Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2007-061 December 31, 2007

Relational Envelope-based Planning

Natalia Hernandez Gardiol

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Relational Envelope-based Planning

by

Natalia Hernandez Gardiol

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2008

Author .....
Department of Electrical Engineering and Computer Science
December 31, 2007

Certified by .....
Leslie Pack Kaelbling
Professor
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Relational Envelope-based Planning

by

Natalia Hernandez Gardiol

Submitted to the Department of Electrical Engineering and Computer Science
on December 31, 2007, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis proposes a synthesis of logic and probability for solving stochastic sequential decision-making problems. We address two main questions: How can we take advantage of logical structure to speed up planning in a principled way? And, how can probability inform the production of a more robust, yet still compact, policy?

We can take as inspiration a mobile robot acting in the world: it is faced with a varied amount of sensory data and uncertainty in its action outcomes. Or, consider a logistics planning system: it must deliver a large number of objects to the right place at the right time. Many interesting sequential decision-making domains involve large state spaces, large stochastic action sets, and time pressure to act.

In this work, we show how structured representations of the environment's dynamics can constrain and speed up the planning process. We start with a problem domain described in a probabilistic logical description language. Our technique is based on, first, identifying the most parsimonious representation that permits solution of the described problem. Next, we take advantage of the structured problem description to dynamically partition the action space into a set of *equivalence classes* with respect to this minimal representation. The partitioned action space results in fewer distinct actions. This technique can yield significant gains in planning efficiency.

Next, we develop an *anytime* technique to elaborate on this initial plan. Our approach uses the *envelope* MDP framework, which creates a Markov decision process out of a subset of the possible state space. This strategy lets an agent begin acting quickly within a restricted part of the full state space, as informed by the original plan, and to judiciously expand its envelope as resources permit. Finally, we show how the representation space itself can be elaborated within the anytime framework.

This approach balances the need to respond to time-pressure and to produce the most robust policies possible. We present experimental results in some synthetic planning domains and in a simulated military logistics domain.

Thesis Supervisor: Leslie Pack Kaelbling
Title: Professor

Acknowledgments

The fellow is either mad or he is composing verses.
— Horace, Satires

I owe enormously to many for being able to embark on and finish this work. My advisor, Leslie Pack Kaelbling, has been a constant and inspiring source of clarity and guidance. Without her singular ability to zero in on the very grain of a question, I am sure I would have been completely lost. The warmth and collegiality of the Learning and Intelligent Systems group at MIT is testament to her hard work and generosity, and I deeply grateful to count myself as one of her students. I am also grateful to the members of my committee, Tomas Lozano-Perez at MIT, Hector Geffner at Universitat Pompeu Fabra in Barcelona, and Tom Dean of Brown University and Google, for their remarkable insight and fresh perspectives on my work.

While at MIT, I have been privileged to know and work with some exceptional people. Sarah Finney, Luke Zettlemoyer, Meg Aycinena, Emma Brunskill, James McLurkin, Nick Matsakis, Hanna Pasula, and Mike Ross have been fantastic colleagues — generous with their wisdom and willing to wrestle with research (and other!) quandaries not their own. I must also thank the MIT Outing Club and the Cycling Team for keeping me sane; I've learned so much from both groups.

I am also extremely grateful to Sridhar Mahadevan, now at the University of Massachusetts, for taking me under his wing while I was an undergraduate at Michigan State University. Were it not for that opportunity to work with him and his group — among them, Georgios Theocharous, Khashayar Rohanimanesh, Silviu Minut — I would certainly not have been able to take the path I did.

I would not be anywhere, of course, without my family. They have supported my endeavors from the very first. They have instilled in me by their own example a sense of curiosity and wonder about the natural world. They are a constant. I thank them for letting me depend on them.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

For an intelligent agent to operate efficiently in a highly complex domain, its only hope is to identify and gain leverage from structure in its domain. Household robots, office assistants, and logistics support systems, for example, will have to solve planning problems "in the wild", in contrast to most planning problems addressed today, which are carefully formulated by humans to contain only domain aspects actually relevant to achieving the goal. Generally speaking, planning in a formal model of the agent's entire "wild" environment will be intractable; instead, the agent will have to find ways to reformulate a problem into a more tractable version at run time.

Not only will such domains require an adaptive representation, but, adaptive aspirations as well. That is, if the agent is under time pressure to act, then, we must be willing to accept some trade-off in the quality of behavior. However, as time goes on, we would expect the agents behavior to become more robust and to improve in quality.

This work is about taking advantage of structured, relational action representations for planning. Our aim is to make small models of big domains in order to act efficiently. We will go about this simplification by reducing, if possible, the number of distinct entities and the number of alternative outcomes under consideration.

To reduce the number of effective entities (that is, domain objects), we adapt our representation: we identify the minimum set of predicates needed to represent our problem, and we identify logical equivalence classes of objects with respect to

those predicates. This, in turn, induces equivalence classes over the state space and allows us to handle a class of states together as a group. This has the important consequence of inducing a partition over the action space, as well. This constrained action space, in conjunction with a *determinizing* step, allows us to begin the search for plans in an informative subset of the decision space. Finally, we want to turn these optimistic plans into increasingly robust policies by incrementally expanding the state-space envelope, as well as the set of predicates used in representation, as time and resources permit.

There are two main themes in our approach: that of leveraging structured representations to maintain a small, compact model of a planning task, and that of managing time pressure by producing policies that improve in expectation with the amount of available computation time.

1.1 Handling imperfect representations

One source of difficulty in a complex domain is the existence of large numbers of objects that are either irrelevant to a given planning problem or, worse, relevant but unnecessary. We are often given these descriptions separately: a general domain description that describes the types of objects available, the kinds of relations and properties they can have, and the set of rules that describes the dynamics; and, a separate problem description that specifies a particular instance of the domain along with a specific planning objective.

A formal description of a domain instance may produce an overwhelmingly large action space even for a modest number of objects in the world. Consider an assembly robot, with a box of thousands of identical gears. The robot needs one of those gears to do its job, so none of those gears are irrelevant. But, because they are equivalent, it ought to be able to consider only a single one of them. Our goal in this work is to exploit the effective equivalence of objects in order to simplify planning. One way to do this is to consider objects to be similar if they share similar properties and have similar relationships to other, similar, objects. In principle, the set of properties

and relationships is given by the set of predicates listed in the domain description. However, if not all of these predicates are equally necessary to achieve the given goal, then considering them all would mean making unnecessary distinctions between objects. We would like to detect this phenomenon and consider distinctions among objects only with respect to the smallest possible set of predicates necessary to achieve a given problem. This would have the effect of reducing the effective size of the state space and, thus, speed up the planning process.

1.2 Managing time-pressure to act

For an agent operating in the real world, a time-crunch is a fact of life: no one will put up with a robot that takes forever to come up with the perfect way to put gears into boxes, for example. So, this work is also about how best to manage the computation of a strategy. When is it appropriate to make a plan, and when is it appropriate to compute a policy? By planning, we refer to the idea of computing a sequence of actions, to use once, for a given state/goal pair. A policy, on the other hand, is a mapping from all states in a space to the appropriate action; in general, we compute a policy when we expect to be given the same task repeatedly and need to consider any possible eventuality. When do we wish to do one instead of the other, and what can we say about the spectrum between the two? We have developed a technique that employs *envelope-based planners*, which are interesting in that they explicitly inhabit the space between a plan and a policy. The *envelope* refers to the subset of states, selected via some appropriate process, that form the basis for a small, approximate model of the agent's policy. Any planning approach can be used as this generating process: the envelope is then initially populated with the states from the plan. What follows is an *anytime* procedure [10] that elaborates on this initial set of states. An *anytime* algorithm is one that generates the best answer with the available information and allowable time; given more computational resources, it will be able to improve on its answer. This strategy allows an agent to make a partial policy that hedges against the most likely deviations from the expected course of action, without

requiring construction of a complete policy.

We cast our planning problem in the framework of Markov decision processes (MDPs) [52]. MDPs are a powerful formalism for framing sequential decision-making problems and are an active research area with a large spectrum of solution methods.

1.3 Representing planning problems as MDPs

The problem of planning has been an important research area of AI since almost the inception of the field. However, even its "simplest" setting, that of deterministic STRIPS planning, has been found to be PSPACE-complete. [9]. Nonetheless, traditional AI planning techniques are often able to manage very large state spaces, largely due to powerful logical representations that enable structural features of the state and action spaces to be leveraged for efficiency. On the other hand, work in the operations research community (OR) has developed the framework of MDPs, which specifically addresses uncertainty in dynamical systems. Being able to address uncertainty (not only in acting, but, also in sensing, which we do not address in this work) is a primary requirement for any system to be applicable to a wide range of real-world problems.

Our aim is to bring together some of these complementary features of AI planning and MDP solution techniques to produce a system that can build on the strengths of both. An important result that enables our approach is that problems of goal-achievement, as typically seen in AI planning problems, are equivalent to general reward problems (and vice versa). [45] Thus, it will be possible for us to take a given planning planning problem and convert it to an equivalent MDP.

1.3.1 Representation in MDPs

An MDP is a tuple, $\langle \mathcal{Q}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$ where: $\mathcal{Q}$ is a set of states; $\mathcal{A}$ is a set of actions applicable in each state; $\mathcal{R}$ is a reward function mapping each state to a real number: $\mathcal{R} : \mathcal{Q} \rightarrow \Re$; and $\mathcal{T}$, the transition function, gives the probability that a state and action pair will transition to another state: $\mathcal{T} : \mathcal{Q} \times \mathcal{A} \times \mathcal{Q} \rightarrow \Re$. A *solution* for an MDP consists in finding the best mapping from states to actions in a way that

