, $s_i = \gamma(s_{i-1}, a_2)$. In this case we say that $\langle s_0, s_1, \ldots, s_n \rangle$ is π's *execution trace* from s_0, and we define $\gamma(s_0, \pi) = s_n$. If s_n satisfies the goal g, then we say that π is a *solution* for the planning problem $P = (O, s_0, g)$.

3 Nondeterministic Planning Problems and Multi-Agency

A *nondeterministic* planning problem is one in which Assumption 3 does not hold. Each action may have more than one possible state-transition; and instead of a single initial state s_0, there is a set S_0 of possible initial states.

The classical representation scheme can be extended to model nondeterministic planning problems, by redefining a *nondeterministic operator* to be a tuple

$$o = (\text{head}(o), \text{precond}(o), \text{effects}_1(o), \text{effects}_2(o), \ldots, \text{effects}_n(o)),$$

where each $\text{effects}_i(o)$ is a set of literals. If a is a ground instance of o and precond(a) is true in a state s, then the result of executing a in s may be any of the states in the following set:

$$\begin{aligned} \gamma(s, a) = \{&(s - \text{effects}_1^-(a)) \cup \text{effects}_1^+(a), \\ &(s - \text{effects}_2^-(a)) \cup \text{effects}_2^+(a), \\ &\ldots, \\ &(s - \text{effects}_n^-(a)) \cup \text{effects}_n^+(a)\}. \end{aligned}$$

A nondeterministic planning problem can be represented as a triple $P = (O, S_0, g)$, where O is a set of nondeterministic planning operators, S_0 is the set of initial states, and g is the goal condition.

3.1 Representing Other Agents' Actions

Multi-agent planning problems can sometimes be translated into nondeterministic single-agent planning problems by modifying the plan-execution agent's actions to incorporate the effects of the other agents' possible responses to those

actions. For example, suppose an agent α is going down a hallway and runs into another agent β going in the opposite direction. Suppose that to get past them, α moves to its right. Then β may either move right (in which case the agents can pass each other) or left (in which case neither agent can pass). As shown in Figure 2, β's two possible actions can be modeled as nondeterministic outcomes of α's move-right action.

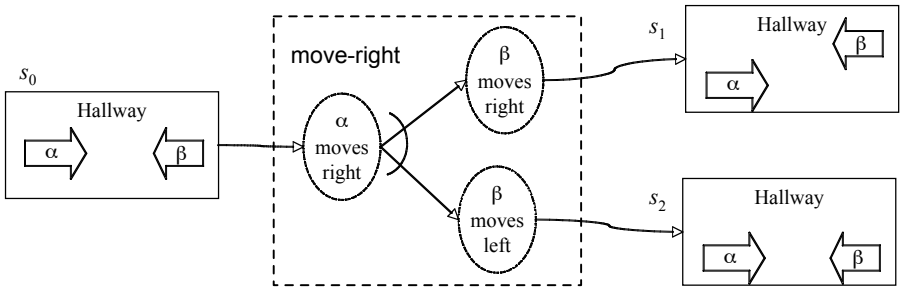

Fig. 2. Two possible outcomes of moving to the right in a hallway.

3.2 Policies and Execution Structures

Most nondeterministic planning problems violate not only Assumption 3 but also Assumption 5, for if an action a can lead to more than one possible state, then we will need a way to provide conditional execution of subsequent actions depending on what state a takes us to. Hence for nondeterministic planning problems, solutions are typically defined to be *policies* rather than plans. A policy is a set π of state-action pairs such that for each state there is at most one action. In other words, π is a partial function from S into A.

Given a policy π, the *execution structure* Σ_π is a graph of all possible execution traces of π in the system Σ. For example, the policy

$$\pi_0 = \{(s_0, \mathsf{move\text{-}right}), (s_1, \mathsf{pass}), (s_2, \mathsf{wait})\}$$

has the execution structure depicted in Figure 3.

3.3 Solutions

Recall that in a classical planning problem, the execution of a plan π in a state s always terminates at a single state $\gamma(s, \pi)$, and π is a solution to the planning problem if $\gamma(s, \pi)$ is a goal state. In nondeterministic planning problems, the execution of a policy π in a state s may terminate at any of several different states or might not terminate at all. Hence we can define several different kinds of solutions to nondeterministic planning problems, depending on which of the executions terminate at goal states [2]:

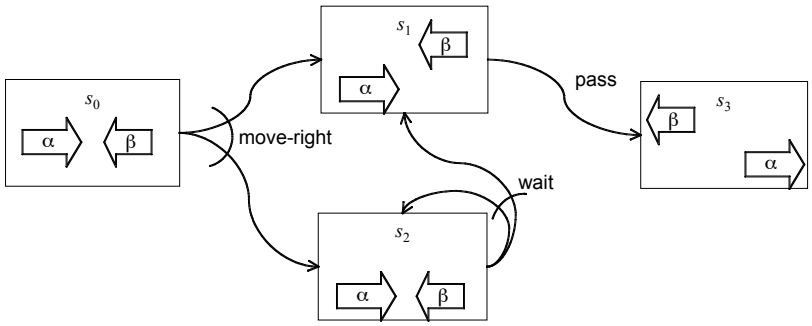

Fig. 3. Execution structure for the policy $\pi_0 = \{(s_0, \mathsf{move\text{-}right}), (s_1, \mathsf{pass}), (s_2, \mathsf{wait})\}$.

- **Weak solutions.** π is a *weak solution* for P if for every state $s \in S_0$ there is at least one execution trace of π that takes us to a goal state, i.e., if for every $s \in S_0$, there is at least one path in Σ_π from s to a state in S_g. For example, if the set of possible initial states in Figure 3 is $S_0 = \{s_0\}$ and if α's goal is to get to the state s_3 in the hallway, then the policy π_0 given earlier is a weak solution, because there exists an execution trace of π_0 that produces s_3. The same is true for the policy

$$\pi_1 = \{(s_0, \mathsf{move\text{-}right}), (s_1, \mathsf{pass})\}.$$

- **Strong solutions.** π is a *strong solution* for P if every execution trace of π produces a goal state, i.e., every leaf node of Σ_π is in S_g. For example, consider a modified version of the hallway problem in which β will *always* move to the right in the state s_2. In this version of the hallway problem, the execution structure for π_0 is not the one shown in Figure 3, but instead is the one shown in Figure 4. Hence π_0 is a strong solution.

