To appear, *Proc. Fourteenth Inter. Conf. on AI (IJCAI-95)*, Montreal, August, 1995

Process-Oriented Planning and Average-Reward Optimality

Craig Boutilier[*]
Department of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z4, CANADA
cebly@cs.ubc.ca
http://www.cs.ubc.ca/spider/cebly/craig.html

Martin L. Puterman[†]
Faculty of Commerce
University of British Columbia
Vancouver, BC V6T 1Z2, CANADA
marty@markov.commerce.ubc.ca
http://acme.commerce.ubc.ca/puterman/puterman.html

Abstract

We argue that many AI planning problems should be viewed as *process-oriented*, where the aim is to produce a policy or behavior strategy with no termination condition in mind, as opposed to *goal-oriented*. The full power of Markov decision models, adopted recently for AI planning, becomes apparent with process-oriented problems. The question of appropriate optimality criteria becomes more critical in this case; we argue that *average-reward optimality* is most suitable. While construction of average-optimal policies involves a number of subtleties and computational difficulties, certain aspects of the problem can be solved using compact action representations such as Bayes nets. In particular, we provide an algorithm that identifies the structure of the Markov process underlying a planning problem – a crucial element of constructing average optimal policies – without explicit enumeration of the problem state space.

1 Introduction

The traditional AI planning paradigm requires an agent to derive a sequence of actions that leads from an initial state to a goal state. While much planning research has focussed on rather unrealistic models that assume complete knowledge of both states and actions, increasingly, research in planning has been directed towards problems in which the initial conditions and the effects of actions are not known with certainty, and in which multiple, potentially conflicting objectives must be traded against one another to determine optimal courses of action. In particular, there has been much interest in *decision-theoretic planning* (DTP) (Dean and Wellman 1991).

The theory of *Markov decision processes* (MDPs) has found considerable popularity recently both as a conceptual and computational model for DTP (Dean et al. 1993; Boutilier and Dearden 1994). Indeed, much recent research has emphasized the complementary nature of work in (for example) operations research (OR) on the foundations and computational aspects of MDPs, and planning models used in AI. Perhaps most important is the exploitation of structure in solving MDPs. Using compact representations of actions (such as influence diagrams or STRIPS operators) one can often group together large classes of calculations with great savings if the domain possesses many regularities (Tatman and Shachter 1990; Boutilier, Dearden and Goldszmidt 1995). We will exploit representations of this form below.

An important distinction that arises when one considers the use of MDPs for planning problems is that between *goal-oriented* planning problems and *process-oriented* problems. A goal-oriented problem is one in which an agent must construct a plan that will change the world from some initial state to one of a specified set of goals states. For example, constructing a plan to achieve a goal proposition G is a goal-oriented problem. Implicit in such problems is the assumption that the evolution of the system, once the goal is achieved, ceases to be of interest. The agent must be given another goal to achieve in order to begin planning and acting again. Such problems have received the bulk of the attention from the planning community, even when uncertainty is involved (though a relaxed definition of success may be used (Kushmerick, Hanks and Weld 1994)). In decision-theoretic settings, goal-based approaches are also common, with utilities used often to discriminate feasible plans (Dean et al. 1993).

A process-oriented problem is one in which there does not (necessarily) exist a goal state of the type described above. More specifically, there may be no state (or goal) such that the agent should stop acting once that state is reached (or the goal is true). Such problems require that the planning agent construct an on-going plan that proceeds indefinitely. While we focus on DTP, where these often occur naturally, process-oriented problems can also arise in more "classical" settings; for example, one might require an agent to construct a plan or *policy* that continuously alternates between states satisfying goals G_1 and G_2. If exogenous events can cause these goals to become false, then such a plan proceeds indefinitely.

MDPs are excellent models for such process-oriented problems: techniques such as *policy iteration* (Howard 1971) can be used to derive optimal plans for *infinite horizon problems* of this type under uncertainty. Unfortunately, the emphasis in recent work using MDPs for DTP has been on goal-oriented problems (Dean et al. 1993; Boutilier and Dearden 1994), albeit conditional and decision-theoretic. This is not to say that these algorithms don't work for process-oriented problems; but no consideration has been given to the issues and special circumstances that might arise when an ongoing process is involved. The full power of MDPs only comes to light

[*]This research was supported by NSERC Research Grant OGP0121843, and the NCE IRIS-II program Project IC-7.
[†]This research was supported by NSERC Grant OGP0005527.

when we have problems that exhibit this continual basis. One aim of this paper is to survey the unique challenges that arise when we attempt to solve process-oriented problems. Issues that take on added importance include representation of exogenous events, design of reward functions, and appropriate optimality criteria.

This last issue, the design of appropriate optimality criteria, has been paid little attention in DTP. MDPs have been used for planning and reinforcement learning quite extensively, and most models measure the goodness of policies using *discounted total reward* (one exception is (Singh 1994)). However, little thought seems to have been given to this choice of optimality measure or to good discounting rates. In fact, for many ongoing processes it seems that the correct (or most useful) measure of a policy is the *average reward* it accrues per unit time. Discounting admits conceptually simpler policy construction algorithms; but small discounting rates introduce unacceptable bias toward quick rewards at the expense of long-term gain, while large (close to one) discounting rates cause algorithms to converge quite slowly.