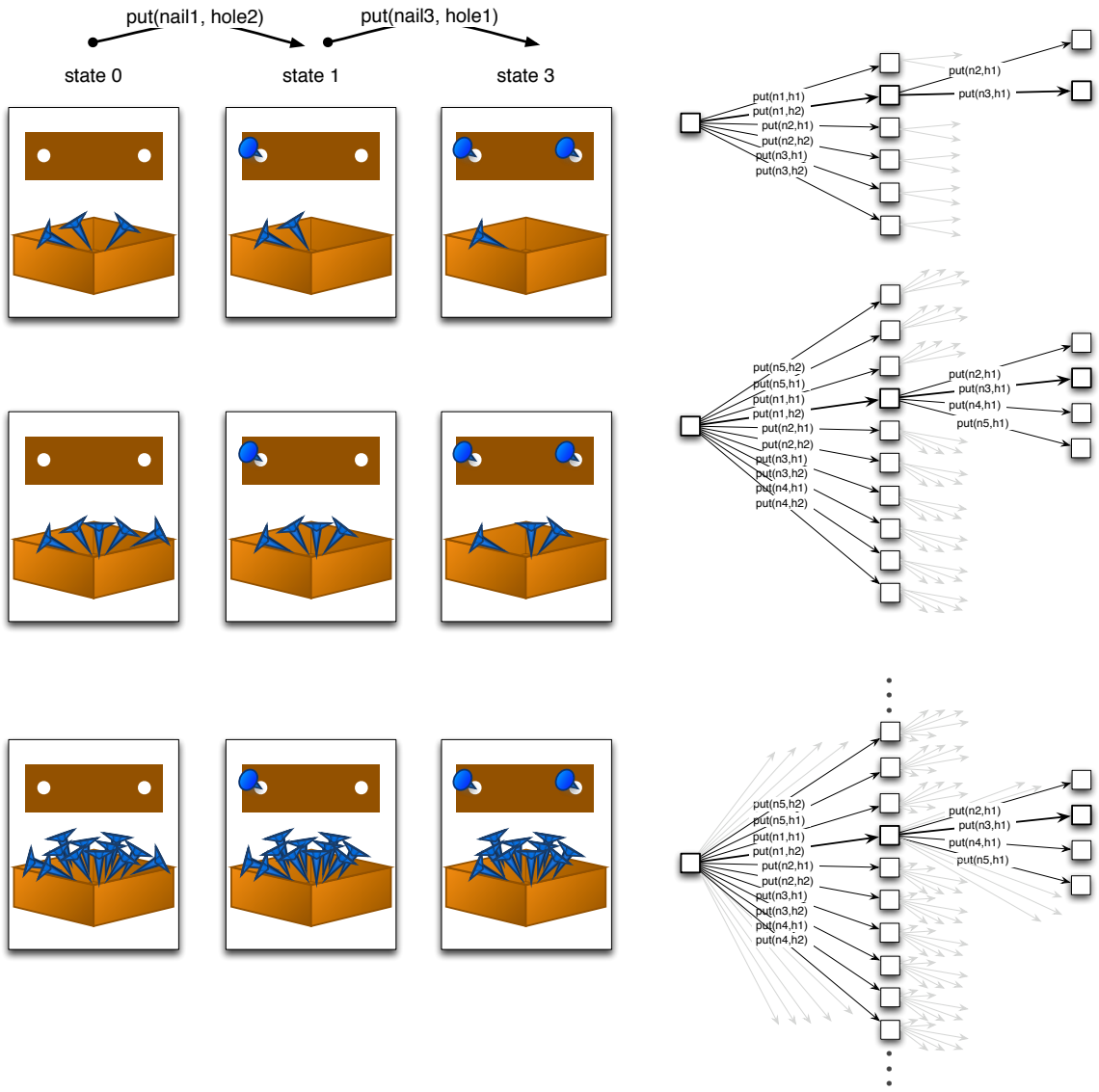


Figure 1-1: In this example, our task is to mount a board with a pair of nails. Given a box of nails, a solution can be carried out in two steps: first pick a nail from the box and put it in an empty position in the board, then, pick another nail from the box and put that in an empty position in the board. We would hope that this solution could be found in a way that is relatively insensitive to the exact number of nails in the box. We would like to avoid succumbing to the combinatorial growth experienced by naive search for the shortest path, shown in the corresponding search trees in the right column.

maximizes long-term reward. This function, π, is called a *policy*.

In the past, much work on finding policies for MDPs considered a state to be an atomic, indivisible entity; that is, one referred to state s_{124} without knowing anything further about its internal structure. More recently, advances have been made in representing a MDP states in terms of *factored* state spaces; that is, an particular state is seen to be a combination of state features. For example, in a two-dimensional grid domain, state s_{124} might be known to correspond to a particular x, y-coordinate, say $(3, 45)$. In this case, the state space is factored into two features: the value of the x-coordinate and the value of the y-coordinate.

Even though a factored state representation is an improvement over an atomic state representation, it still has its limitations. The state features as described above correspond to *propositions* about a state: e.g., the x-coordinate has value 3. This means that the policy π, the transition function T, and the reward function R must cover possible combinations of values of all of the state features. When there are lots of features, or if the features can take on a large range of values, the size of the state space grows combinatorially. As long as the transition function T and the reward function R have a compact representation that can be exploited for efficiency during planning, however, we can be relatively insensitive to the size of the state space.

In this work, we take advantage of a more compact way of representing state transitions (i.e., actions). That is, rather than a state being composed of a set of propositional features, we think of it as being composed instead of a set of logical relationships between domain objects. Since these predicates can make assertions about logical *variables*, a single predicate may in fact represent a large number of ground propositions. This lets us use a single transition rule to represent many ground state transitions.

1.3.2 Rule language

A well-specified planning problem contains two basic elements:

Domain Description : The domain description specifies the *dynamics* of the world,

the *types* of objects that can exist in the world, and the set of logical *predicates* which comprise the set of relationships and properties that can hold for the objects in this domain.

Problem Instance : To specify a given problem instance, we need an *Initial World State*, which is the set of ground predicates that are initially true for a given set of objects. We also need a *Goal Condition,* which is a first-order sentence that defines the task to be achieved. The goal condition is usually a conjunction, though disjunctive conditions are legal, and it may be universally or existentially quantified.

The dynamics of the domain must be expressed in a particular rule language. In our case, the language used is the Probabilistic Planning and Domain Definition Language (PPDDL) [59], which extends the classical STRIPS language [18, 44] to probabilistic domains. This allows for a very natural description of rule effects, such as conditional effects, negated preconditions, quantified effects, and so on.

In Figure 1-2 we see a full problem specification in PPDDL. In the top half of the figure is the domain description: the definition of types, objects, predicates, and rules for all problem instances. In the bottom half of the figure is a definition of a concrete problem instance: the objects, the initial state, and the goal. The overall set of objects (i.e., the universe of discourse) for a particular problem is composed of the objects given in the problem instance and the set of objects in the domain description – listing an object in the domain description is shorthand for saying that it will form part of all problem instances.

Next, let's look at the list of predicates in Figure 1-2 closely. Each $n-ary$ predicate takes a list of arguments of length n. We will use the term *predicate* to refer to a name of a property or relation, such as `on-top-of`; when a predicate is asserted over a list of arguments, such as `(on-top-of ?top ?bottom)`, that is properly called an *atom* or *relation*.

The names of variables begin with a question mark, so as to distinguish them from domain objects (or constants). The name of a variable may also be followed by a *type*

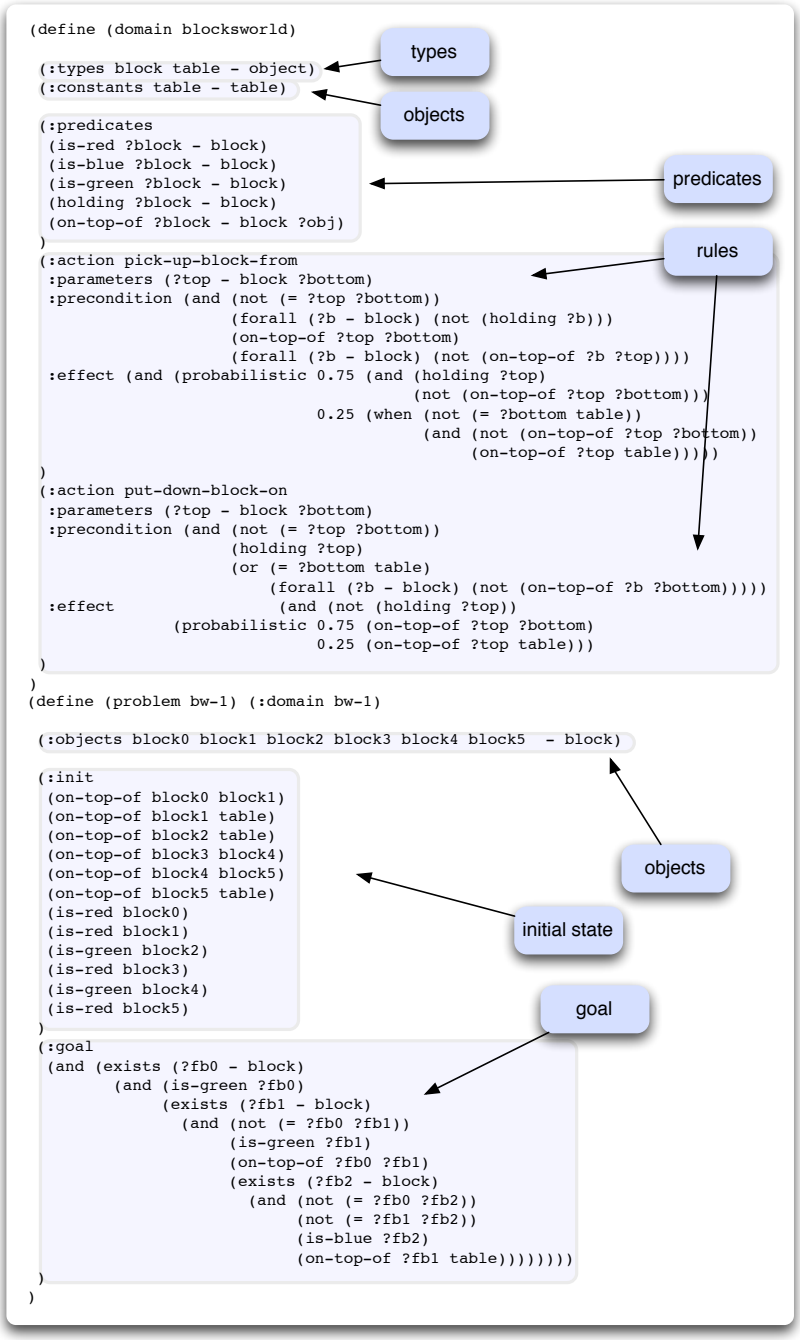


Figure 1-2: A complete PPDDL specification of a blocks-world planning problem.

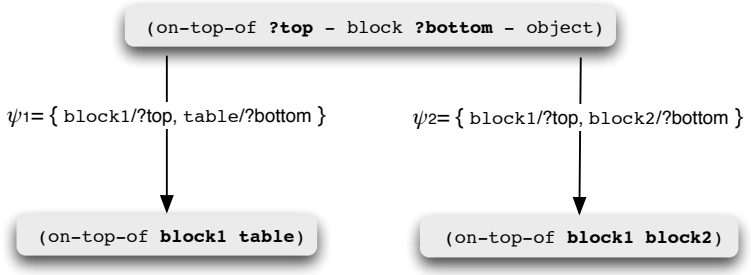


Figure 1-3: An example of two legal substitutions for the `on-top-of` predicate.

specificiation. In this case, writing `?top - block` means that the variable *top* should be of type "block"; not specifying a type means the variable can be of any type. The types may be defined in a hierarchical way: in our example, the types "block" and "table" are subtypes of the "object" type. Types are important because they restrict the ways in which variables can be *bound* to domain objects. Consider Figure 1-3. In this figure we have the relation `(on-top-of ?top ?bottom)` and two different *substitution*, or, *binding* lists, ψ_1 and ψ_2, which are assignments of each variable to a replacement. Applying ψ_1, for example, produces the ground atom, or *proposition*, `(on-top-of block1 table)`. Note that it would not be a legal to substitute `table` (which is of type "table") for *top*, which is a "block"; however, since *bottom* is of type "object" and "table" is a subtype of object, replacing *bottom* with `table` is perfectly fine. Once we have produced proposition, we can assign a truth value to it.