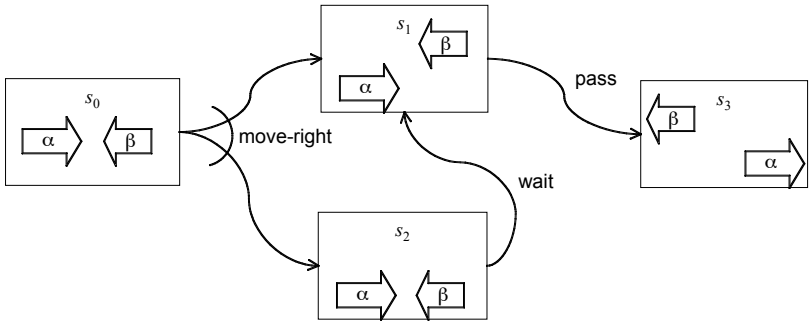

Fig. 4. π_0's execution structure if β *always* moves to the right in s_2.

– **Strong-cyclic solutions.** π is a *strong-cyclic solution* for P if every *fair* execution trace of π takes us to a goal state. A fair execution trace is one such that for every cycle C in which there is an action having more than one outcome, the execution trace will traverse C at most finitely many times before exiting C.

  The concept of a fair execution trace can be understood intuitively as follows. Even though we are not attaching probabilities to the outcomes of an action, we would not normally say that a state s' is a possible outcome of executing a in s unless there is a nonzero probability that a will take us to s'. For example, in Figure 3, the wait action has two possible outcomes s_1 and s_2, so it is fair to assume that both of these outcomes have nonzero probability of occurring. Consequently, if C is a cycle in Σ_π (e.g., the wait action's outcome s_2) and if one or more of the actions in C has another possible outcome (e.g., the wait action's outcome s_1), then the probability of remaining in C forever is 0, so any execution trace that remains in C forever is unfair.

3.4 Partitioning States into Equivalence Classes

To illustrate how combinatorial explosion can occur in multi-agent planning, we now consider a multi-agent planning domain called the Robot Navigation domain [3, 4, 2]. In this problem domain, a robot is supposed to move around a building such as the one shown in Figure 5, picking up packages and delivering them to their destinations. There is another agent in the building, a "kid," who is running around and opening and closing doors. The kid can move much faster than the robot, hence the kid may open or close each of the n doors in between each of the robot's actions.[1]

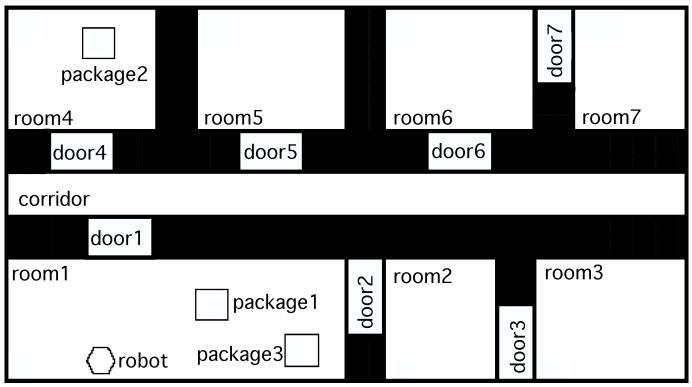

Fig. 5. A state in a Robot Navigation problem.

[1] Equivalently, one could assume that there are n kids, each of whom is playing with a different door.

If the building contains n doors and we model the kid's actions as nondeterministic outcomes of the robot's actions, then each of the robot's actions has 2^n possible outcomes: one for each possible combination of open and closed doors. If we represent each of an action's 2^n outcomes explicitly, then for every policy π and every state-action pair $(s, a) \in \pi$, the execution structure Σ_π will have 2^n successor states. In general, a solution policy will have exponential size and will take doubly exponential time to generate. This is not very good!

In the Robot Navigation domain, the size of the search space can be reduced by constructing policies over sets of states rather than individual states. For example, if the robot is in room room1 and wants to go into the hallway, then it matters whether door door1 is open but it does not matter whether any of the other $n - 1$ doors is open or closed. To go through door1, we only need to plan for two sets of states: the set S of all states in the robot is in room1 and door1 is open, (in which case the robot should move through the door), and the set of states S' in which the robot is in room1 and door1 is closed (in which case the robot should try to open the door).

More generally, we will want to represent π not as a set of pairs

$$\pi = \{(s_1, a_1), \ldots, (s_n, a_n)\},$$

where $s_1, \ldots, s_n$ are distinct states, but instead as a set of pairs

$$\pi = \{(S_1, a_1), \ldots, (S_k, a_k)\},$$

where $\{S_1, \ldots, S_k\}$ is a partition of $\{s_1, \ldots, s_n\}$. We'll call this a *partition-based* representation of π.

To represent a set of states, we can use a boolean formula that is satisfied by every state in the set. For example, suppose open1 is the proposition that door1 is open, and in1 is the proposition that the robot is in room1. Then we can use the boolean expression open1 $\wedge$ in1 to represent the set of all states in which door1 is open and the robot is in room1.

The MBP planner [5, 2] uses a representation of the kind described above.[2] In the Robot Navigation Domain, this representation enables MBP to avoid the exponential explosion described above: MBP can solve Robot Navigation problems very quickly [5, 2].

3.5 When the States Are Not Equivalent

MBP's state-representation scheme works well only when the state space can be divided into a relatively small number of equivalence classes. One illustration of this limitation occurs in the Hunter-and-Prey domain [7]. In this planning domain, the world is an $n \times n$ grid (where $n \geq 2$) in which an agent α called the *hunter* that is trying to catch one or more agents $\beta_1, \ldots, \beta_k$ called *prey*.

[2] More specifically, the boolean formulas are represented as Binary Decision Diagrams (BDDs) [6].

The hunter has five possible actions: move-north, move-south, move-east, or move-west, and catch. Each of the first four actions has the effect of moving the hunter in the stated direction, and is applicable if the hunter can move that direction without going outside the grid. The catch action has the effect of catching a prey, and is applicable only when the hunter and the prey are in the same location. For example, Figure 6(a) shows a situation in which the hunter has three applicable actions: move-north, move-west, and move-south.

Each prey has also five actions: a stay-still action which keeps the prey in the same square it was already in, and move-north, move-south, move-east, and move-west actions. These actions are similar to the hunter's actions described above, but with an additional restriction: at most one prey occupy a square at any one time, so it is not possible for two or more prey to perform movements that put them into the same square.