Unfortunately, finding average-optimal policies complicates most policy construction algorithms. Some algorithms such as *value iteration* (Bellman 1957; Puterman 1994) will work in almost the same form as for discounted problems, but only if one can establish the underlying "reachability" or *communicating* structure of the process. The second aim and key technical contribution of this paper is the development of an algorithm that determines this structure using a compact representation of the MDP's dynamics. Unlike existing algorithms for structure classification (Fox and Landi 1968), our algorithm exploits the problem representation to avoid enumeration and traversal of the underlying state space. This is an important feature because the planning state space grows exponentially with the number of variables or features present.

In Section 2 we will sketch a rather simplified, but in many respects realistic example to illustrate these considerations. We argue that many realistic problems ought to viewed as process-oriented rather than goal-oriented. We emphasize the importance of exogenous events (especially *user commands*) and considerations of appropriate reward structure. In Section 3, we describe the basic MDP model and policy construction techniques. In addition, we discuss compact representations of MDPs, the separation of events from actions, and point to ways in which these can be used to speed-up policy construction. In Section 4, we argue that average reward optimality is often appropriate for such problems and point out the difficulties involved in computing average-optimal policies. We also present the main technical contribution of the paper, namely, an algorithm that determines the underlying *communicating structure* of an MDP, a crucial step in the computation of average-optimal policies. By exploiting our action representation, (potentially exponential) reductions in time and memory requirements are possible for many problems, as compared to traditional state-space algorithms.

2 A Process-Oriented Planning Problem

Oft-used "gopher" domains are commonly viewed as goal-oriented planning problems. We have an agent (say a robot) that is designed to perform certain tasks for its owner (the user). Most planning algorithms suggest that the user will ask the robot to perform some task or achieve some goal.[1] The robot will construct a plan to achieve that goal, and then execute the plan. When that goal is achieved the robot waits, doing nothing, until another request is issued. This cycle of "Get goal; Achieve goal" is pervasive in classical and decision-theoretic models. However, this cycle of achieving goals in order is rather unrealistic for a number of reasons:

1. Many goals are not specifiable in this manner. Consider simple *maintenance goals* such as "keep the lab tidy." This is not a goal that can be achieved then abandoned. Though maintenance goals are used in classical planning, they typically specify constraints, such as subgoals and safety constraints, that the agent is not permitted to violate while achieving a primary goal. These serve a somewhat different purpose than true maintenance *goals*.
2. A user should not have to wait until a previous goal is satisfied before issuing another request; or if the robot stores requests in the order issued, it may not be desirable to have the robot delay achievement of later goals while completing earlier ones. A new goal may preempt previous goals — and there is no reason to expect some goals not to be preempted *indefinitely*.
3. We should not expect an agent's actions at any given time to be directed toward the achievement of a single goal proposition. Should multiple objectives be obtainable more readily, or at lower cost, by interleaving or sharing certain actions to achieve those objectives, an architecture that forces consideration of a single goal at any one time will produce suboptimal behavior.
4. An agent should plan not only for its current objectives, but also in *anticipation* of new goals or contingencies. An agent whose *raison d'être* is mail delivery may be well-served by positioning itself near the mailroom at certain times (if it has no other pressing tasks).

It should be clear that many of the problems to which classical goal-oriented planning techniques are currently applied may more naturally be thought of as process-oriented problems. While Point 1 indicates that some objectives are truly ongoing, Points 2–4 suggest that even multiple or recurring goals extended in time interact in ways that make the process-oriented perspective most suitable.

To make our discussion more concrete, we will focus on a particular example of a "gopher" robot with three primary responsibilities: to pickup and deliver mail to a user, to deliver coffee to the user, and to keep the user's lab tidy. This is not a goal-oriented problem in the classical sense. Keeping the lab tidy is certainly an on-going process. Mail arrives continually as does the user's need for coffee.[2] To formalize this problem we assume the six domain variables. *Loc*, the location of the robot, takes one of five values LO, LL, LM, LH, LC (office, lab, mailroom, hallway, and coffeeroom, connected in a cyclic fashion). T indicates lab tidiness with five values $T0$ (messiest) to $T4$ (tidiest). We also have four boolean variables denoting whether there is mail in the user's box (M), an outstanding coffee request by the user (*CR*), the robot has

[1] For example, *software agents*, as commonly conceived, often have this flavor.

[2] In (Boutilier and Puterman 1995) we give a full description of this problem, and further details of our algorithms.

mail (*HRM*), or the robot has coffee (*HRC*). This gives rise to a problem with 400 states.

Process-oriented problems typically arise in systems that change in certain ways independently of the agent's actions. Changes that demand the agent's continuing attention require that we model *exogenous events* that change the state of the system. An especially important class of such events will be *user commands*: so our agent can react to requests, we treat user commands as particular exogenous events that cause facts like "there is an outstanding request to do X" to become true. These are not goals in the classical sense, however, for an agent is under no obligation to drop what it is doing and immediately (or *ever*) satisfy the request. Requests must be balanced with other objectives in the derivation of an optimal course of action for the agent. The variable *CR* above serves this purpose — it indicates whether the user has issued a coffee request that remains unfulfilled. In order to model our problem, we assume three exogenous events occur occasionally: the arrival of mail (causing M), the user requesting coffee (causing *CR*), and the lab becoming untidy (causing T to decrease one unit). We assume the probability of any of these events occurring at a given time is known. Clearly, optimal plans vary with these probabilities. For instance, the robot may "hang out" at the mailroom if mail arrival is likely.