A logical atom or sentence is said to be *ground* or *closed* when all of its variables have been ground to domain objects; conversely, and atom or sentence is *unground* or *open* if it has at least one free occurrence of a variable — i.e., an occurrence of a variable that is neither ground nor constrained by quantification.

Finally, a domain description must define some rules. A rule is essentially a complex logical sentence (an implication, if you will). It is composed of: a set of arguments or parameters — these are the free variables; a *precondition*; and, an *effect*. The precondition specifies the condition that must be true in a state s in order to be able to apply that rule in the state s. The effect specifies the changes (or, a distribution over sets of changes) that occur to s as the result of applying the rule.

Now we consider an example of how to use a rule. Say we are in the initial state, s_0, as given above; which rule applies? There are many ways to go about this correctly. Our approach will be as follows:

1. Put the rule's preconditions and and effects into conjunctive normal form. Because we have a finite "universe," we can partially ground any universally quantified clauses by re-writing them them as a conjunction of n clauses, where n is the number of domain objects and where the i^{th} domain object substitutes for the quantified variable in the i^{th} clause. This process transforms the precondition into a conjunction of atoms, ρ.

2. A rule applies in a state if its precondition is true in the interpretation associated with the state. Recall that states are represented as conjunctions of `true` ground atoms (and propositions omitted from the state are assumed to be `false`). Thus, determining if an antecedent is true in a state reduces to finding a subset of the state's ground atoms with which the antecedent can be unified. This unification produces a binding list, ψ (or set of binding lists, if there is more than one way to unify the antecedent).

3. Apply the substitution ψ to each effect to produce a conjunction of atoms, η corresponding to the effect. Care must be taken in the case of conditional effects: in this case, the condition on the effect is another conjunction of atoms ρ'. To determine if the effect will be triggered, we append ρ' to ρ, and recommence the search for a unification of the new, composite ρ given the contents of the existing binding, ψ. If there is no such unification, an empty sentence is returned; otherwise, a conjunction of atoms corresponding to the effect is returned.

4. Compute the resulting state. Given the starting state s, we remove from s all the propositions that are *negated* in η (this is also referred to as the "delete list," for obvious reasons), and, we add in all the non-negated propositions (i.e., the "add list"). To be formal, we write:

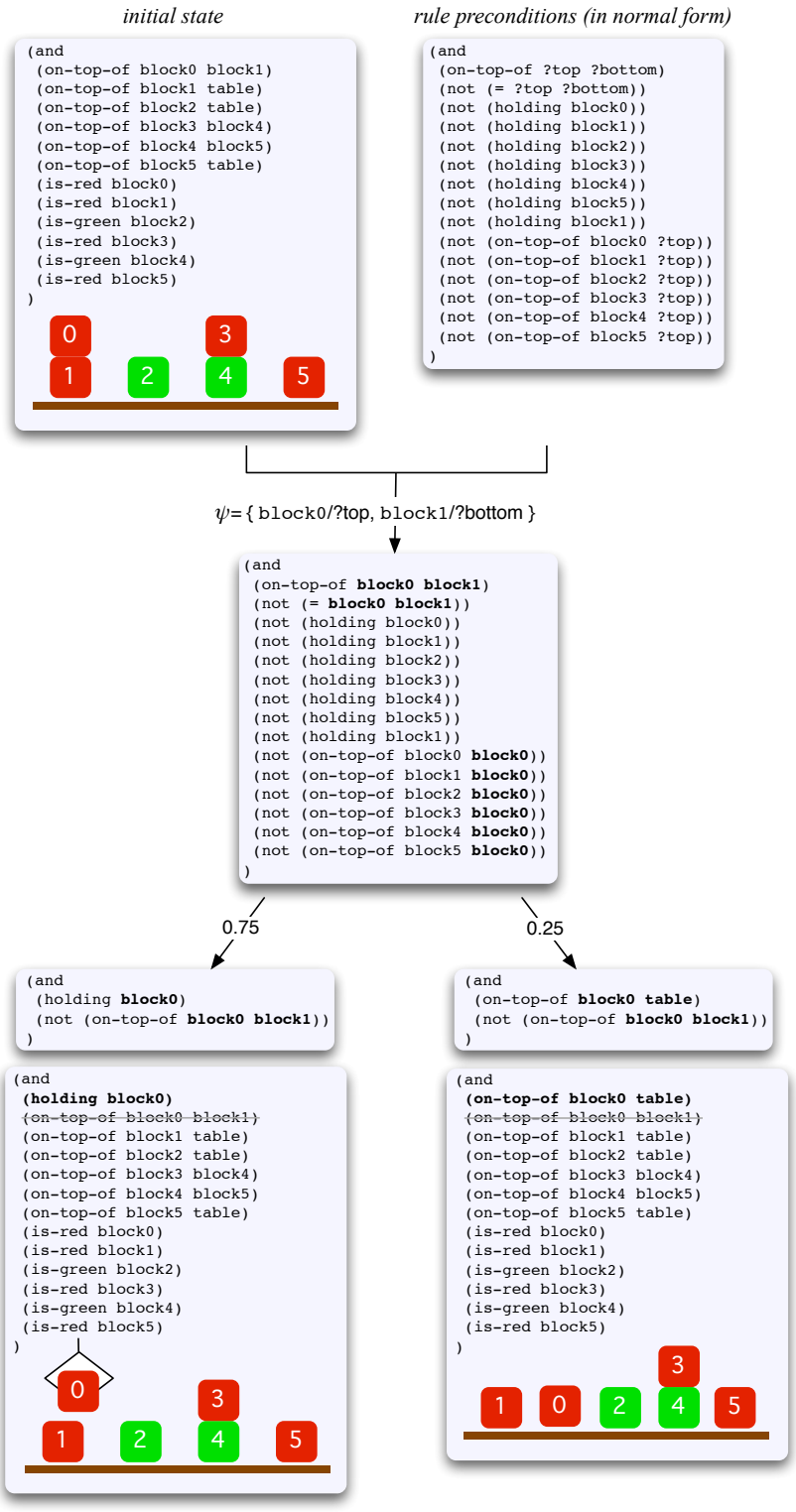


Figure 1-4: Applying the `pick-up-block-from( block0, block1 )` action in the initial state.

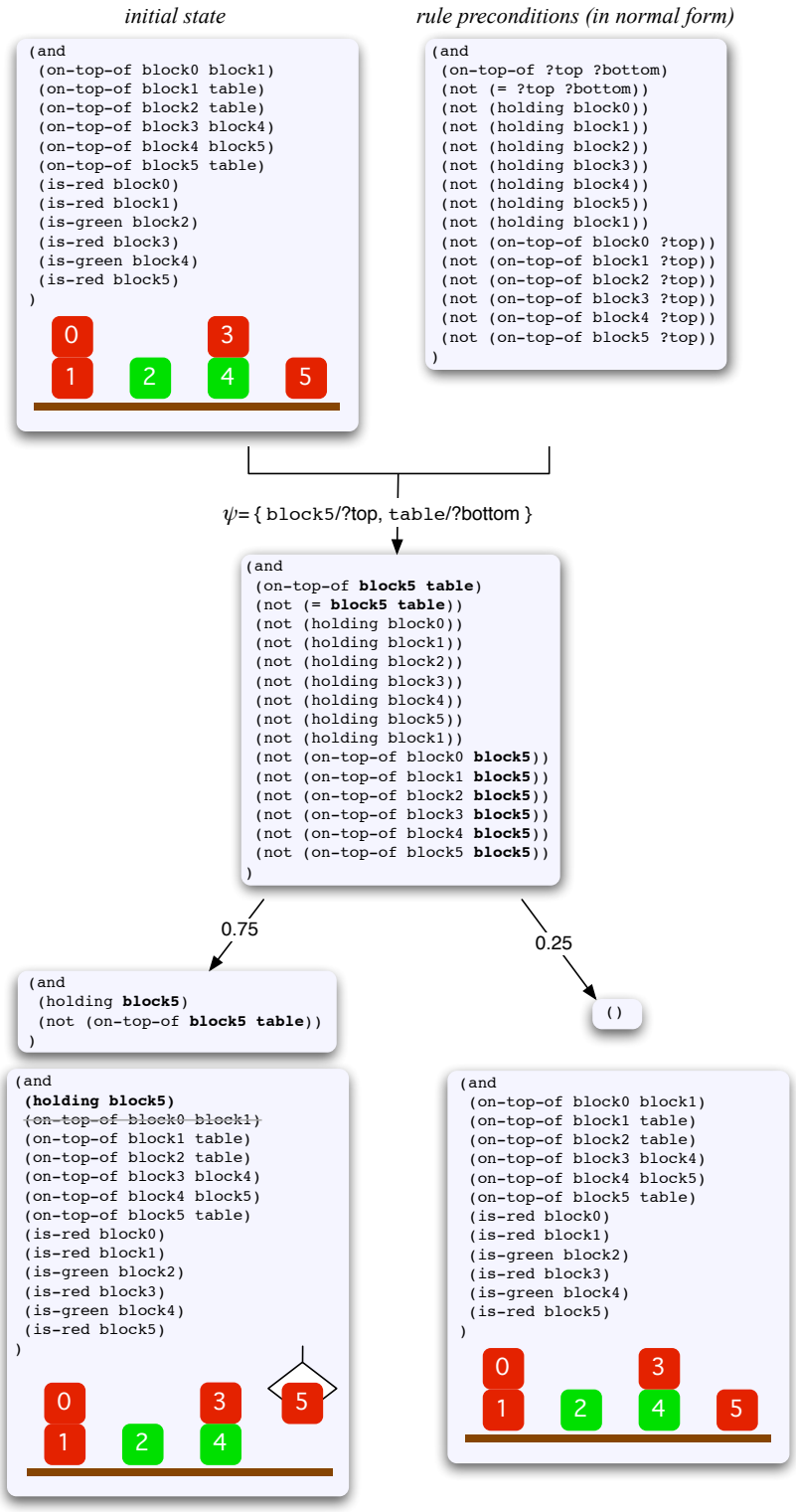


Figure 1-5: Applying the `pick-up-block-from( block5, table )` action in the initial state.

Figure 1-4 and Figure 1-5 show two different examples of this procedure given the initial state s_0, the rule `pick-up-block-from`, and two possible binding lists. The first figure shows the case of picking up block 0 from block 1, which yields a distribution over two possible states: one with probability 0.75 in which we succeed in picking up the block, and one with probability 0.25 in which we drop the block on the table. The second figure shows the case of picking up block 5 from the table. Here, the condition `(not (= ?bottom table))` was false with {`table`/?*bottom*}. So, this means there is probability 0.75 of picking up the block, and, probability 0.25 of *no change*.