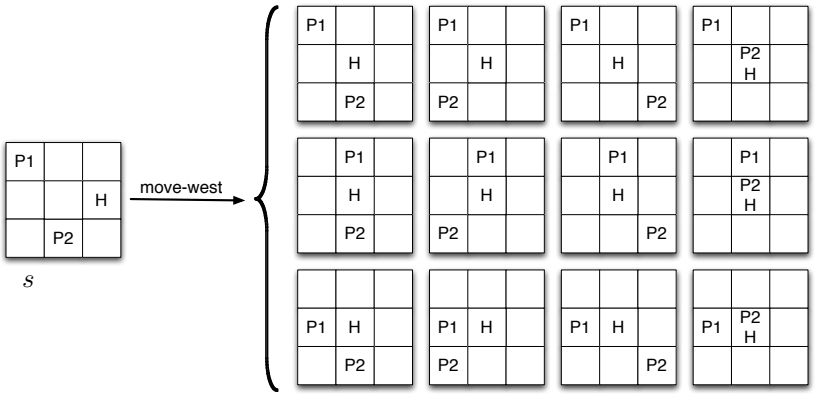

Fig. 6. A Hunter-and-Prey problem with two prey on a 3×3 grid. H represents the hunter's location, and P1 and P2 represent the locations of the two prey. In the state s shown at left, there are three possible moves for P1 and four for P2, hence twelve possible states that may be produced if the hunter moves west.

We can represent the possible actions of the prey as nondeterministic outcomes of the hunter's actions; Figure 6 gives an example. If there are m prey, any one of the hunter's actions may have up to 5^k outcomes; the exact number depends on the locations of the prey. A state's total number of predecessors or successors can be even larger.

On one hand, MBP can handle increases in grid size quite easily if there is just one prey (see Figure 7). This is because MBP can classify the locations of the hunter and prey into a small number of sets (e.g., in the set of all locations where the prey's x coordinate is 5 and the hunter's x coordinate is below 5, MBP might plan for the hunter to move East).

On the other hand, MBP's running time increases dramatically if we increase the number of prey (Figure 8). What causes MBP problems is the restriction that no two prey can be at the same place at the same time. This restriction means that unlike the doors' open/closed status in Robot Navigation problems, the prey's locations in Hunter-and-Prey problems are not independent of each other. When reasoning about the set of states in which prey p_i is in square (x, y), MBP cannot ignore the locations of the other prey because p_i's presence at (x, y) means that the other $m - 1$ prey must be in squares other than (x, y). MBP's running time grows because there are many different states in which this can happen and MBP cannot represent them as a small number of sets of states.

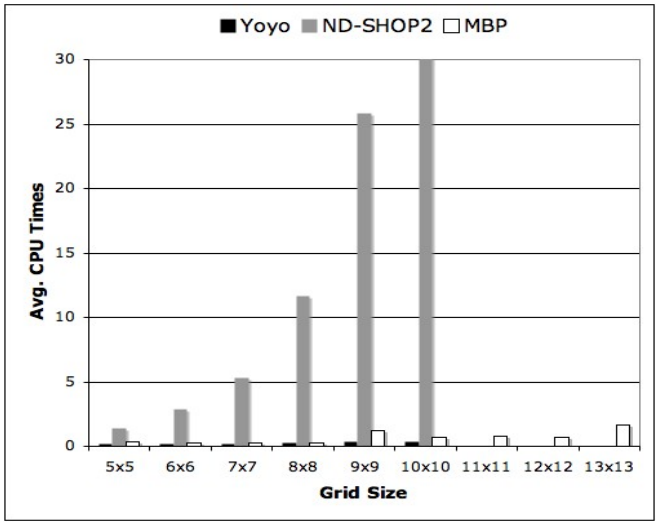

Fig. 7. Running time for MBP, ND-SHOP2, and Yoyo in Hunter-and-Prey problems with one prey and varying grid size.

3.6 Maintaining Focus on the Current Task

Another way of avoiding combinatorial explosion is to focus on one task t at a time, ignoring all actions except for those relevant for performing t. In the Hunter-and-Prey problem with a large number of prey, this means focusing on one prey at a time, and ignoring all of the other prey until this one has been caught.

In order to maintain focus on a particular task, we need a way to specify what the tasks are, and what actions are relevant to each task. One way to accomplish this is to use Hierarchical Task Network (HTN) planning [8–10]. In HTN planning, the objective of a planning problem is not expressed as a goal to be achieved, but instead as a task to be performed. Tasks are represented

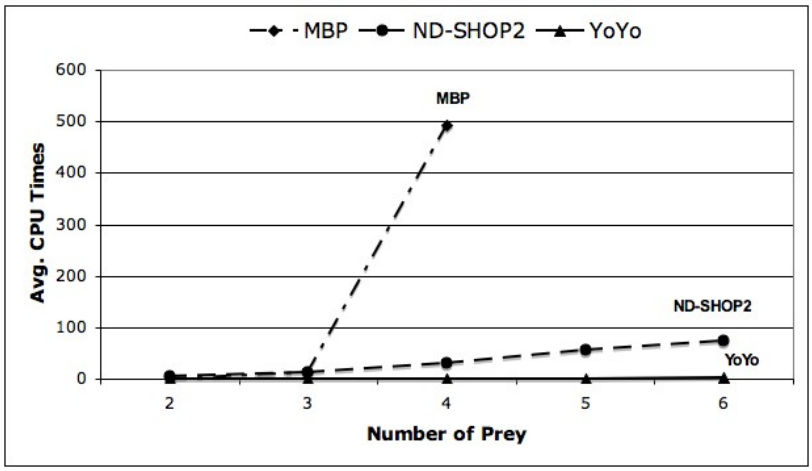

Fig. 8. Running time for MBP, ND-SHOP2, and Yoyo in Hunter-and-Prey problems with varying numbers of prey on a 4×4 grid.

as syntactic entities that look like logical atoms, but their semantics is different: they represent activities (e.g., actions or collections of actions) rather than conditions on states of the world.