Our robot has a number of actions at its disposal: it can move through its domain in either direction (actions *Go*◌ and *Go*◌); it can pickup mail (*PuM*), successfully if in the mailroom and there is mail; it can deliver mail (*DelM*) in its possession to the user; it can pour coffee (*PC*) if in the coffee room; it can deliver this coffee (*DelC*) to the user in the office (causing a request *CR* to be fulfilled); it can tidy the lab (*Tidy*) by one unit; and it can do nothing (*Stay*).

To construct a plan, an agent must be able to predict the system state after execution of an action. Here however these predictions must account for the possible occurrence of exogenous events. A common technique for incorporating events is to "roll in" the probability of exogenous event occurrences and their effects into the action description. For example, when the robot considers the effect of *Go*◌, not only will it know that its location changes, it expects mail to arrive with some probability as well. However, the natural specification of the problem suggests that a user should be permitted to specify exogenous events and their effects independently of the action specification. So in addition to the eight actions, we assume that the three events described above (denoted *ArrM*, *ReqC*, and *Mess*) are specified independently in much the same format as actions. Unlike actions, whose occurrence is controlled by the agent, events must also come with a description of the conditions under and probabilities with which they may occur. For instance, we might assume that *ArrM* occurs with probability 0.2 at any "stage" (see below).

In order to construct a plan or policy, we can automatically "roll in" the event probabilities and effects into the action descriptions. This is usually a straightforward process; but problems arise when an action and an event affect the same variable in different ways. For instance, suppose the action *PuM* is executed at a certain stage in the plan (causing $\overline{M}$) and the event *ArrM* occurs at the same stage (causing M). There are no general principles by which the "true" effect of the action-event pair can be constructed from the information provided. Thus we assume that for any such conflicts, the user is willing to specify the "net effect" on the variable in question. In our domain, most action-event pairs have predictable effects on variables and the few contentious cases are resolved explicitly; for example if *ArrM* occurs concurrently with *PuM*, M is true (there is *more* mail to pick up). We describe action-event merging formally in the next section.

Also taking on added importance in process-oriented models is the representation of goals and objectives. If goals are classical (discrete propositions), how should one represent the fact that one goal should be achieved before another, or that a goal *has* been achieved and that the next can be pursued? In a decision-theoretic setting, how should one assign rewards or costs to fulfillment of objectives (or lack thereof)? In a process-oriented problem, the usual approach of assigning rewards to states in which objectives are satisfied becomes problematic — since the objective may remain true in subsequent states, there is a danger of "over-compensating" an agent for satisfying an objective once. On the other hand, associating rewards with state transitions (e.g., a transition to a good state from a bad one) has its own difficulties. We discuss these issues in detail in (Boutilier and Puterman 1995).

For this problem, and many in which there are separate objectives to be balanced, a useful reward model is one where *penalties* are associated with states in which objectives are unsatisfied. For instance, at any state where there is an outstanding user request *CR*, the agent is penalized. Such request variables become false when the objective (in this case, successful coffee delivery) is met. The magnitude of the penalty reflects the relative importance of the objective. In our example, we associate (additive) penalties with the following propositions: *CR* (an outstanding coffee request); $M \vee$ *HRM* (undelivered mail); and Tn if $n < 4$ (with penalties varying with degree of tidiness). The magnitudes of the penalties capture the relative priority of mail, coffee and tidiness. Optimal plans vary considerably with the relative importance of these objectives. For example, the robot may move to the mailroom if there are no current tasks and mail has high priority.

3 MDPs and their Representation

We model a DTP problem as a *completely observable MDP*. These are ideal for representing stochastic domains without classical goals, and especially process-oriented problems. We assume a finite set of states S, a set of actions $\mathcal{A}$ and a reward function R. An action takes an agent from one state to another, with each transition corresponding to a *stage* of the process. The effects of actions cannot be predicted with certainty; hence we write $Pr(s_1, a, s_2)$ to denote the probability that s_2 is reached given that action a is performed in state s_1. These transition probabilities can be encoded in an $|S| \times |S|$ matrix for each action.[3] Complete observability entails that the agent always knows what state it is in. We assume a bounded, real-valued *reward function* R, with $R(s)$ denoting the (immediate) utility of being in state s.[4] For our purposes an MDP consists of S, $\mathcal{A}$, R and the set of transition distributions $\{Pr(\cdot, a, \cdot) : a \in \mathcal{A}\}$.

A plan or *policy* is a mapping $\pi : S \rightarrow \mathcal{A}$, where $\pi(s)$ denotes the action an agent will perform whenever it is in state

[3] We assume any action can be attempted in any state.

[4] Costs can also be associated with actions in general.

s.[5] Policies naturally encode strategies suited for process-oriented problems; there is no notion of a finite sequence of actions or termination condition as in the classical setting. Given an MDP, an agent ought to adopt a policy that maximizes the expected *value* of its (potentially infinite) trajectory through the state space. Typically value depends in a compositional way on the states (in particular, the rewards $R(s)$) through which an agent passes. The most common value (and optimality) criterion in DTP for infinite-horizon problems is *discounted total reward*: the current value of future rewards is discounted by some factor β ($0 < \beta < 1$); and we want to maximize the expected accumulated discounted rewards over an infinite time period. The expected *value* (under this measure) of a fixed policy π at any given state s can be shown to satisfy (Howard 1971):

$$V_\pi(s) = R(s) + \beta \sum_{t \in S} Pr(s, \pi(s), t) \cdot V_\pi(t)$$

The value of π at any initial state s can be computed by solving this system of linear equations. A policy π is *optimal* if $V_\pi(s) \geq V_{\pi'}(s)$ for all $s \in S$ and policies π'.