We will say "rule" or "operator" when we mean an open or unground rule such as they appear in the domain description, and we will say "action" to denote a ground instance of a rule.

The benefit of supporting the PPDDL formalism is access to the benchmark planning domains, such as those used in the ICAPS planning competitions held over the past few years. This permits our work to be more directly comparable to related approaches.

1.3.3 Encoding Markovian dynamics with rules

As mentioned above, an MDP is traditionally defined as a tuple, $\langle \mathcal{Q}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$ where: $\mathcal{Q}$ is a set of states; $\mathcal{A}$ is a set of actions; $\mathcal{R}$ is a reward function; and $\mathcal{T}$ is a transition function.

As a step towards working with more compact models of a domain, we define a relational MDP (RMDP) as a tuple $\langle \mathcal{P}, \mathcal{Z}, \mathcal{O}, \mathcal{T}, R \rangle$:

States: The set of states $\mathcal{Q}$ is defined by a finite set $\mathcal{P}$ of relational predicates, representing the relations that can hold among the finite set of domain objects, $\mathcal{O}$. Each RMDP state is an *interpretation* of the domain predicates over the domain objects. By *interpretation*, we mean a mapping from all ground predicates to truth values. For example, given the atom *on(A,B)* and domain objects `block1` and `block2`, we would produce the propositions `on(block1, block2)` and `on(block2, block1)`, which might be respectively assigned {`true, false`}, {`false, true`}, {`false, false`}, but probably not {`true, true`}.

Actions: The set of ground actions, likewise, depends, on the set of rules $\mathcal{Z}$ and the objects in the world.

Transition Dynamics: For the transition dynamics, we use a compact set of rules based on the standard Probabilistic Planning and Domain Definition Language (PPDDL) [59] as discussed above. To briefly review, a rule's behavior is defined by a precondition and a probabilistic effect, each expressed as conjunctions of logical predicates. A probabilistic effect describes a distribution over a disjoint set of logical outcomes. A rule applies in a state if its precondition is true in the interpretation associated with the state. Each outcome then describes a possible resulting ground state. In our system, we currently use rules that are designed by hand; they may, however, be obtained via learning [60, 50].

For each action, the distribution over next states is given compactly by the distribution over outcomes encoded in the rule schema. The rule outcomes themselves usually only specify a subset of the domain predicates, effectively describing a set of possible resulting ground states. To fill in the values of the domain predicates not menioned in the outcome, we assume a static frame: state predicates not directly changed by the rule are assumed to remain the same.

Rewards: A state is deterministically mapped to a scalar reward according to function $R(s)$. This can be given as, say, a list of conjunctions associating particular conditions (for example, the goal condition) with a scalar reward or penalty.

Given this basic understanding, we can now begin to put together the pieces of our approach.

Chapter 2

Preliminary notions

As alluded to above, the difficulty of planning effectively in complex, ongoing problems is maintaining an efficient, compact model of the world in spite of potentially large ground state and action spaces.

2.1 Envelope-based Planning

Plexus works by considering a subset of states with which to form a restricted MDP, and then searching for an optimal policy in this restricted MDP. The state space for the restricted MDP is called the *envelope*: it consists of a subset of the whole system state space, and it is augmented by a special state called OUT representing any state outside of the envelope. The algorithm then works by alternating phases of *envelope alteration*, which adds states to or removes states from the envelope, and *policy generation*, which computes a policy for the given envelope. In order to guarantee the anytime behavior of the algorithm, Dean *et al.* extensively study the issue of *deliberation scheduling* to determine how best to devote computational resources between envelope alteration and policy generation.

A small example of refining an initial plan is shown in Figure 2-1, which consists of a sequence of fringe sampling and envelope expansion. A complete round of deliberation involves sampling from the current policy to estimate which *fringe* states — states one step outside of the envelope — are likely. The figure shows the incorpora-

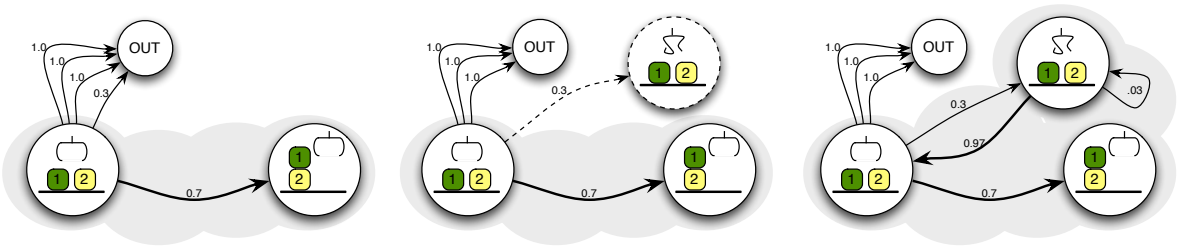


Figure 2-1: A tiny example of envelope-based planning. The task is to make a two-block stack in a domain with two blocks. The initial plan is consists of a single *move* action, and the initial envelope (far left) reflects this action sequence. The next step is to sample from this policy, and the potentially bad outcome of breaking the gripper is noticed (middle). After expanding the envelope to include this outcome, the policy is revised to include executing a "repair" action from the newly incorporated state (far right).

tion of an alternative outcome of the policy action in which the gripper breaks. After the envelope is expanded to include the new state, the policy is re-computed. In the figure, the policy now specifies the fix action in case of gripper breakage.

Thus, deliberation can produce increasingly sophisticated plans. The initial planner needs to be quick, and as such may not be able to find conditional plans, though it can develop one through deliberation; conversely, searching for this conditional plan in the space of all MDP policies, without the benefit of the initial envelope, could potentially have taken too long.

The Plexus algorithm was originally developed for atomically represented robot-navigation domains, which generally have the characteristics of high solution density, low dispersion rate (i.e., a small number of outgoing transitions at each state), and continuity (i.e., that the value of a state can be reasonably estimated by considering nearby states). These features made it reasonable to execute a depth-first search in order to find the first set of states for the envelope. Arbitrary relational planning domains may not necessarily share these characteristics.

The principal observation here is that, since our planning domain is expressed as an RMDP with the transition dynamics as a set of logical rules, then why not exploit the vast array of techniques from classical AI planning to find an initial envelope efficiently? The structure of the logical rules can also be taken advantage of in the

envelope elaboration, as well.

2.2 Finding the initial envelope

Given that we have a set of rules and the problem description, the next step in envelope-based planning is finding the initial envelope. We know that in a relational setting, the underlying MDP space implied by the full instantiation of the representation is potentially huge.

When a PPDDL-style rule is grounded in a domain, it yields an exponential number of actions as the number of domain objects grows. Since large numbers of actions will grind any forward-searching procedure to a halt, we want to avoid considering all the actions during our plan search. Constraining the search appropriately will be essential in this phase.

In Chapters 4 and 5, this will be discussed in detail. But for now, we note the essence of the issue: To cope with a potentially large branching factor, we have identified a technique called *equivalence-class sampling*.

We partition into equivalence classes the actions that produce *similar* effects with respect to our basis set of predicates. We will define this similarity in some detail further on. The forward-search can then proceed by considering only a *canonical* action from each class. The canonical action, which we will define in detail later, is representative of the effects of any action from that class. Eliminating redundant actions in this way has the potential to significantly reduce the branching factor. In addition, we can carry out an analysis based on the predicates necessary to achieve the goal to identify a minimal set of domain predicates that can be used to produce a solution. Such a minimal effective representation, if one exists, can further help mitigate the combinatorial effects of searching in large domains. As we will see later, this procedure happens dynamically and is informed by the given start state and goal condition.

2.3 From a plan to a policy

Turning the initial trajectory into a space-efficient MDP in which to do *policy generation* will also require some care, which we will discuss in Chapter 6. We must go from the sequence of actions returned by initial plan to a set of states that will comprise our envelope MDP.

In essence, we compute the set of MDP states iteratively by applying the sequence of actions in our plan starting from the initial state. An important feature of the transformation, however, is that transitions that initiate in an envelope state but do not land in an envelope state are redirected to a special OUT state. The leftmost MDP in Figure 2-1 shows this for a small example task.

Envelope expansion, or *deliberation*, involves adding to the subset of world states in our envelope MDP. When the envelope is created from the initial trajectory, only the highest-probability outcome is considered for each action. In the deliberation phase, we analyze what the other probable outcomes might be for the actions in our policy. If we find that a particular other outcome is relatively likely, or carries a high penalty, we may wish to incorporate that outcome into our envelope MDP.

Importantly, we would like the resulting MDP to take advantage of the minimal representation discovered in the previous step. This will mean that each MDP state becomes an abstract state, actually representing a set of possible underlying states. This will have implications for how we compute transition probabilities and, thus, compute desirable policies.

To implement envelope-based planning in relational domains, then, we need a set of probabilistic relational rules, which tell us the transition dynamics for a domain; and we need a problem description, which tells us the states and reward. Together, the description of a domain and a problem instance fully specify our planning task. Figure 2-2 gives a high-level schematic for the REBP system. As we will see, using the equivalence-based envelope method, we can take advantage of relational generalization to produce good initial plans efficiently, and use envelope-growing techniques to improve the robustness of our plans incrementally as time permits. REBP is a plan-

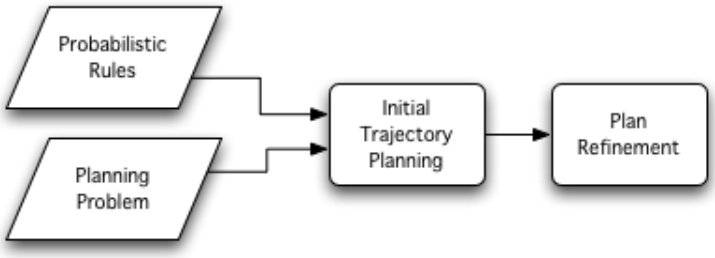


Figure 2-2: A high-level schematic of the REBP planning system. There are two main inputs to the system: a set of probabilistic rules. and a description of the planning problem. The next process is to find an initial plan quickly. The final process is to refine the initial plan as resources permit.

ning system that tries to dynamically reformulate an apparently intractable problem into a small, easily handled problem at run time.

Chapter 3

Background and Related Work

In the next few sections, we cover some background material and review past work as it relates to specific facets of our approach.

3.1 Representation issues in relational MDPs

As we saw previously, structured representations for MDPs allow the expression of world dynamics in a compact, problem-size independent way. However, when it comes time to plan with such representations, many approaches end up working with propositionalized versions of the problem, because doing so lends itself to established solution techniques. What representational and algorithmic techniques can we use to avoid working directly in potentially large ground state and action spaces?

3.1.1 First-order RMDP methods

One way of working with a model whose size is independent of the problem instance, of course, is never to ground the model at all.

Some recent approaches towards solving MDPs describe states in terms of logical formulae. They use structured representations of world dynamics to estimate the value of a state without ever resorting to a ground description.