In HTN planning, the description of a planning domain includes not only the planning operators for the domain but also a collection of *HTN methods*, which are prescriptions for how to carry out various tasks by performing collections of subtasks. Planning is done by applying methods to tasks to decompose them into smaller and smaller subtasks, until *primitive* tasks are reached that correspond directly to actions. If the actions are executable, the resulting plan is (by definition) a solution to the planning problem.[3]

ND-SHOP2 [13] is an HTN planning algorithm for nondeterministic domains. To solve Hunter-and-Prey problems with k prey, we can run ND-SHOP2 using methods that say basically the following:

- Method for the task catch-all-uncaught-prey
  if there are no uncaught prey, do nothing.
  else
    do subtask chase(β_i) for an arbitrarily selected uncaught prey β_i
    do subtask catch-all-uncaught-prey

[3] The status of HTN planning is somewhat controversial in the AI planning research community [11, 12]. AI planning theorists have a preference for "domain-independent" planning, in which the planner is given no specific knowledge about a domain other than the definitions of the planning operators for that domain. In contrast, HTN planning is quite popular among people who do practical applications of AI planning, because they want to be able to use the knowledge they have about the problems they are trying to solve, and HTN methods provide a way to encode such knowledge.

– Method for the task chase(β_i)
  if β_i is at the hunter's location, do action catch(β_i)
  else if β_i is to the north
    do action move-north, subtask chase(β_i)
  else if β_i is to the south
    do action move-south, subtask chase(β_i)
  else if β_i is to the east
    do action move-east, subtask chase(β_i)
  else if β_i is to the west
    do action move-west, subtask chase(β_i)

Note that the method for catch-all-uncaught-prey recursively invokes itself whenever any uncaught prey remain. This tells the hunter to keep chasing prey until all of them have been caught. Similarly, to tell the hunter to keep chasing β_i until β_i has been caught, the method for the task chase(β_i) recursively invokes chase(β_i) except in the case where the hunter catches β_i.

As shown in Figure 8, ND-SHOP2 runs much faster than MBP on problems where there are multiple prey and a small grid size. On the other hand, since ND-SHOP2 does not have MBP's ability to classify states into a small number of sets, it has difficulty dealing with large grid sizes, as shown in Figure 7.

3.7 Combining Focusing with Reasoning about Equivalent States

To plan for a large number of prey on a large grid, we would like to combine MBP's ability to reason about sets of states with ND-SHOP2's ability to focus on individual tasks. There is a planner called Yoyo that does this. The details are complicated and we will not describe them here, but the basic idea is as follows: Yoyo does ND-SHOP2's HTN decomposition, but instead of doing it over individual states, Yoyo does it over sets of states represented as BDDs. As shown in Figures 7 and 8, Yoyo outperforms both MBP and ND-SHOP2 on Hunter-and-Prey problems; and it also has been shown to outperform MBP and ND-SHOP2 in several other problem domains [14, 15].

3.8 Interleaving Planning and Acting

One of the biggest sources of difficulty in solving the Robot Navigation and Hunter-and-Prey problems is that we were trying to solve them *offline*, i.e., to generate the entire policy before executing it. The problems with exponential blowup in the size of the policy occurred because of the need to deal with all of the possible contingencies.

One of the reasons why AI planning research has traditionally focused on offline planning is that many planning problems contain *unsolvable* states, i.e, states from which it is impossible to reach any of the goal states. In such planning problems, it is important for the plan executor to avoid executing actions that

will take it to unsolvable states. One way to avoid such actions is to generate an entire solution plan before the plan executor starts executing.

Neither the Robot Navigation and Hunter-and-Prey domains contain unsolvable states, hence they can be solved via *online* planning, in which the plan executor executes each action as soon as the planner generates it. In these two domains, online planning is much easier to do than offline planning, because the planner only needs to plan for one of the possible outcomes of each action, namely the outcome that the plan executor encounters when executing the action.

As an example, here is a simple online-planning algorithm for solving Hunter-and-Prey problems:

while there are no uncaught prey **do**
  if there is a prey β_i in the same location as the hunter
    then execute $\mathsf{catch}(\beta_i)$
  else select a prey β_i arbitrarily, and move toward it

It is easy to prove that if the actions of the prey satisfy the fairness assumption discussed in Section 3.3, then the above algorithm is guaranteed to eventually catch all of the prey. In the Hunter-Prey domain, the fairness assumption means that if we keep coming back to the same state sufficiently many times, the prey will eventually do something different.

One could accomplish much the same thing by using a real-time search algorithm such as RTA* (real-time A*) [16, 7]. Furthermore, one could take almost any forward-search planner for deterministic planning problems (e.g., FastForward [17], TLPlan [18], or SHOP2 [19]), and modify it so that action selection is replaced with action execution: rather than appending an action to its plan and inferring the next state, the planner would immediately execute the action and observe the state directly.

The idea of modifying a classical planner to interleave planning and execution is somewhat similar to A-SHOP [20], but A-SHOP did not interleave planning and execution in the way that we are discussing here. Its objective was to generate a plan, not to execute it; and its interactions with other agents were purely for information-gathering. On the other hand, the idea has more in common with agent systems based on the BDI model [21], such as PRS [22], AgentSpeak [23], or RAP [24]. A recent system, CanPlan [25, 26], explicitly combines BDI reasoning with HTN planning.

4 Using Predictive Agent Models

One of limitation of the translation scheme in Section 3.1 is that the model of the other agents is trivial: it tells what actions the other agents *might* perform in different situations, but provides no way to help us predict how likely the agent will be to perform these actions. For good decision-making, it can sometimes be quite important to have such predictions.

As an example, consider the game of roshambo (rock-paper-scissors). The Nash equilibrium strategy for this game is to choose randomly among rock,

paper, and scissors, with a probability of 1/3 for each choice; and the expected utility of this strategy is 0 regardless of what strategy the opponent uses. But in a series of international competitions among computer agents that played roshambo, some of the programs did much better than the equilibrium strategy [27–29]. They did so by building predictive models of the opponent's likely moves, and using these models to aid in choosing their own moves.

Another example is the game of kriegspiel [30, 31]. This game is an imperfect-information version of chess, and its strategies are much more complicated than the strategies for roshambo—in fact, the game was used during the 19th century by several European countries as a training exercise for their military officers. It has been shown experimentally [32] that better play can be obtained by an opponent model that assumes the opponent will make moves at random, instead of using the minimax opponent model that is conventionally used in chess programs.