Techniques for constructing optimal policies for discounted problems have been well-studied. While algorithms such as modified policy iteration (Puterman and Shin 1978) are often used in practice, an especially simple algorithm is *value iteration*, based on Bellman's (1957) "principle of optimality." We discuss value iteration because it can, under certain conditions, be used directly for average-reward problems as we describe below. Algorithms such as policy iteration may be much more complex in average-reward settings.

We start with a random value function V^0 that assigns some value to each $s \in S$. Given value estimate V^i, for each state s we define $V^{i+1}(s)$ as:

$$V^{i+1}(s) = \max_{a \in \mathcal{A}} \{R(s) + \beta \sum_{t \in S} Pr(s, a, t) \cdot V^i(t)\}$$

The sequence of functions V^i converges linearly to the optimum value in the limit. After some finite number n of iterations, the choice of maximizing action for each s forms an optimal policy π and V^n approximates its value.[6]

The above specification of MDPs requires that one spell out the transition matrices for each action and a reward function over the explicit state space S. Even for a relatively simple problem like the "gopher" example, with 400 states this can be prohibitive. Clearly, we do not expect users to specify problems in such an explicit form. Recently, a number of action representations such as STRIPS and influence diagrams have been applied to the problem of representing stochastic actions and MDPs generally (Kushmerick, Hanks and Weld 1994; Boutilier and Dearden 1994; Tatman and Shachter 1990). We adopt the "two-slice" *temporal Bayes network* (Dean and Kanazawa 1989). For each action, we have a Bayes net with one set of nodes representing the system state prior to the action (one node for each variable), another set representing the world after the action has been performed, and directed arcs representing causal influences between the these sets (see (Boutilier, Dearden and Goldszmidt 1995) for a more detailed discussion of this representation).

Figure 1 shows the specification of the *action network* for *PuM*, describing the effect of *PuM* independent of any event occurrences. The tables for the postaction variables describe the effects of the action. Nodes labeled *Persist* are unaffected and retain their preaction value (persistence tables are constructed automatically).

The *event network* for *ArrM* in Figure 1 has a somewhat different form. While the effects of events are specified as with actions (we omit persistence variables for conciseness), we must also indicate the probability of the event occurring. The *ArrM* network contains a double-circled node denoting the occurrence of the event in question, with an unconditional probability table. The parents of event nodes (though this example has none) are those variables that influence the probability of the event occurrence (e.g., *ArrM* could depend on the time of day).

Finally, the *net effect network* for *PuM* is shown: we notice that its effect on *Loc*, *HRC* and *HRM* is the same. Its effect on *CR* and T is altered, corresponding to the events *ReqC*, *Mess*; but the combination is derivable automatically. The contention between the effect of *PuM* and *ArrM* on the variable M has to be resolved by the user — in this case, we assume more mail arrives (i.e., the robot picks up mail at the beginning of the period). Implicit in this type of specification is the modeling assumption that the action and event networks simply describe what hold at the endpoints of a given stage. The action network for *PuM* says that if the robot is in the mailroom and there is mail at the beginning of a stage, the robot has the mail at the end of the stage. It makes no assumptions about how this effect is manifest *during* the intervening interval. Therefore, when combined with the event *ArrM* (interpreted similarly), we cannot predict the interactions of their effect on the contentious variable M: the user must resolve the conflict. We do, however, assume that explicit effects take precedence over "persistence" variables.

We note that these tasks should not be viewed as classical goals. Depending on the event probabilities and the importance of it objectives, under some circumstances tasks can be ignored. For example, if mail is far more important than tidiness and mail constantly arrives, the robot will never stop to tidy the lab under the optimal policy.

4 Average Reward Optimality

With goal-oriented problems, there is a straightforward measure of success. In many decision-theoretic problems, such as finite-horizon influence diagrams, one can sum the expected utility per stage of the policy. But for infinite-horizon process-oriented problems, the total accumulated reward typically diverges, making any direct comparison between policies meaningless. Thus discounting factors are often introduced. With a discounting rate less than one, total discounted reward will be bounded and comparisons can be carried out.

Unfortunately, the choice of discounting rate can have a drastic influence on optimal policies. A discounting rate such as 0.9 is hard to justify in our robot example, and can induce an unacceptable bias toward quick rewards. This essentially means that a unit reward achieved at stage $n + 1$ of the process is (currently) worth 90% of the value of a unit reward achieved at stage n — the motivation for discounting is pri-

[5] Such policies are *stationary*: action choice depends only on the state, and not the stage. For the problems we consider, optimal stationary policies always exist.

[6] We discuss *stopping criteria* in Section 4; see (Puterman 1994).

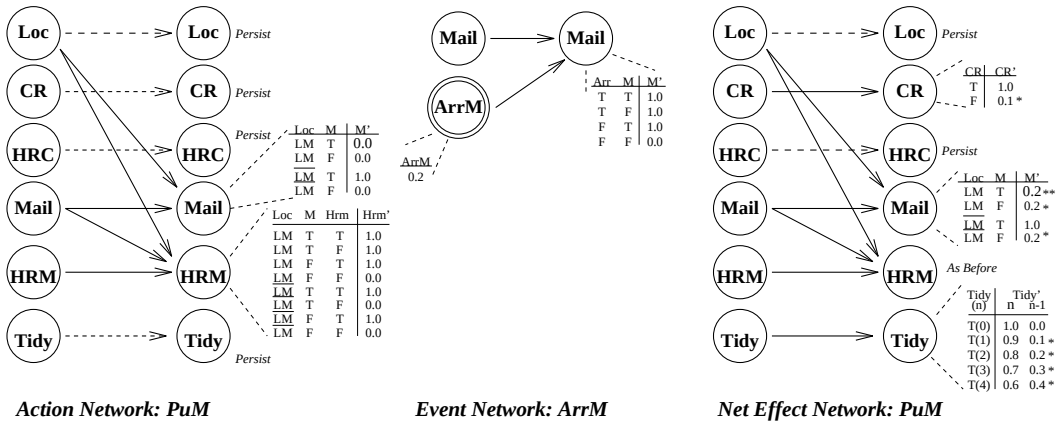

Figure 1: Action, Event and Net Effect Networks

marily economic. But it is difficult to provide an economic justification for discounting in problems such as these.