In one of the earlier papers on this topic, Boutilier *et al.* find policies for first-

order MDPs expressed in the situation calculus. Their technique is called *symbolic dynamic programming* (SDP), and is based on *first-order decision-theoretic regression* (FODTR) [8]. SDP solves for the value-function of a first-order domain by manipulating logical expressions that stand for sets of underlying states. These logical statements express states that have the same value function; by regressing that expression backwards through a logical action operator, one obtains an expression and a value for the states in that action's pre-condition. Keeping the set of regressed expressions compact — a new logical formula is created for every action that might have produced a given state — seems to be hard and to require complex theorem-proving. It may be that in the worst case, the algorithm produces one logical expression corresponding to each ground state. In subsequent work, Sanner and Boutilier build on the FODTR work with *approximate linear programming* for First-order MDPs (FOALP). The idea is to represent the value function of a domain more compactly, as the linear combination of a set of logical basis functions. Each basis function is intended to represent some aspect of the goal and combines additively with the other basis functions. Sanner and Boutilier develop a first-order version of the approximate linear programming algorithm to solve for the required weights. This approach has the advantage of avoiding the combinatorially exploding representation of the value function of SDP and of other approaches using an exact value iteration approach [30, 39]. However, the computational challenge shifts to deriving and ensuring consistency of the potentially large number of constraints (since there are, in principle, an infinite number of situations to consider). Sanner and Boutilier show some experiments in a simulated elevator domain, and note that the FOALP approach produces policies which evaluate favorably compared to some intuitive heuristically-guided policies. They do also note that running times in their experiments ranged from five minutes with one basis function to two hours for six basis functions. Unlike many MDP approaches, however, FOALP does provide bounds on the value functions.

Grossman *et al.* [30] also propose a symbolic version of the value iteration algorithm; instead of the situation calculus, however, their approach uses the fluent calculus. The situation calculus expresses a state as the result of a sequence of ac-

tions starting from the initial state; thus, to access the value of a fluent, a situation term must sometimes be "unrolled" until the value of the fluent is determined. In contrast, the fluent calculus manipulates the values of the fluents directly and expresses a state as the collection of fluent values. It is essentially a predicate logic. Otherwise, the idea is essentially the same as before; i.e., to find a minimal, symbolic partition of the state space and associate each partition with the correct utility value. Like Boutilier *et al.*, Grossman *et al.* model a stochastic action as a collection of "deterministic" actions, the choice of which is under "nature's control"; each component outcome is associated with a probability of being "chosen." As before, the algorithm starts from the goal and regresses through each action operator to obtain an expression and a value for preceding states. The same complexity issues with maintaining the compactness of the regressed expressions are also factors in this approach.

Symbolic treatments of First-order MDPs are attractive in the promise of being able to represent a value function logically, independently of the size of the ground state space. However, in practice, it seems that the computational challenges are significant. In fact, propositional approaches are still extremely compelling in practice due to their simplicity and well-understood behavior.

3.1.2 Trading off first-order and fully-ground

So, on the one hand we have propositional approaches which are simple but suffer from combinatorial explosion; and, on the other hand we have symbolic approaches which seek to maintain a small representation, but, involve considerable computational heft.

The underlying message is nevertheless clear: the more an agent can compute logically and the less it attends to particular domain objects, the more general its solutions will be. We propose a middle path: we agree to ground things out, but in a principled, restricted way. We will represent world dynamics by a set of relational rules. Relational representations allow the structure of the domain to be expressed in terms of object *properties* rather than object identities and thus yield a much more compact representation of a domain than the equivalent propositional version can. Furthermore, relational representations permit the exploitation of structure in the

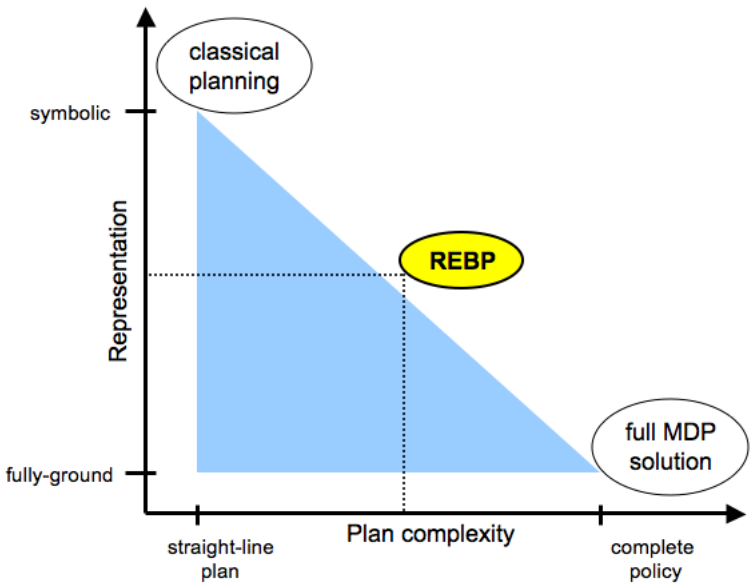


Figure 3-1: Our approach explicitly inhabits the space between fully ground and purely logical representations, and between straight-line plans and full MDP policies.

dynamics: for example, in a blocks world, it often does not matter which block is picked up first as long as a stack of blocks is produced in the end. If it were possible to identify under what conditions actions produce equivalent kinds of effects, the planning problem could be simplified by considering a representative action (from each equivalence class) rather than the whole action space. That is one way in which we can limit the impact of grounding a domain. Then, by extending the envelope method of Dean *et al.*[13] to use these structured dynamics, we can work with an *envelope* of states and refine the policy by gradually incorporating nearby states into the envelope. This approach avoids the wild growth of purely propositional techniques by restricting attention to a useful subset of states. Our approach strikes a balance along two axes: between fully ground and purely logical representations, and between straight-line plans and full MDP policies.

3.2 Planning in a deterministic model

Part of our job in implementing the envelope method of Dean *et al.*[13] will be to compute an initial envelope of states. To make this computation efficient, we consider

making a plan in a determinized version of our domain model: we simplify the original probabilistic model by assuming the *most likely* effect as the only effect for each rule.

In the case of deterministic actions, a solution plan is said to exist if there is a sequence of actions that leads from the starting state to the goal. In the case of stochastic actions, however, we have no control over the actual outcome; we only know the distributions over outcomes. What does it mean for a solution straight-line plan to exist in this case? Since a straight-line plan considers only a single outcome at each step, we designate the *anticipated* outcome to be the one expected by the planning procedure; usually, this is just the most likely outcome. Thus, in this case, a plan exists if there is a sequence of actions whose expected outcomes yield a state sequence that leads to the goal. We accept as a goal state any state that entails the goal conditions g.

Because the envelope algorithm will later consider devations from the original set of states, planning with deterministic approximation of our original model provides Once we have taken this step, our task reduces to a deterministic planning problem. We will need to take advantage of the structure of the domain dynamics in order to plan efficiently, however, since combinatorial explosion may still produce unwieldy state and action spaces.

The reduction of action space in order to plan more efficiently is an idea that has a rich history. We try to improve upon these past approaches by doing the reduction dynamically and in the presence of arbitrarily complex relational structure. We survey some relevant past work in the following sections.

3.2.1 Reduction of action spaces

A great deal of work in identifying action "symmetries" to reduce the effective action space has come out of work on constraint satisfaction problems (CSPs). Constraint satisfaction refers to the problem of deciding what values it is possible to assign to a set of variables such that the specified equality and/or inequality constraints are maintained.

Ellman [15] describes an approximation technique for solving constraint-satisfaction

problems (CSPs). In his problem, there are objects in the domain that must be assigned to variables: for example, a set of jobs, each of a certain duration, that must be assigned to start times in such a way that all jobs finish before a deadline; or, a set of balls of a certain weight, that must be assigned to one of two partitions, such that the total weight in each partition is equal. The objects in the domain have one characteristic measure (such as duration or weight), and these objects are clustered with respect to this measure in order to obtain a prescribed number of equivalence classes. These classes of "symmetric" objects enable entire subsets of solutions to be pruned: if a particular assignment is not a solution to the CSP, it is not necessary to test an assignment that just permutes the assignments of variables to the objects in the class.

While finding equivalence classes of objects seems like the right idea in spirit, the use of a simple feature to guide the clustering will be insufficient in truly relational domains.

Joslin and Roy [38] use the idea of isomorphisms to detect symmetry in planning problems represented as constraint-satisfaction problems. Importantly, this computation is done as a pre-processing step (rather than in-line) and goals must be specified as fully ground sentences.

In the context of planning, the work by Fox and Long [19, 20, 21] attempts to identify symmetries in action effects in order to prune the action space. This work is the most closely related to ours.

Fox and Long first present a notion of symmetric states that is used to simplify planning [19]. That is, if two actions result in symmetric outcomes, then it is only necessary to consider one of them in planning.

Object equivalence is defined as follows. Two objects are defined to be equivalent if they have the same initial and final properties and attributes. The example in the paper is of the gripper domain: There are two rooms, a robot with two grippers, and a number of balls begin in one room. The task is to move all the balls to the next room. This problem is highly symmetric; however, unless ball identities are abstracted away, the search for a plan becomes mired in permuting the different

orderings of when particular balls are moved.

Object equivalence is established at the beginning of the planning process. Two actions are considered to be symmetric if their parameter lists contain objects drawn from the same symmetry collections. Once an object is acted on, it loses symmetry with its original group. Fox and Long describe the different bookkeeping techniques needed to track which objects belong to which symmetry group as planning proceeds. The main shortcoming of the technique as described here is that symmetry groups are not re-computed as planning progresses, and thus the advantage gained by the computation is quickly lost as different actions are tried.

In the follow-up paper [20], Fox and Long define object symmetry as before. Two objects are *functionally identical* if they share identical initial states and can make only identical transitions. That is, two objects are identical if they are never explicitly named in any operator schema, and if substituting one for the other yields an identical problem description. However, now, their algorithm is capable of maintaining symmetries during the planning process. This is achieved by replacing the previous level-independent data structure with a level-dependent one. In this context, "level" refers to a level in the plan-graph construction. The experiments reported are on a variation of the Gripper domain, extended to six rooms.

In the 2005 paper, the authors show experiments in a blocks world domain. However, there is an important element introduced here, that of an *almost-symmetry* among objects. That is, objects can be made *almost symmetric* by abstracting away the specific domain objects to which they are related: for example, it matters that `block 1` is *on* something, but, it doesn't matter what. Then, `block 1` can be considered almost symmetric to any other block that is also *on* something, regardless of what that something is or how many there are.

The next step in Fox and Long's approach is to figure out which objects are equivalent by analyzing their relations in the initial state and goal condition. Then, the search for a plan uses an existing heuristic, called the FF "helpful actions" heuristic, [36] to put at the top of the list the actions that work on objects almost-symmetric to ones that have already been worked on. Intuition is that if it has been determined

to be good idea to take an action on `block 1`, then down the line, it might be a good idea to do the same thing with an equivalent block.