If β is an agent, we will define a *predictive model* of β to be a function $\hat{\beta}$ such that for each state s, $\hat{\beta}(s)$ is a probability distribution over the set of actions that β can perform in s. $\hat{\beta}$ need not necessarily be an accurate predictor of β's moves (although an accurate model is obviously preferable to an inaccurate one).

If we are playing a game G with β and we have a predictive model $\hat{\beta}$, then we can use $\hat{\beta}$ to translate the game into a Markov Decision Process (MDP). Sections 4.1 and 4.2 give quick summaries of what an MDP planning problem is and how the translation process works, Section 4.3 discusses how to partition states into equivalence classes, and Section 4.4 gives a case study on a game called the Noisy Iterated Prisoner's Dilemma (Noisy IPD).

4.1 MDP Planning Problems

A *Markov Decision Process (MDP)* planning problem is like a nondeterministic planning problem, but with the following changes:

- For state $s \in S_0$, there is a probability $P(s)$ that the initial state is s.
- If the set of possible outcomes for action a in state s is $\gamma(s, a) = \{s_1, \ldots, s_j\}$, then each of them has a probability $P(s, a, s_i)$, with $\sum_{i=1}^{j} P(s, a, s_i) = 1$.
- For each action a there is a numeric cost $c(a) \in \mathcal{R}$.
- For each state s there is a a numeric reward $r(s) \in \mathcal{R}$.
- There is a numeric *discount factor* δ, with $0 < \delta \leq 1$.[4]
- In most formulations of MDPs there is no explicit "goal states," but the same effect can be accomplished by giving these states a high reward and making them terminal states (i.e., states with no applicable actions) [33].

Given a policy π and an execution trace $T = \langle s_0, s_1, \ldots \rangle$, we can compute T's probability by multiplying the probabilities of the actions' outcomes:

$$P(T|\pi) = P(s_0)P(s_0, \pi(s_0), s_1), P(s_1, \pi(s_1), s_2), \ldots \ .$$

[4] In the MDP literature, the the discount factor is usually represented as γ, but that conflicts with our use of γ to represent the state-transition function.

The *utility* of the execution trace is the cumulative discounted difference between the rewards and costs:

$$U(T) = \sum_i \delta^i r(s_i) - \sum_i \delta^i c(\pi(s_i)).$$

The objective is to find a policy π having the highest *expected utility*

$$E(\pi) = \sum_T P(T|\pi)U(T).$$

Section 3.1 discussed how to extend the classical planning representation to represent nondeterministic planning problems. A similar approach can be used to represent MDPs, by including in the action representation the action's cost and the state-transition probabilities.

4.2 Translating Games into MDPs

Suppose two agents α and β are playing a game G, and let $\hat{\beta}$ be a predictive model for β's actions. Then we can use this model to translate G into an MDP planning problem $M(G, \hat{\beta})$. The translation is similar to the one described in Section 3.1, with the following additions:

- Each state in the game is a state in the MDP.
- As before, we represent β's possible actions as nondeterministic outcomes of α's actions—but this time we use $\hat{\beta}$ to compute probabilities for each of the outcomes. For example, suppose that in Figure 2, $\hat{\beta}$ says there is a probability of 3/4 that β will move right and a probability of 1/4 that β will move left. Then we would assign $P(s_0, \mathsf{move\text{-}right}, s_1) = 3/4$ and $P(s_0, \mathsf{move\text{-}right}, s_2) = 1/4$.
- We can obtain the actions' costs and the states' rewards directly from the definition of the game. For example, in chess the cost of each action would be 0, the reward associated with each nonterminal state would be 0, and the reward associated with each terminal state would be 1, −1, or 0, depending on whether the state is a win, loss, or draw.

4.3 Partitioning States into Equivalence Classes

Section 3.4 discussed how to decrease the size of the search space in a nondeterministic planning problem, by partitioning the set of states $\{s_1, \ldots, s_n\}$ into a set of equivalence classes $\{S_1, \ldots, S_k\}$ such that for each equivalence class S_i, the plan-execution agent will do the same action a_i at every state in S_i. Something similar can sometimes be done in MDPs, if an additional requirement can be met: every state in S_i must have the same expected utility.

As an example, consider the Iterated Prisoner's Dilemma (IPD). This is a well-known non-zero-sum game in which two players play n iterations (for some n) of the Prisoner's Dilemma, a non-zero-sum game having the payoff matrix shown in Table 1.

Table 1. Payoff matrix for the Prisoner's Dilemma. Each matrix entry (u_1, u_2) gives the payoffs for agents α and β, respectively.

		β's move:	
		Cooperate	Defect
α's move:	Cooperate	(3,3)	(0,5)
	Defect	(5,0)	(1,1)

In the Prisoner's Dilemma, the dominant strategy for each agent is to defect; and in the Iterated Prisoner's Dilemma, the Nash equilibrium is for both agents to defect in every iteration. But the iterations give each agent the opportunity to "punish" the other agent for previous defections, thus providing an incentive for cooperation [34, 35]. Consequently, there are empirical results (e.g., [35]) showing that several non-equilibrium strategies do better in general then the Nash equilibrium strategy. The best-known of these is Tit For Tat (TFT), a strategy that works as follows:

- On the first iteration, cooperate.
- On the i'th iteration (for $i > 1$), make the move that the other agent made on the $i - 1$'th iteration.

Suppose our predictive model $\hat{\beta}$ is the following approximation of TFT:

- On the first iteration, cooperate with probability 0.9.
- On the i'th iteration (for $i > 1$), with probability 0.9 make the same move that α made on the $i - 1$'th iteration.

In the IPD, each history (i.e., each sequence of interactions among the two players) is a different state, hence after i iterations we may be within any of 4^i different states. But since $\hat{\beta}(s)$ depends solely on what happened at the previous iteration, we can partition the states at iteration i into four equivalence classes such that $\hat{\beta}$ is invariant over each equivalence class:

- $S_{i,C,C} = \{$all states in which the pair of actions at iteration i was $(C, C)\}$;
- $S_{i,C,D} = \{$all states in which the pair of actions at iteration i was $(C, D)\}$;
- $S_{i,D,C} = \{$all states in which the pair of actions at iteration i was $(D, C)\}$;
- $S_{i,D,D} = \{$all states in which the pair of actions at iteration i was $(D, D)\}$.

This gives us the MDP shown in Figure 9, in which each state (i, a_1, a_2) corresponds to the equivalence class S_{i,a_1,a_2}.