In process-oriented problems, we are primarily interested in the steady-state performance of our agent. As such, *expected average reward per stage* is the most appropriate measure of a policy. By choosing discount rates very close to one, optimal discounted policies may be similar to average-optimal policies; however, discounted algorithms may converge very slowly (e.g., value iteration) or involve ill-conditioned systems (e.g., policy iteration). Furthermore, directly computing average-optimal policies conforms closely to our intuitions about long-term processes. We present a brief summary of average-optimality and its computation, but refer to (Puterman 1994) for a detailed exposition.

The *expected average reward* or *gain* of a policy is:

$$g_\pi(s) = \lim_{n \to \infty} \frac{1}{n} V_\pi^n(s)$$

where $V_\pi^n(s)$ is the expected total reward when π is used for n stages starting at state s. Intuitively, the gain describes the steady-state average reward one can expect of a policy when starting in state s. A policy is *average (or gain) optimal* if it is not dominated by another policy in the usual sense, according to this measure.[7] In our finite state setting, average-optimal stationary policies always exist.

Computing average-optimal policies involves a number of subtleties that make approaches such as policy-iteration rather complex. However, one of the interesting aspects of this optimality measure, which can be exploited for computational gain, is its sensitivity to the *chain* or *communicating structure* of the MDP. We can classify an MDP according to the Markov chains induced by the stationary policies it admits. For a fixed Markov chain, we can group states into maximal *recurrent classes* such that each state reaches every other state in that class eventually; states belonging to no recurrent class are called *transient*. An MDP is *recurrent* if each policy induces a Markov chain with a single recurrent class. An MDP is *unichain* if each policy induces a single recurrent class with (possibly) some transient states. An MDP is *communicating* if for any pair of states s, t, there is *some* policy under which s can reach t. We call other policies *noncommunicating*.[8]

Unichain and recurrent MDPs are especially well-behaved: the gain of every stationary policy is constant (i.e., $g_\pi(s)$ is identical for all $s \in S$), and methods such as policy and value iteration can be used in a relatively straightforward way. But planning problems will seldom exhibit this structure. To be recurrent, we must know the agent will visit each state infinitely often *no matter what policy it adopts*. It will almost always be the case that an agent *can choose to* avoid certain states. As soon as we have a domain where an agent can move to a certain sections of the state space and remain there (e.g., *Stay*), the MDP will not be unichain or recurrent.

While not quite so well-behaved, communicating models have the nice feature that optimal policies (though not all policies) must have constant gain. While policy iteration becomes much more complicated in this case, value iteration can be used directly. To construct an optimal policy, we run value iteration as described above with $\beta = 1$, stopping when the *span*[9] of the difference between two consecutive estimates is small; in other words, value iteration stops when $Sp(V^{i+1} - V^i) \leq \varepsilon$ for some small ε. Thus when the difference between two value estimates is nearly constant, we are close to an average optimal policy. However, this algorithm can only be used under conditions when we know the optimal gain is constant; otherwise the algorithm may not converge.[10] Otherwise more complex methods are required. Thus, the identification of the underlying chain structure of an MDP becomes an important computational tool for constructing average optimal policies.

We note that the techniques of (Boutilier, Dearden and Goldszmidt 1995) can be applied in this setting, allowing value iteration to work on groups of states instead of com-

[7] We assume this limit exists. This may not be the case if the MDP admits policies that are *periodic*; in this case, the definition may use a slightly more robust *Cesaro limit* (Puterman 1994).

[8] In the full paper we discuss *weakly communicating* MDPs, which share nice features with communicating MDPs.

[9] The span of a function V on S is defined as $Sp(V) = \max_{s \in S} V(s) - \min_{s \in S} V(s)$.

[10] The algorithm may also not converge if the MDP admits periodic chains; but *aperiodicity transformations* that introduce a small amount of noise can be used. Note also that setting $\beta = 1$ is not problematic; *relative value iteration* can be used if undiscounted values get too large. See (Puterman 1994) for these details.

puting over an explicitly enumerated state space, if it can be factored (e.g., using a Bayes net). Thus, our representation can be exploited for computational gain as well.

4.1 Discovering Communicating Structure

We expect many DTP problems to be communicating. These problems are such that an agent *could* with positive probability reach any state from any other state. However, noncommunicating problems are not rare in planning domains (e.g., if there are "irreversible choices", such as a robot going down "unclimbable" stairs, or an agent breaking an egg). Thus, we must take care to classify an MDP before attempting to construct an average optimal policy. If the MDP is communicating, value iteration can be used directly. The classification algorithm we use has the added advantage that it can be used to apply value iteration (piecewise) to general MDPs (as we sketch below).