The ideas presented are appealing: one would like to be able to identify equivalent states and eliminate plans that traverse equivalent sequences. And, the authors do prove soundness and completeness of their approach. Nevertheless, there is a rather weak notion of what makes objects equivalent: it relies essentially on unary properties of objects rather than their participation in a complex web of relationships. Furthermore, the notion is incompatible with a first-order statement of a goal: the calculation of symmetry requires analyzing an object's properties both in the initial state and a goal state, so, a fully ground goal state is required. It would not be feasible to specify a goal condition such as, "put all of the blocks on the table into a stack", which actually specifies a number of satisfying ground states.

The goal of Haslum and Jonsson [34] is also to reduce the number of operators (actions) in order to reduce the branching factor and speed up search. They define the notion of redundant operator sets. Intuitively, an operator is redundant with respect to an existing sequence of operators if it does not produce any effects different from those already produced by the the sequence. The set of redundant operators, considering sequences up to a pre-determined length, are computed before starting to plan; however, this is a computation that is PSPACE-hard in general. An approximate algorithm is also given. In the familiar blocks-world, for example, this method would remove an atomic *move* action, since its effects would be redundant to the two-step sequence of *pickup* and *putdown* actions. Planning efficiency increases when such redundancies are found, even though their presence is a function of a given domain specification and perhaps not a fundamental characteristic of the problem. A search for this type of redundancy is something that could be used in combination with our algorithm, since each approach seeks redundancies of different kinds.

Other related work is that of Guere and Alami [31]. In their approach, they define the idea of the "shape" of a state. A state "shape" is in some sense a notion of equivalence: the main idea is if there is a substitution of one ground state's object names for another state's object names that produces the same list of facts, then

the two states have the same "shape". An algorithm is given to try to construct all the "shapes" for a particular domain instance. To extract a plan/solution, it looks for an action that connects a state in the starting "shape" to a state in the goal "shape". There is potential for concern in that one needs to have computed all the members/substitutions of the "shape" classes offline for a particular domain instance (say, for a blocks world with 50 blocks). In our work, by contrast, we try to estimate equivalence classes on the fly given the current state of the search. As a result, we can avoid considering shapes of states that are in very distant parts of the state space from the initial state.

3.2.2 Heuristic search methods

Conceptually, our strategy is also related to MDP planning algorithms that take advantage of a known starting state and a heuristic estimate of state values in order to avoid exploring an entire state space for a solution. This is a very powerful strategy, since, by conditioning on a known initial state, we can avoid parts of the state space that are "far" from the path between the initial state and the goal.

In one of the first approaches of this type, Real-Time Dynamic Programming (RTDP) [1], each trial simulates a greedy policy until the goal is reached (or some number of steps have passed), and then it updates the value function only over the visited states. This is also described as "asynchronous value iteration", since not all states are updated on each round of the algorithm. Because it uses a heuristic to estimate the value of unknown states, it is quick to produce good policies but slow to converge on the whole state space since only "good" states are visited. RTDP is a real-time algorithm, in the sense that execution and value-function updates can be interleaved. It converges with probability one to an optimal policy for any state s in the known set of starting states. RTDP assumes a complete and accurate model of its environment, which is specified as an atomic MDP; or, it can also learn this model as it goes along (a variant called *adaptive* RTDP). Barto *et al.* show results of some experiments in their "race track" domain, a grid-based control problem in which a simulated car must decide how to use its actions (accelerating or braking in one of

the cardinal directions) to most quickly reach a "finish line" given a set of states that form a "starting line". The "shortest path" heuristic was used to ranke candiate states.

A subsequent algorithm, LAO* [32], again uses a heuristic to estimate the value of a state. It is based on the classic A* algorithm, which returns solutions in the form of a sequence of actions; LAO* finds solutions that may take the form of cyclic graphs. In contrast with RTDP, it is an offline algorithm: i.e., it searches for a solution before beginning execution. It searches in the space of directed, possibly loopy graphs, in which each node is a state and each arc is an action transition. It searches by building a path from the start state to a non-terminal state, called a partial plan, whose value is given by the heuristic estimate for the so-called *tip* state. On each iteration of the algorithm, LAO* chooses the best partial plan, and non-deterministically chooses which action transition out of the tip state from which to continue the search. DP backups are done only on the expanded part of graph. LAO*, by virtue of its descendence from A*, inherits a number of techniques for efficient search and thus converges more quickly than the original RTDP. Like RTDP, LAO* is designed to solve stochastic shortest-path problems given a start state and to find a solution that minimizes the expected cost of reaching a goal state. LAO* can also be extended to solve infinite-horizon problems. LAO* converges to an ϵ-optimal solution after a finite number of iterations, given an admissible heuristic evaluation function. Hansen and Zilberstein evaluate LAO* on the "race track" problem of Barto *et al.* and show that it is able to converge faster than RTDP.

A more recent approach, Labeled Real-Time Dynamic Programming (LRTDP) [6], is an improvement that speeds up convergence of the original RTDP algorithm. It does this by labeling a state and its downstream as solved, so it can focus on updating the values of unvisited states.

Heuristic planning approaches share our objective of seeking to avoid the evaluation of a whole, potentially large, state space by using a heuristic estimate. However, computing a good heuristic can often be rather expensive, and it may not always be obvious how to choose a good heuristic. Furthermore, these approaches are agnostic

on the problem of potentially large action spaces, which is impossible to ignore for relational domains.

3.2.3 Dynamic Replanning, or Plan Repair

This body of work consists of heuristic methods for robotic path planning for domains in which the dynamics, or knowledge about the dynamics, may change. These approaches usually consist of first finding an initial path with a traditional search, and when new information about the domain is obtained, considering only a subset of the MDP states for re-evaluation [17, 41, 55]. Replanning, or repairing an existing plan is a powerful idea: when the unexpected happens, it is often more efficient to repair an established plan rather than to restart from scratch.

One of the first algorithms in this line is the Focussed D* algorithm by Stentz [55]. The focussed D* algorithm first finds a path using an A*-like search; then, as the robot moves and updates its path-cost information, the planned path is incrementally modified by considering the remaining discrepancy between the goal and the robot's *current* position. The algorithm is characterized as a generalization of A* to domains in which the dynamics change over time. However, like in A*, the domain dynamics are assumed to be deterministic.

The D* Lite algorithm [41] is similar in spirit to Focussed D*, but is algorithmically different. It is based on the Lifelong Planning A* algorithm (LPA*) [40, 42], which dynamically re-computes the shortest path from start to goal as new information about path costs is received. D* Lite improves on LPA* to dynamically recompute the shortest path from the goal to its current position. In addition to being at least as efficient as Focussed D*, D* Lite is accompanied by an extensive theoretical analysis by virtue of its foundation on LPA*[42]. D* Lite also assumes a deterministic domain.

To deal with non-deterministic MDP domains, Ferguson and Stentz propose a dynamic programming (DP) algorithm called Focussed Dynamic Programming (FDP) [17]. The idea is to choose some subset of states for DP updates, and then to to order those updates in the most effective way. FDP uses two heuristics: a less expensive heuristic decides which states get inserted into a queue, and a relatively more expensive one

is used to prioritize the states in the queue. The basic FDP algorithm uses these two heuristics to avoid computing values for all states in the MDP when planning a path from start to goal. Three changes are needed in the basic algorithm to allow an existing plan to be repaired when new information is received: the states' value estimates need to be revised if path knowledge changes; the algorithm must keep in consideration all states that might possibly become relevant; and, computation must be focused on the robot's current position (which is constantly changing).

So far, the work in the area of plan repair depends crucially on the use of grid-based coordinates, since the heuristics that estimate path distances must be able to measure the distance between two positions.

3.3 Equivalent transition sequences

The correctness of our overall approach relies on the idea that there can be equivalent transition sequences in a dynamic system, and that a characteristic of a particular transition sequence (such as reaching a goal state, or, failing to reach a goal state) is true of any transition sequence to which it is equivalent.

This notion appears in some of the earliest work on the use of symmetry for simplifying analysis of dynamic systems. Some of these approaches can be found in the context of model checking for Petri nets.

Starke [54] was one of the first to take advantage of symmetries in reachability analysis for Petri nets. Petri nets are models of concurrent, discrete-time, dynamic systems that consist of place nodes, transition nodes, and directed arcs that connect place nodes to transition nodes. [51]. Noticing that reachability graphs, from which all the behavioural properties of a Petri net can be derived, were often prohibitively expensive to compute in full — their size corresponds to the number of states of the system modelled by the net — Starke developed a method for detecting symmetries in the net and thereby producing a *factorized* reachability graph. This factorized reachability graph, which represents symmetric transitions only once, can be much smaller and faster to compute than the original reachability graph.

Later, Emerson and Sistla [16] showed how to exploit such symmetry in the field of model-checking. They assume a system composed of many identical processes that has a global state transition graph $\mathcal{M}$. Given any group G contained in the set of automorphisms of $\mathcal{M}$, they define the graph $\bar{\mathcal{M}}$ to be the *quotient* structure of $\mathcal{M}$ with respect to G. Then, when some specification formula f is to be tested in the model $\mathcal{M}$, it is sufficient to test it in the potentiallly much smaller model $\bar{\mathcal{M}}$ instead.

This idea has shown up more recently in the work of Rintanen [53], who has considered equivalence at the level of transition sequences for use in SAT-based planners. As a pre-processing step, the problem designer defines a function E that partitions the domain states into classes, and automorphisms are found in the graph representing the transitions between all the states. A formula is generated to encode when two transition sequences are interchangeable, as well as another formula that prevents examining two transitions when they are known to be interchangeable. These formulae are added to the SAT formula for the planning or model checking problem. These formulae can sometimes be quite large, and the design function E will need to be specified by a designer for any particular application.

3.4 Foundation techniques

Our work relies heavily on some preceding techniques, which we will review in the next few sections.

3.4.1 Fast Forward and the FF heuristic

The Fast-Forward planning system (FF) developed by Hoffmann and Nebel is one of the best known and most successful planning systems for propositional domains. It is a forward-chaining, heuristic-based, state-space planner. Its success is due in part to its ingenious heuristic (the principle of which is originally due to Bonet and Geffner in their HSP system [5]). To make its heuristic estimate, FF *relaxes* the given task into a simpler, relaxed task by ignoring the delete lists of all rule operators. The heuristic value of a state, i.e., how far we estimate the state to be from the goal, is

just the number of steps taken to solve this relaxed task. Since the number of steps will be an underestimate of the true distance, this technique produces an admissible heuristic.

Underlying the computation of the FF heuristic is a Graphplan-style algorithm. The Graphplan algorithm was originally developed for traditional STRIPS domains and is very effective [3] in those domains. Graphplan finds the shortest straight-line plan by iteratively growing a forward-chaining structure called a *plan graph* and testing for the presence of goal conditions at each step.