The two main problems are (1) how to obtain an appropriate predictive model, and (2) how to use the MDP to plan our moves. As a case study, we now discuss these problems in the context of a program called DBS [36, 37] that plays a game called the *Noisy IPD*.

4.4 The Noisy Iterated Prisoner's Dilemma

The Noisy IPD is a variant of the IPD in which there is a small probability, called the *noise level*, that accidents will occur. In other words, the noise level is

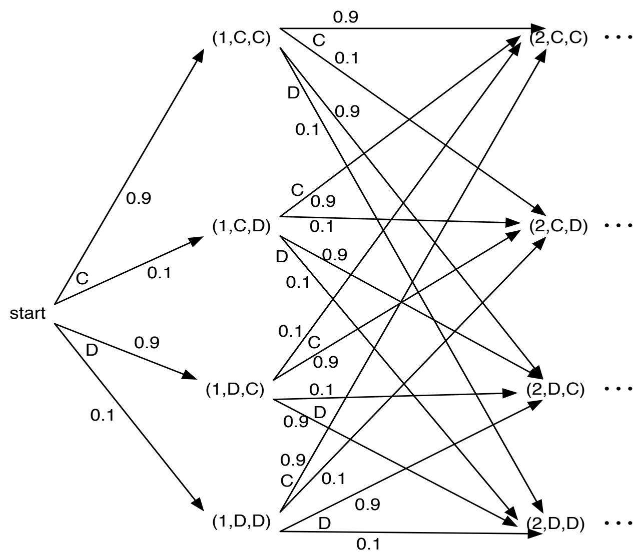

Fig. 9. An MDP in which each state is a set of equivalent game states.

the probability of executing “cooperate” when “defect” was the intended move, or vice versa.

Accidents can cause difficulty in cooperating with others in real life, and the same is true in the Noisy IPD. Strategies that do quite well in the ordinary (non-noisy) IPD may do quite badly in the Noisy IPD [38–43]. For example, if both α and β use TFT, then one accidental defection may cause a long series of defections by both agents as each of them retaliates for the other’s defections.

One way to deal with noise is to be more forgiving in the face of apparent misbehavior. For example, a strategy called Tit For Two Tats, which defects only when the other agent has defected twice in a row, can usually avoid the mutual-retaliation problem described above. One problem with such a strategy is that other agents can take advantage of it by occasionally defecting on purpose, without being punished for doing so.

Another way to deal with noise is to use a predictive model of other agent’s behavior to filter out the noise, as discussed later in this section.

Modeling the other agent’s behavior. As the game proceeds, DBS uses its observations of β’s behavior to build a predictive model $\hat{\beta}$ that will give probabilistic predictions of β’s future behavior. DBS’s predictive model is a set of rules of the form

$$\hat{\beta} = \{C_1(s) \to P_1, \;\; C_2(s) \to P_2, \;\; \ldots, \;\; C_m(s) \to P_m\},$$

where $C_1(s), \ldots, C_m(s)$ are mutually exclusive conditions (i.e., at most one of them is true in s), and P_i is the predicted probability that β will cooperate in a state that satisfies $C_i(s)$.

In principle, $\{C_1(s), \ldots, C_n(s)\}$ may be any set of mutually exclusive conditions, but which conditions should we use? If the conditions are too simple, then they will be incapable of accurately representing β's behavior, but if a condition $C_i(s)$ is too complicated, then it may be infeasible to learn an accurate value for P_i. DBS uses the following set of four very simple conditions:

- $C_1(s)$ is true if both agents cooperated on the previous iteration;
- $C_2(s)$ is true if α cooperated and β defected on the previous iteration;
- $C_3(s)$ is true if α defected and β cooperated on the previous iteration;
- $C_4(s)$ is true if both agents defected on the previous iteration.

One way to compute P_i is as follows (this is not exactly how DBS does it, but is an approximation). Let $0 < t < 1$ be a constant called the *threshold*, and $k > 0$ be an integer constant called the *window size*. Let S_i be the set of all states in the last k iterations that satisfy $C_i(s)$, Q_i be the set of all states in S_i in which β cooperated, and $r_i = |Q_i|/|S_i|$. Then we can set

$$P_i = \begin{cases} 0, & \text{if} \quad 0 \leq r_i \leq t, \\ r_i, & \text{if} \quad t < r_i < 1 - t, \\ 1, & \text{if} \quad 1 - t \leq r_i \leq 1. \end{cases}$$

Here are some of the reasons for computing P_i in the manner specified above:

- The conditions C_1, C_2, C_3, C_4 are inadequate to represent most IPD strategies over the entire course of a game, but they can often do well at representing the *recent* behavior of an IPD strategy. Hence we only compute P_i over the last k iterations rather than the entire history of the game.
- Clarity of behavior is an important ingredient of long-term cooperation, hence most successful IPD agents exhibit behavior that is at least partly deterministic, and we would like to model this. In the ordinary IPD, if β always cooperates when C_i is satisfied, then the ratio $r_i = 1$ will model this deterministic behavior. But consider the Noisy IPD with a noise level of, say, 10%. If β always cooperates when C_i is satisfied, then noise will transform 10% of these into defections. Hence $r_i = 0.9$ on average, which fails to model β's deterministic behavior. Hence in cases where r_i is close to 0 or close to 1, we'll want to hypothesize that β is actually behaving deterministically. The threshold t accomplishes this.

DBS computes its P_i values in a manner that is similar but not identical to the one described above. The main difference is that instead of using the ratio r_i, DBS uses a weighted ratio in which recent iterations are weighted more heavily than less recent iterations. For details, see [36].

Filtering noise. In cases where $\hat{\beta}$ predicts deterministic behavior (i.e., it predicts the probability of cooperation to be either 0 or 1), DBS can use this deterministic prediction to detect anomalies that may be due either to noise or a genuine change in the other agent's behavior. If a move is different from a deterministic prediction, this inconsistency triggers an *evidence collection process* that will monitor the persistence of the inconsistency in the next few iterations of the game. The purpose of the evidence-collection process is to try to decide whether the violation is due to noise or to a change in the other player's policy.

Until the evidence-collection process finishes, DBS assumes that the other player's behavior is the behavior predicted by $\hat{\beta}$, rather than the behavior that was actually observed. Once the evidence collection process has finished, DBS decides whether to believe that the other player's behavior has changed, and updates $\hat{\beta}$ accordingly.