An efficient algorithm for classifying Markov chains known as the *Fox-Landi algorithm (FL)* (Fox and Landi 1968) can be extended to the classification of MDPs by considering the "reachability" matrix for the MDP. Roughly, we construct a single transition matrix that assigns positive probability to entry i, j if there is any action that moves the process from state i to state j with nonzero probability. FL works by constructing paths through the state space using this reachability matrix, producing a labeling and grouping of all states. Roughly, a start state i is chosen and a path is constructed by adding a state j reachable from i, a state reachable from j, and so on. If the path ever loops, the entries in the loop are merged into one "superstate." Note a path can always be extended, although it might form a cycle. If the cycle (or superstate) cannot be extended (i.e., all states reach only other states in the cycle), then the states in the cycle are grouped into the same recurrent class. All states on the path leading to (but not part of) the cycle are classified as transient. Then a new unclassified start state is chosen. If, in path extension, a previously classified state (either recurrent or transient) is ever reached, all states on the path are transient, and we begin again. If FL classifies all states as recurrent and *puts them in the same recurrent class*, then the MDP is communicating and value iteration can be used to solve it.

This form of the FL algorithm requires explicit enumeration of the state space, and fails to exploit regularities captured in our representation of the system dynamics. To avoid this, we present a *structured Fox-Landi* algorithm (SFL) that uses the action descriptions directly. SFL can be used to classify an MDP directly, or more generally classify any compactly represented Markov chain. Furthermore, in conjunction with a structured implementation of value iteration, it can be used to compute average-optimal policies for arbitrary MDPs (regardless of chain structure).

Schematic states and paths: The key feature of the SFL algorithm is its use of a schematic representation for states, paths and cycles, allowing entire groups of paths to be extended in a single operation. The schematic path building and cycle detection operation then itself involves a number of crucial components, which we briefly describe.

Schematic states (s-states) represent groups of states corresponding to a partial variable assignment. For example, we use $\langle LL \cdot \cdot \cdot \cdot \cdot \rangle$ to capture a state where *LL* (lab) is true, and the other variables (M, T, etc.) have some fixed value. In general, an s-state consists of n slots to represent values of n domain variables. A slot can be filled in various ways. It can have a fixed value such as *LL*, or an *arbitrary* fixed value from a certain set, denoted (LL, LO). This represents any fixed state with one of the specified values. We abbreviate *all* values of a variable using a dot as shown above; and we use an overline to denote the complement of the value set.

Schematic paths (s-paths) are constructed by applying actions to s-states — since actions have local effects, only certain portions of an s-state are affected. This can be viewed as implicitly extending every state consistent with the s-state. For example, in Figure 2(a) the s-state above is extended to the state $\langle LO \cdot \cdot \cdot \cdot \rangle$. This is reached by applying action *Go*◌ (whose effect can be read from its network). This s-path of length two actually represents the 80 true paths induced by assignment to the variables. An s-path with fixed values represents the set of paths where the variable has some fixed value everywhere in that path (unless a different value occurs later in the path). We can also represent cycles schematically as single states. The notation $\{T1, T0\}$ in Figure 2(c) means that any value in that set is "reachable" from any other value. Thus, it captures a cycle between states where $T0$ and $T1$ hold (all else equal). $\{L*\}$ abbreviates a cycle among all possible values of variable *Loc* (see Figure 2(a)).

A key element in path construction and cycle detection is *unification*, to test whether two s-states intersect (i.e., share states). Unification is straightforward – it identifies the states shared by two s-states (the unifier), as well as those they do not, in a symbolic fashion. It is used to join two s-paths or form a cycle; but in general, when two paths are joined at an s-state, the unification is not complete (i.e., there will be states that are not shared). In this case, the s-paths will *split*: a concatenated s-path will be formed using the unifier (common states), and the remaining states will be split off symbolically, leaving two more specific s-paths (we see this below). A detailed exposition of path splitting is not possible here.

Finally, because an s-path represents a group of paths, and can be split into more specific s-paths, we must keep track of partially constructed s-paths that have not been extended to completion. Unlike ordinary FL, which only ever builds one path, we must keep an *open list* of such partial s-paths. When extending the *current path*, we will try to unify the head with earlier states in the path (to create cycles) or an existing path on the open list. By creating cycles whenever possible, the problem representation tends to stay compact.

Structured Fox-Landi Algorithm: We give a high-level sketch of the SFL algorithm (Figure 3), and describe its application to our example (Figure 2). We defer a detailed description to (Boutilier and Puterman 1995) along with more formal definitions and a proof of correctness. The example here blurs a number of steps in the algorithm for conciseness.

We begin by choosing the initial s-state in Figure 2(a). It is called the *current path*, and the main loop of the algorithm constantly extends the current path by applying an action and choosing some possible outcome of that action. In this case the action *Go*◌ is applied several times, extending the path to length 5. The sixth application returns to the initial state in the path. This is detected by the unification procedure during cycle detection. Whenever the current path is extended, the new head state is compared to all (unclassified) visited s-states, either those earlier in the path or those on the *open list*.