Figure 3-2 shows an illustration of the Graphplan plan graph in action. The example is a blocks-world planning task, with initial state given at left, and goal condition at right. The first layer of the plan graph simply lists the set of propositions available in the initial state. Next, the algorithm determines what actions are applicable given the propositions in the previous layer, and keeps track of which actions compete with each other for propositions — these are called *mutual exclusivity*, or *mutex* for short, constraints. The next layer in the graph consists of all the propositions added by the actions in the last layer, in addition to all the propositions in the previous layer, propagated by implicit "maintenance" actions. If, in the latest layer of the graph, the necessary propositions exist to satisfy the goal condition, the Graphplan algorithm executes a backtracking search from the most recent layer to the earliest, for a mutex-free set of actions that produces the propositions necessary to achieve the goal. In Figure 3-2, the goal is to have `block3` on `block1`, `block1` on `block2`, and `block2` on the table. The propositions necessary to satisfy the goal are circled in red. In this case, the action sequence of picking up `block3` and then putting it down on `block1` is sufficient, since the remaining propositions were true in the initial plan layer and appear in the final layer via only maintenance actions.

3.4.2 MDP model minimization

The idea behind MDP model minimization is to group states together that exhibit the same response to action effects, thereby preserving the Markovian property of the system [11, 26, 12, 27].

Figure 3-2: An illustration of the Graphplan algorithm finding a solution for a block-stacking task given the initial state at left, and the goal condition at right. Maintenance actions are shown with dotted lines. For simplicity, the mutual-exclusivity constraints are not shown.

The idea of MDP minimization is based on techniques from FSA minimization [37]. In a deterministic FSA, states can be considered equivalent if they exhibit the same output and state transitions under all actions. A relation that groups pairs of states in this way is termed a *bisimulation*. FSA minimization algorithms depend on the notion of a SPLIT operation: given a partition P of the state space, two blocks of states B and C in the partition, and an action, produce a new partition that refines the block B into smaller groups of states that transition similarly into C.

Givan *et al.*[11, 26] consider model minimization as it applies to planning. They argue that it may be unnecessarily aggressive to distinguish between states whose behavior differs on action sequences not leading to the goal. They sketch how classical goal-regression planning can be cast as computing an approximate partial FSA minimization. Partial means that the partition may be coarser (i.e., group more states together) than the true minimal partition, and approximate means that the sets of states may overlap, rather than being a true partition. The sketch is as follows: in the first step, the SPLIT operation starts from the goal state, and regresses an action backward, producing a refinement that consists of blocks corresponding to the preconditions that enable the execution of the action. Unless care is taken, however, a naive regression algorithm may continue to produce splits without realizing that the generated blocks, described by boolean formulas, denote the same, or overlapping, sets of ground states. Unfortunately, enforcing mutual exclusivity requires testing a boolean formula for satisfiability, which is NP-complete. Givan et al. suggest an algorithm that modifies the original solution in two ways: first, the language that describes the minimized MDP is reduced (i.e., a reachability analysis throws out fluents deemed to be *irrelevant*); and second, a less optimal but easier to compute SPLIT operation (i.e., it splits at least as much as the optimal operation) is applied in the smaller space.

In the stochastic setting, it is necessary to consider the probability of a transition between states, not just the output behavior given a particular transition sequence. The notion of *stochastic bisimulation* is proposed as a criterion for state equivalence. [11, 27] Two states i and j are said to belong to a stochastic bisimulation

relation, E, if they share two properties. First, their immediate rewards must be equal both to each other and to that of the other states in their respective blocks; and second, given an action α and two states i' and j' in the relation, the probability of the transition under α from i to i''s block must equal that of the transition from j to j''s block. The stochastic bisimulation E induces a partition P on the group of states; in other words, bisimilar states get grouped into the same block.

Unfortunately, computing an exact stochastic bisimulation relation with an optimal SPLIT operation is NP-hard. One may ease the computational burden by considering a less than optimal SPLIT operation; there is, of course, a trade-off between the ease of computation and the amount of minimization that is achieved. Dean and Givan [27] show how the previous work of Boutilier and Dearden [7] can be cast as iteratively computing an approximate stochastic bisimulation partition.

Another way to relax the problem of MDP minimization is to consider states that share *approximately* the same transition behavior. Dean et al. [12] formalize this with their notion of approximate homogeneity. The main computational tool is an MDP in which transition probabilities and rewards are closed intervals, rather than scalar values. An approximately optimal policy can be found for such a *bounded-parameter MDP* (BMDP) using an extension of the usual value iteration algorithm for MDPs called *interval value iteration*. Thus, partition blocks are summarized with interval statistics, and the goal is to find a partition whose component states have reward and transition probabilities that differ by less than ϵ, called the ϵ-*approximate stochastic bisimulation property*. An initial partition is constructed based on immediate reward, and then is successively refined by searching for clusters of approximately-similarly behaving sub-blocks.

We will need results from the work in bounded-parameter MDPs when we transform a plan trajectory into an envelope MDP. While the above approaches involve propositional MDPs and do not also involve grouping actions, we note that the states in our approach are abstract states, which represent groups of underlying states. Thus, many of the results — in particular, the interval value iteration algorithm — are directly applicable to our setting.

Chapter 4

Formally defining equivalence

The complexity of planning is driven primarily by the length of the solution and the branching factor of the search. The solution length can sometimes be effectively reduced using hierarchical techniques. The branching factor can often be reduced, in effect, by an efficient heuristic. We will provide a novel method for reducing the branching factor by dynamically grouping the agent's actions into *state-dependent equivalence classes*, and only considering a single action from each class in the search. This method can dramatically reduce the size of the search space, while preserving correctness and completeness of the planning algorithm. It can be combined with heuristic functions and other methods for improving planning speed.

To be clear about the kinds of things we'll be working with, we provide as an example PDDL description of a blocks-world planning problem in Figure 4-1, just to have it handy. This is almost the same as the PPDDL description in Figure 1-2; however, in this chapter we will only consider deterministic effects.

4.1 Assumptions and definitions

Assumption 1 (Sufficiency of Object Properties)**.** *We assume a domain object's function is determined solely by its properties and relations to other objects, and not by its name.*

An important consequence of this assumption is that it will be necessary to support

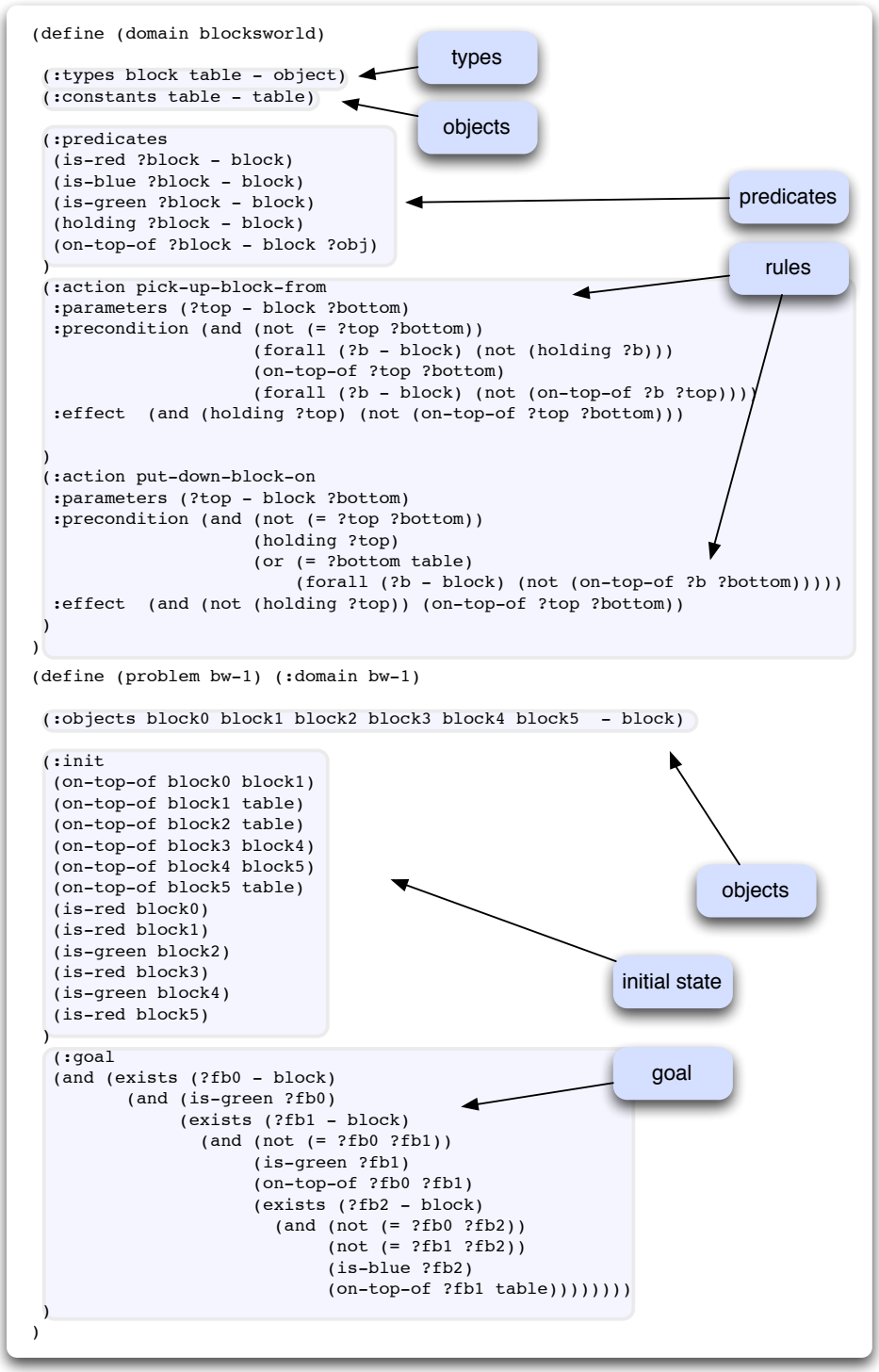


Figure 4-1: A complete PDDL specification of a planning problem.

fully quantified goal sentences, a considerable generalization to the propositional goals typically handled by planning systems. If we are in a setting in which a few objects' identities are in fact necessary, say by being named in the goal sentence, then we encode this information via supplementary properties. That is, we add a relation such as *is-block14(X)* that would only be true for block14. Obviously, if identity matters for a large number of objects, the approach presented here would not generate much improvement.

Intuitively, we mean to say that two objects are equivalent to each other if they are related in the same way to other objects that are, in turn, equivalent.

We will start by defining an equivalence relation on states. To do this, we will view the relational state description of a state s as a graph, called the *state relation graph*, and denoted $\mathcal{G}_s$. The nodes in the graph correspond to objects in the domain, and the edges correspond to binary relations between the objects. Relations with more than two arguments, e.g., *refuel(h1,level1,level2)*, can be represented making edges that "split" the relation, e.g., *refuel*$_1$*(h1,level1)* and *refuel*$_2$*(level1,level2)*. In addition, nodes and edges have a *label*, $\mathcal{L}$, which is a set of strings. The label for each node contains the object's type and the values of any other unary predicates in the domain; the label for each edge contains the relation's name. Two states are equivalent if there is a one-to-one mapping between the objects that preserves node and edge labels of the state relation graphs. That is:

State equivalence: *Two* states s_1 *and* s_2 *are* equivalent*, denoted* $s_1 \sim s_2$*, if there exists an isomorphism,* ϕ*, between the respective state relation graphs:* $\phi(\mathcal{G}_{s_1}) = \mathcal{G}_{s_2}$.