Planning DBS's moves. Since the MDP in Figure 9 is infinite, DBS cannot generate the entire MDP. Instead, DBS plans its moves by generating and solving a truncated version of the MDP that ends at an arbitrary cutoff depth d (DBS uses $d = 60$). It is easy to compute an optimal policy π for the truncated MDP, using dynamic programming.

In an ordinary offline-planning problem, once the planner had found π, the plan executor would simply run π to completion. But this approach would not work well for DBS, because the predictive model $\hat{\beta}$ is only an approximation. By generating π, DBS may be able to make a good move for DBS in the current state, but we cannot be sure whether π will specify good moves in all of the subsequent states. Hence, instead of running π to completion, DBS executes only the first action of π, and recomputes π at every turn. Since the size of the MDP is polynomial in d, this does not require very much computation.

Performance. The 20th Anniversary Iterated Prisoner's Dilemma Competition [44] was actually a set of four competitions, each for a different version of the IPD. One of the categories was the Noisy IPD, which consisted of five runs of 200 iterations each, with a noise level of 0.1. 165 agents participated. Nine of them were different versions of DBS. As shown in Table 2, seven of these were among the top ten. Only two programs that did better: BWIN and IMM01.

BWIN and IMM01 both used the *master-and-slaves* strategy, which worked as follows: Each participant in the competition was allowed to submit up to 20 agents, and some of the participants submitted a group of 20 agents that could recognize each other by exchanging a pre-arranged sequence of Cooperate and Defect moves. Once the agents recognized each other, they worked together as a team in which 19 "slave" agents fed points to a single "master" program: every time a slave played with its master, the master would defect and the slave would cooperate, so that the master gained 5 points and the slave got nothing. Every time a slave played with a program not on its team, the slave would defect, to minimize the number of points gained by that program. BWIN and IMM01 were the "master" agents in two different master-and-slave teams.

Table 2. Scores of the top 10 programs, averaged over the five runs.

Rank	Program	Avg. score
1	BWIN	433.8
2	IMM01	414.1
3	DBSz	408.0
4	DBSy	408.0
5	DBSpl	407.5
6	DBSx	406.6
7	DBSf	402.0
8	DBStft	401.8
9	DBSd	400.9
10	lowESTFT_classic	397.2

DBS, in contrast, did not use a master-slave strategy, nor did it conspire with other agents in any other way. Despite this, DBS remained competitive with the masters in the master-and-slaves teams, and performed much better than the average score of a master and all of its slaves. A more extensive analysis [45] shows that if the size of each master-and-slaves team had been limited to less than 10, DBSz would have placed first.

5 Discussion and Conclusions

In general, planning gets very complicated when there are other autonomous agents to deal with. In order to accomplish this, it is essential to have a way to reduce the size of the search space. A variety of techniques have been developed in the AI planning literature for reducing search-space size in single-agent planning problems, and this paper has discussed how to utilize these techniques in multi-agent planning problems by translating the multi-agent planning problems into equivalent single-agent planning problems. In particular, we discussed two cases: one in which we wanted to achieve a given set of goals regardless of what the other agents might do, and one in which we wanted to maximize a utility function.

When the objective was to achieve a given set of goals regardless of the other agents' actions, the only model we needed of the other agents was what actions they were capable of performing in each state of the world. In this case, the approach was to model the other agents' actions as nondeterministic outcomes of the plan-execution agent's actions, and solve the problem offline in order to produce a policy for the plan-execution agent to use.

When the objective was to maximize a utility function, it mattered a great deal how likely or unlikely the other agent's actions might be. Hence, the approach in this case was to translate the multi-agent problem into an MDP in which the other agents' actions were represented as probabilistic outcomes of the plan-execution agent's actions. The probabilities of the outcomes were taken from a predictive model that was built by observing the other agent's behavior. This predictive model required updating as the game progressed; hence it

was necessary to do the planning *online*: at each move of the game, the planner constructed a new MDP based on the updated model, and solved this MDP to decide what move the plan-execution agent should make next.

In both cases, an important technique for making the problem feasible to solve was to *partition states into equivalence classes.* In the nondeterministic planning problems in Section 3.4, the equivalence classes were based on what action we wanted to do in each state. In the MDP planning problems in Section 4.2, the equivalence classes were based not only on the action to be performed but also on the state's expected utility.

In Section 3.6 we used HTN methods to achieve additional reduction in the size of the search space by pruning unpromising paths. This approach was not used in Section 4.2, because the simple structure of a repeated game like the IPD does not lend itself to this approach. However, HTN methods have been used successfully to prune parts of the search space in more complicated games, such as bridge [46].

Although the state-aggregation and HTN pruning techniques were quite successful in the cases discussed in this paper, they each have limitations that may cause difficulty in more complex problem domains. Here are two examples:

- Section 3.5 showed that there are relatively simple classes of problems in which state aggregation does not work well. In Section 3.8 we pointed out that the problem becomes much easier to solve if the planning is done online rather than offline—and we believe one promising avenue for further work is to develop techniques for deciding when to do the planning offline and when to do it online.
- Our opponent-modeling technique for the Noisy IPD is a relatively simple one, and more complex games require more sophisticated opponent models. We believe that the development of techniques for generating good opponent models will be a very important task.

Acknowledgments. This work has been supported in part by AFOSR grants FA95500510298, FA95500610405, and FA95500610295, DARPA's Transfer Learning and Integrated Learning programs, and NSF grant IIS0412812. The opinions in this paper are those of the authors and do not necessarily reect the opinions of the funders.