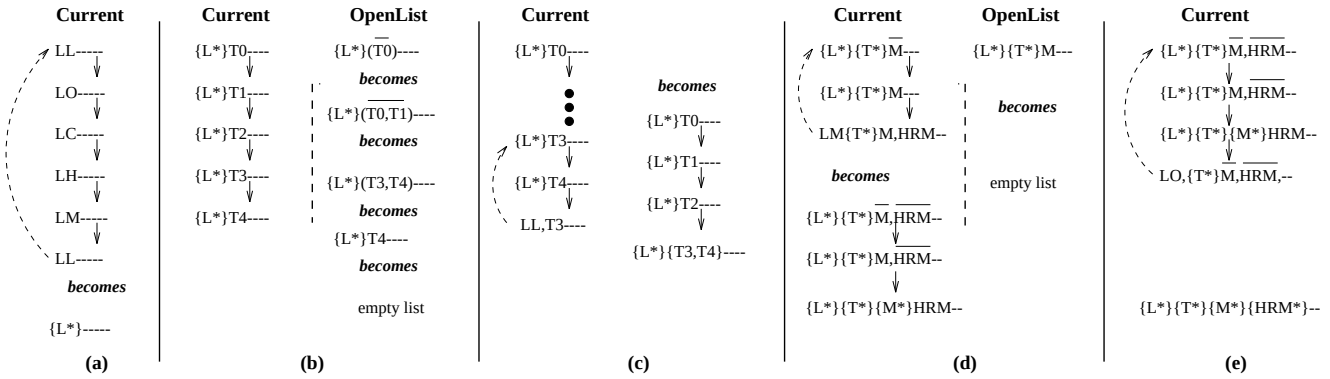

Figure 2: The Structured Fox-Landi Algorithm

1. **Initialization:** Set current path to be some s-state.
2. **Termination:** If all states labeled, exit.
3. **Path Extension:** Extend the current path:
   (a) Choose an action and apply to current state (specialize current state if needed, and split current path, adding residual to open list)
   (b) Add outcome to head of current path
4. **Recurrent Class Detection:** If no action extension is possible (i.e., cannot leave cycle): a) label cycle (head) as recurrent and rest of path as transient; b) choose new current path from open list (or choose an unvisited state).
5. **Classification:** If current state unifies with a labeled state, classify (possibly specialized) current path as appropriate (splitting path as in 3, if needed)
6. **Cycle Detection:** If current state unifies with a previous state on current path, form cycle at head of path (possibly splitting path as in 3)
7. **Path Joining:** If current state unifies with a state on open list, append (part of) path on open list to current path (possibly splitting both paths as in 3)

Figure 3: Sketch of Structured Fox-Landi

In this case a cycle is detected and the path is collapsed into the s-cycle at the bottom of Figure 2(a). Thus, the robot can (with nonzero probability) reach any location from any other without disturbing other variables.

We continue in Figure 2(b) by specializing this cycle (the head of the current path) with the value $T4$. The "rest" of the s-cycle is still valid: it is *split* from the current path and added to the open list for extension in the future. We apply the action *Stay* several times under which the lab (with nonzero probability) gets messier, giving us the current path in Figure 2(b). At each point, one fewer "instance" of T is left on the open list, since the head state at each path extension step unifies with a specialization of the s-state on the open list (detected in path joining). By the end, the open list is empty.

In Figure 2(c), the action *Tidy* is applied at the head of the current list. While *Tidy* only has the desired effect when *LL* holds, the condition $\{L*\}$ ensures that the necessary condition *LL* is reachable. But after the action, *LL* remain true. Cycle detection discovers that this new state unifies with a previous state on the path, and the new cycle is formed (the second path in Figure 2(c)). With several more applications of *Tidy*, we easily get to the state $\langle\{L*\}\{T*\}\cdots\rangle$.

It is worth noting, at this point, that we have discovered that the "subprocess" consisting of variables *Loc* and T is now known to be communicating, although we haven't explicitly constructed a path through all 25 states ($5 \cdot 5$) of this process. Instead, we have shown that all values of *Loc* communicate and that all values of T communicate under some value of *Loc*. This simple subprocess illustrates the spirit of SFL. We expect that problems that can be decomposed into groups of variables that have strong mutual influence (within groups), but relatively constrained influences between groups, will be very well-suited for SFL (see "Heuristics" below).

We continue in Figure 2(d) by considering the variable M. We start with value $\overline{M}$, and extend it with *Stay* (making M true due to possible mail arrival); this unifies with the initial open list, making it empty. In an effort to form a quick cycle, we apply action *PuM*. The condition *LM* is satisfied by $\{L*\}$, and holds following the action. Another effect however in *HRM*. This unifies with the initial state; but forces the current path to split: only *HRM* becomes part of the cycle (nothing in *PuM* can force the robot to lose the mail). The split chain $\overline{HRM}$ stays apart from the cycle. Finally, in Figure 2(e) we extend this chain with the *DelM* action: if *LO* holds then *HRM* becomes false and the path collapses into a cycle. The variables *CR* and *HRC* will behave similarly, and thus our MDP is communicating.

For other problems, the algorithm is somewhat more complex. Here we notice that each s-state can be extended to a novel s-state by some action until the obvious final step. If there are multiple recurrent classes, when we complete the construction of a maximal cycle, some effort is required to ensure that it is a maximal class. In particular, we must ensure that no action can move the system out of that class of states. However, given the schematic representation of cycles and paths and the structured action representations, this can usually be verified quite readily. Even in the worst case (with *no* exploitable structure), the effort is no more than that needed to construct the reachability matrix for FL.

Heuristics: We note that in our example the algorithm verifies the communicating structure in under 30 steps of path extension. Even with the overhead of unification, this is considerably better than the $O(|S|^2)$ steps (in this case, roughly 160,000) required by FL. Of course, we have exploited "good" action and outcome choices in performing the algorithm here. A crucial aspect of SFL is the use of heuristic information encoded in the action representation when choosing the "direction" in which to extend a path.