References

1. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann (May 2004)
2. Cimatti, A., Pistore, M., Roveri, M., Traverso, P.: Weak, strong, and strong cyclic planning via symbolic model checking. Artificial Intelligence **147**(1-2) (2003) 35–84
3. Pistore, M., Bettin, R., Traverso, P.: Symbolic techniques for planning with extended goals in non-deterministic domains. In: Proceedings of the European Conference on Planning (ECP). (2001)

4. Pistore, M., Traverso, P.: Planning as model checking for extended goals in non-deterministic domains. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), Seattle, USA, Morgan Kaufmann (August 2001) 479–484
5. Bertoli, P., Cimatti, A., Pistore, M., Roveri, M., Traverso, P.: MBP: a model based planner. In: Proceeding of ICAI-2001 workshop on Planning under Uncertainty and Incomplete Information, Seattle, USA (August 2001) 93–97
6. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys **24**(3) (1992) 293–318
7. Koenig, S., Simmons, R.G.: Real-time search in non-deterministic domains. In: IJCAI-1995. (1995)
8. Tate, A.: Project planning using a hierarchic non-linear planner. Technical Report 25, Department of Artificial Intelligence, University of Edinburgh (1976)
9. Sacerdoti, E.: A Structure for Plans and Behavior. American Elsevier (1977)
10. Erol, K., Hendler, J., Nau, D.S.: Complexity results for hierarchical task-network planning. Annals of Mathematics and Artificial Intelligence **18** (1996) 69–93
11. Kambhampati, S.: Are we comparing Dana and Fahiem or SHOP and TLPlan? a critique of the knowledge-based planning track at ICP. http://rakaposhi.eas.asu.edu/kbplan.pdf (2003)
12. Nau, D.: Current trends in automated planning. AI Magazine **28**(4) (2007) 43–58
13. Kuter, U., Nau, D.: Forward-chaining planning in nondeterministic domains. In: Proceedings of the National Conference on Artificial Intelligence (AAAI). (July 2004) 513–518
14. Kuter, U., Nau, D., Pistore, M., Traverso, P.: A hierarchical task-network planner based on symbolic model checking. In: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS). (June 2005) 300–309
15. Kuter, U., Nau, D., Pistore, M., Traverso, P.: Task decomposition on abstract states, for planning under nondeterminism. Artificial Intelligence (2008) To appear.
16. Korf, R.: Real-time heuristic search. Artificial Intelligence **42**(2–3) (1990) 189–211
17. Hoffmann, J., Nebel, B.: The FF planning system: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research **14** (2001) 253–302
18. Bacchus, F., Kabanza, F.: Using temporal logics to express search control knowledge for planning. Artificial Intelligence **116**(1-2) (2000) 123–191
19. Nau, D.S., Muñoz-Avila, H., Cao, Y., Lotem, A., Mitchell, S.: Total-order planning with partially ordered subtasks. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), Seattle (August 2001)
20. Dix, J., Muñoz-Avila, H., Nau, D.S., Zhang, L.: IMPACTing SHOP: Planning in a multi-agent environment. In Sadri, F., Satoh, K., eds.: Proc. Second Workshop on Computational Logic and Multi-Agent Systems (CLIMA), London, Imperial College (July 2000) 30–42
21. Bratman, M.E.: Intentions, Plans, and Practical Reason. Harvard University Press (1987)
22. Georgeff, M.P., Lansky, A.L.: Reactive reasoning and planning. In: Proceedings of the National Conference on Artificial Intelligence (AAAI). (1987) 677–682 Reprinted in [47], pages 729–734.
23. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In van Hoe, R., ed.: Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands (1996)
24. Firby, R.J.: Adaptive execution in complex dynamic worlds. PhD thesis 672, Yale University (1989)

25. Sardiña, S., de Silva, L., Padgham, L.: Hierarchical planning in BDI agent programming languages: A formal approach. In: aamas, Hakodate, Japan (May 2006) 1001–1008
26. Sardiña, S., Padgham, L.: Goals in the context of BDI plan failure and planning. In: aamas, Honolulu, HI (May 2007) 16–24
27. Billings, D.: The first international RoShamBo programming competition. ICGA Journal **23**(1) (2000) 42–50
28. Billings, D.: Thoughts on RoShamBo. ICGA Journal **23**(1) (2000) 3–8
29. Billings, D.: The second international roshambo programming competition. http://www.cs.ualberta.ca/ darse/rsbpc.html (2001)
30. Li, D.: Kriegspiel: Chess Under Uncertainty. Premier (1994)
31. Li, D.: Chess Detective: Kriegspiel Strategies, Endgames and Problems. Premier (1995)
32. Parker, A., Nau, D., Subrahmanian, V.: Overconfidence or paranoia? search in imperfect-information games. In: Proceedings of the National Conference on Artificial Intelligence (AAAI). (July 2006)
33. Boutilier, C., Dean, T.L., Hanks, S.: Decision-theoretic planning: Structural assumptions and computational leverage. Journal of Artificial Intelligence Research **11** (1999) 1–94
34. Aumann, R.: Acceptable points in general cooperative n-person games. In Luce, R.D., Tucker, A.W., eds.: Contributions to the Theory of Games. Volume 4. Princeton University Press (1959)
35. Axelrod, R.: The Evolution of Cooperation. Basic Books (1984)
36. Au, T.C., Nau, D.: Accident or intention: That is the question (in the iterated prisoner's dilemma). In: International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS). (2006)
37. Au, T.C., Nau, D.: Is it accidental or intentional? a symbolic approach to the noisy iterated prisoner's dilemma. In Kendall, G., Yao, X., Chong, S.Y., eds.: The Iterated Prisoners Dilemma: 20 Years On. World Scientific (2007) 231–262
38. Axelrod, R., Dion, D.: The further evolution of cooperation. Science **242**(4884) (1988) 1385–1390
39. Bendor, J.: In good times and bad: Reciprocity in an uncertain world. American Journal of Politicial Science **31**(3) (1987) 531–558
40. Bendor, J., Kramer, R.M., Stout, S.: When in doubt... cooperation in a noisy prisoner's dilemma. The Jour. of Conflict Resolution **35**(4) (1991) 691–719
41. Molander, P.: The optimal level of generosity in a selfish, uncertain environment. The Journal of Conflict Resolution **29**(4) (1985) 611–618
42. Mueller, U.: Optimal retaliation for optimal cooperation. The Journal of Conflict Resolution **31**(4) (1987) 692–724
43. Nowak, M., Sigmund, K.: The evolution of stochastic strategies in the prisoner's dilemma. Acta Applicandae Mathematicae **20** (1990) 247–265
44. Kendall, G., Yao, X., Chong, S.Y.: The Iterated Prisoner's Dilemma: 20 Years On. World Scientific (2007)
45. Au, T.C., Nau, D.: An analysis of derived belief strategy's performance in the 2005 iterated prisoner's dilemma competition. Technical Report CSTR-4756/UMIACS-TR-2005-59, University of Maryland, College Park (2005)
46. Smith, S.J.J., Nau, D.S., Throop, T.: A planning approach to declarer play in contract bridge. Computational Intelligence **12**(1) (1996) 106–130
47. Allen, J.F., Hendler, J., Tate, A., eds.: Readings in Planning. Morgan Kaufmann (1990)