The main guiding principle is that we attempt to find the "local communicating structure" of individual or small groups of

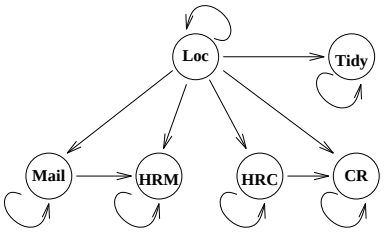

Figure 4: Influence Graph for Example Problem

variables that are, to some extent, shielded from the influence of other variables. In particular, we try to find short s-cycles in small groups of variables, choosing particular variables and outcomes that will unify with earlier states. We choose the variables to extend using an *influence graph* that describes influences between variables (see Figure 4). In our example, *Loc* is expanded first since *no other variables under any action* influence the probability of *Loc* (as indicated by the graph): the structure of *Loc* is independent of any other conditions. In our example, this means under all circumstances it can be ignored when determining the structure of other variables. All variables are partially ordered by the graph and are expanded roughly reflecting this order.[11]

4.2 Exploiting Communicating Structure

Our algorithm has three outcomes of interest: either a single recurrent class is discovered, a single class plus transient states, or more than one recurrent class (plus possibly transient states). If our aim is to simply categorize an MDP as communicating or not, the algorithm can be terminated as soon as any transient states (or multiple recurrent classes) are identified. If identified as communicating, a simple algorithm like value iteration, or related methods based on structured representations (Boutilier, Dearden and Goldszmidt 1995), can be used to determine the average optimal policy.

If the algorithm discovers more than one recurrent class then the MDP is *multichain* (i.e., general). If a single recurrent class is discovered together with transient states, then it may be *weakly communicating* or multichain. Weakly communicating MDPs also have constant gain and can be solved using value iteration; however, determining this fact requires examination of individual policies, something our algorithm does not currently do. If the process is multichain, more complex methods may have to be used.

However, Ross and Varadarajan (1991) have proposed a method for decomposing general MDPs. We are currently adapting this method for use with SFL to constructing average optimal policies using (piecewise) value iteration. Roughly, the recurrent classes identified by SFL can be "solved" independently using value iteration (since they must have constant gain). Then these states are "eliminated." Transient states are reclassified in this reduced MDP, and FL is run again on the remainder of the state space (ignoring these recurrent classes). The second level of FL provides new recurrent classes for which optimal gain (in the sub-problem) is constant. These can be pieced together with the previously classified states to determine a new policy: if the gain in the subproblem is greater, these states adopt actions that keep them from the earlier states. The procedure continues until all states are classified.

[11] The precise meaning of the graph and its construction are described in (Boutilier and Puterman 1995).

5 Concluding Remarks

We have argued that many planning problems are process-oriented and that special consideration must be given to these, especially in the choice of reward and action representation. We also claim that average-optimality is the most appropriate measure of performance for many process problems, and have presented the SFL algorithm to determine the communicating structure of an MDP, an important part of constructing average-optimal policies, using compact action representations. We are currently exploring further heuristics for the algorithm, conducting experiments to determine general problem characteristics that predict good performance of SFL as compared to standard FL, and extending our approach to multichain problems.

Future research includes applying these ideas to semi-Markov models, where actions can take varying amounts of time, and the use of more general modeling assumptions for events. The discovery of weakly-communicating MDPs using structured paths is also of interest.

References

Bellman, R. E. 1957. *Dynamic Programming*. Princeton U. Press.

Boutilier, C. and Dearden, R. 1994. Using abstractions for decision-theoretic planning with time constraints. *AAAI-94*, pp.1016–1022, Seattle.

Boutilier, C., Dearden, R., and Goldszmidt, M. 1995. Exploiting structure in policy construction. *IJCAI-95*. This volume.

Boutilier, C. and Puterman, M. L. 1995. Communicating Structure and Average Optimal Policies. Tech. report, Univ. British Columbia, Vancouver. (Forthcoming).

Dean, T., Kaelbling, L. P., Kirman, J., and Nicholson, A. 1993. Planning with deadlines in stochastic domains. *AAAI-93*, pp.574–579, Washington, D.C.

Dean, T. and Kanazawa, K. 1989. A model for reasoning about persistence and causation. *Comp. Intel.*, 5(3):142–150.

Dean, T. and Wellman, M. 1991. *Planning and Control*. Morgan Kaufmann, San Mateo.

Fox, B. L. and Landi, D. M. 1968. An algorithm for identifying the ergodic subchains and transient states of a stochastic matrix. *Comm. of the ACM*, 2:619–621.

Howard, R. A. 1971. *Dynamic Probabilistic Systems*. Wiley.

Kushmerick, N., Hanks, S., and Weld, D. 1994. An algorithm for probabilistic least-commitment planning. *AAAI-94*, pp.1073–1078, Seattle.

Puterman, M. L. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, New York.

Puterman, M. L. and Shin, M. 1978. Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, 24:1127–1137.

Ross, K. W. and Varadarajan, R. 1991. Multichain Markov decision processes with a sample-path constraint: A decomposition approach. *Math. of Op. Res.*, 16(1):195–207.

Singh, S. P. 1994. Reinforcement learning algorithms for average-payoff markovian decision processes. *AAAI-94*, pp.700–705.

Tatman, J. A. and Shachter, R. D. 1990. Dynamic programming and influence diagrams. *IEEE Trans. Sys., Man and Cyber.*, 20(2):365–379.