In Figure 4-2, we have an example of state equivalence. To determine if states s_1 and s_2 are equivalent, we construct their respective state relation graphs: $\mathcal{G}_{s_1}$ and $\mathcal{G}_{s_2}$. For each object in the state, we add one node to the graph. This node is labeled with the object's type and any unary properties. For example, note node 0 in s_1 is labeled with the type `Block` and the property *isgrey*(). Next, for each relation between objects in the state, we add one edge between the corresponding nodes in the graph. This edge is labeled with the name of the relation. Referring to the figure

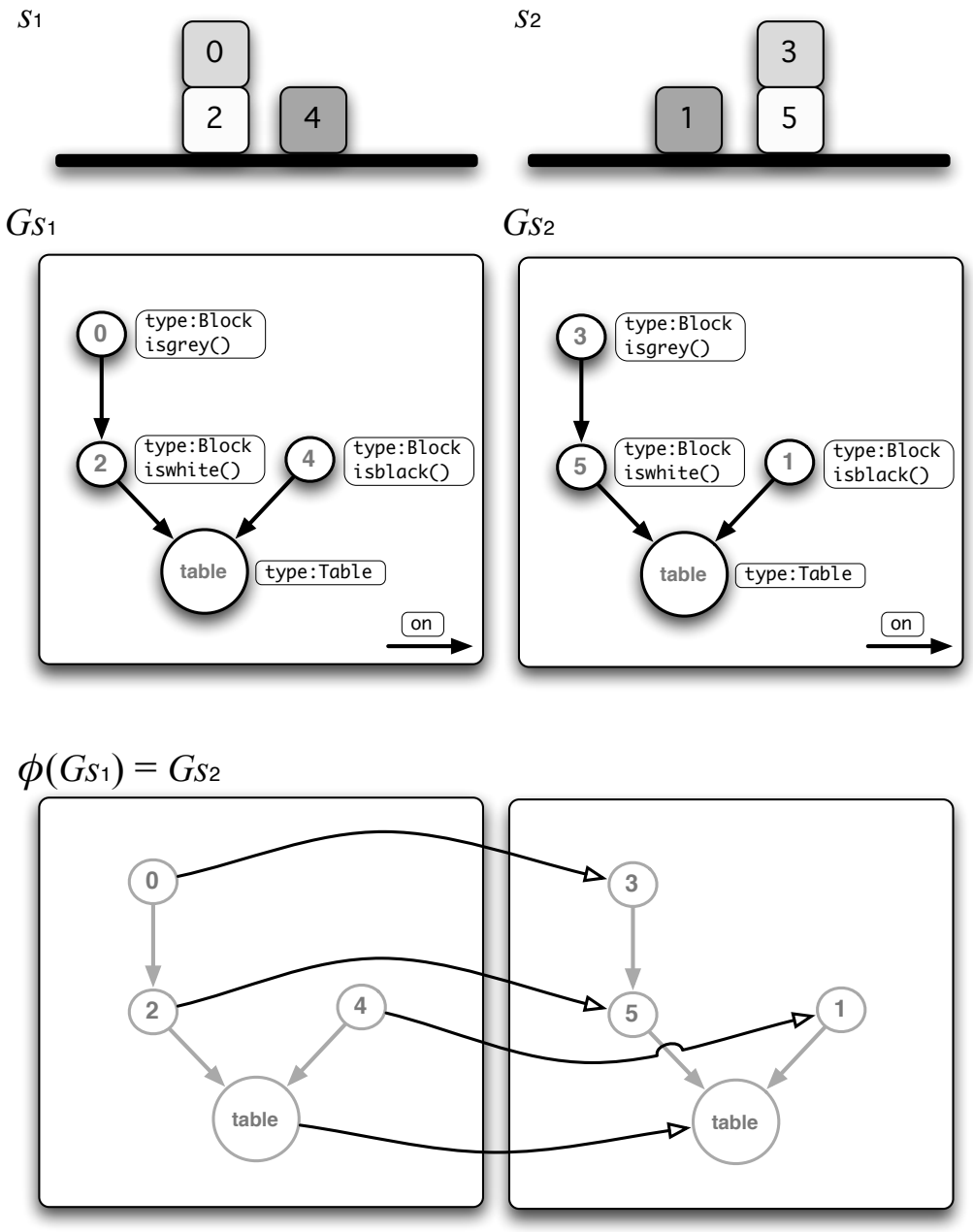


Figure 4-2: An example of determining equivalence between states s_1 and s_2. The first step is to construct the state relation graphs G_{s_1} and G_{s_2}. Nodes are labeled with their corresponding object's type and properties, and edges are labeled with the corresponding relation's name. Then, we look for a mapping, ϕ, between the two graphs.

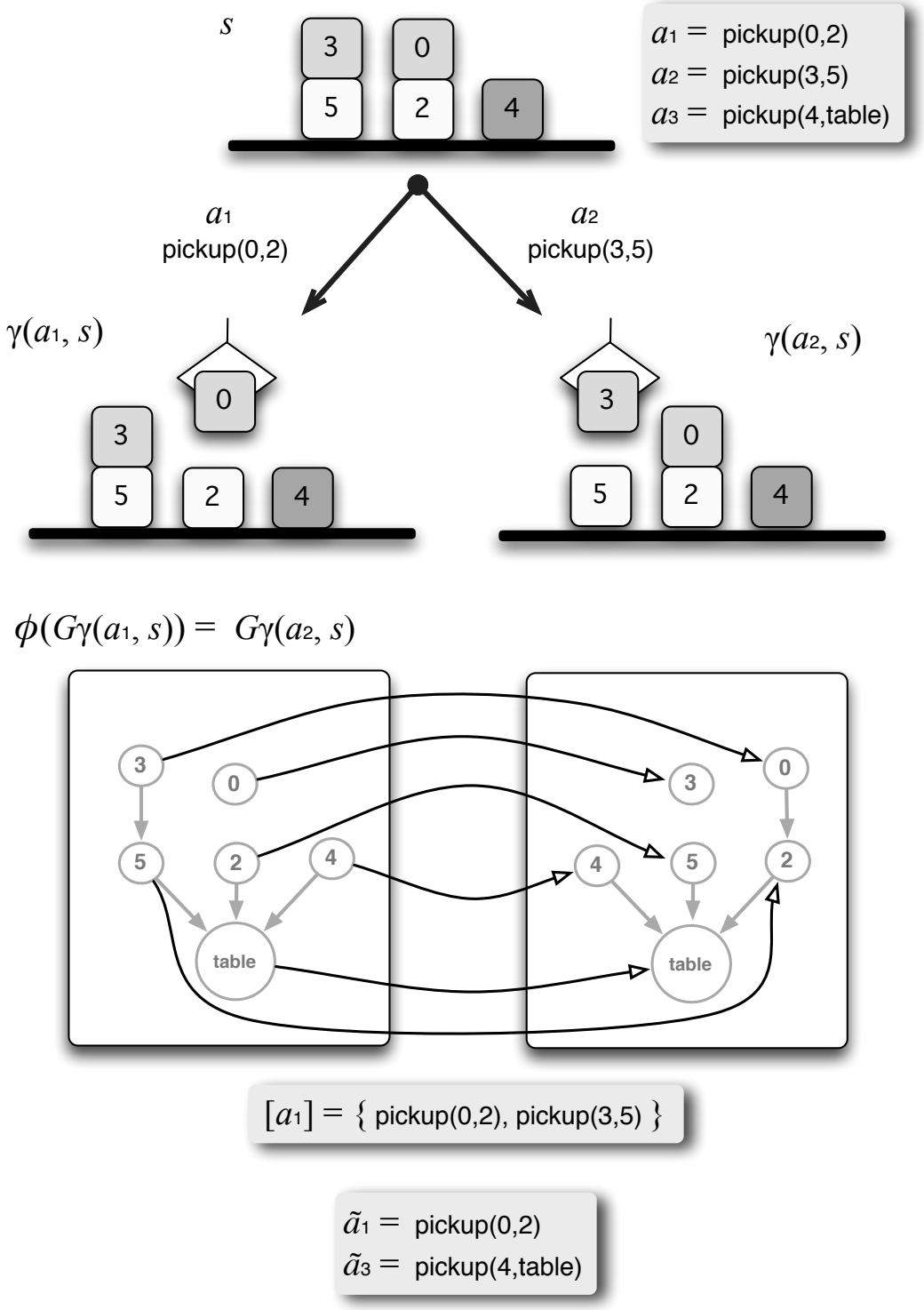


Figure 4-3: An example of determining whether two ground actions belong in the same equivalence class. Two ground actions are equivalent, by definition, if they result in equivalent successor states.

again, we see that, for example, the edge between nodes 0 and 2 is labeled `on`, since the *on* relation holds between blocks 0 and 2 in s_1. Finally, we look for a mapping, ϕ, between the two graphs that respects the given labeling.

We use the notation $\gamma(a, s)$ to refer to the state that results from taking action a in state s. If a is an action following the PDDL syntax, then we calculate $\gamma(a, s)$ by removing from s all the atoms in a's *delete list* and adding all the atoms in a's *add list*. We will sometimes refer to this calculation of $\gamma(a, s)$ as "propagating" s through the dynamics of a. More precisely, we write:

$$\gamma(a, s)) = \phi(s \cup add(a) \setminus del(a)),$$

Where $del(a)$ refers to set the atoms that are negated in the effect of a, and $add(a)$ refers to the set of atoms that are non-negated in the effect of a.

With respect to a given state s, then, we define two ground actions a_1 and a_2 to be equivalent if they produce equivalent successor states, $\gamma(a_1, s)$ and $\gamma(a_2, s)$:

Action equivalence: *Two* actions a_1 *and* a_2 *are* equivalent *in a state* s*, denoted* $a_1 \sim a_2$*, iff* $\gamma(a_1, s) \sim \gamma(a_2, s)$

An example is shown in Figure 4-3. We see how taking two instances of the *pickup* action in state s produce two different successor states that can be found to be equivalent to each other, using the procedure described above. In this case, we would put these two actions in to the same equivalence class. This process is repeated for all pairs of ground actions in s.

Just a word on notation : we will write $[o]$ to designate the equivalence class (which is a set) of an entity o. It will sometimes be convenient to use the shorthand notation $\tilde{o}$ to indicate that we are referring to o as a representative entity of its equivalence class.

Since our objective is to group *all* actions in a state into equivalence classes, this definition can be unwieldy to use directly in the calculation of equivalent actions: it requires propagation of the state through each action's dynamics in order to look for pairs of resulting states that are equivalent. We can, instead, *overload* the notion

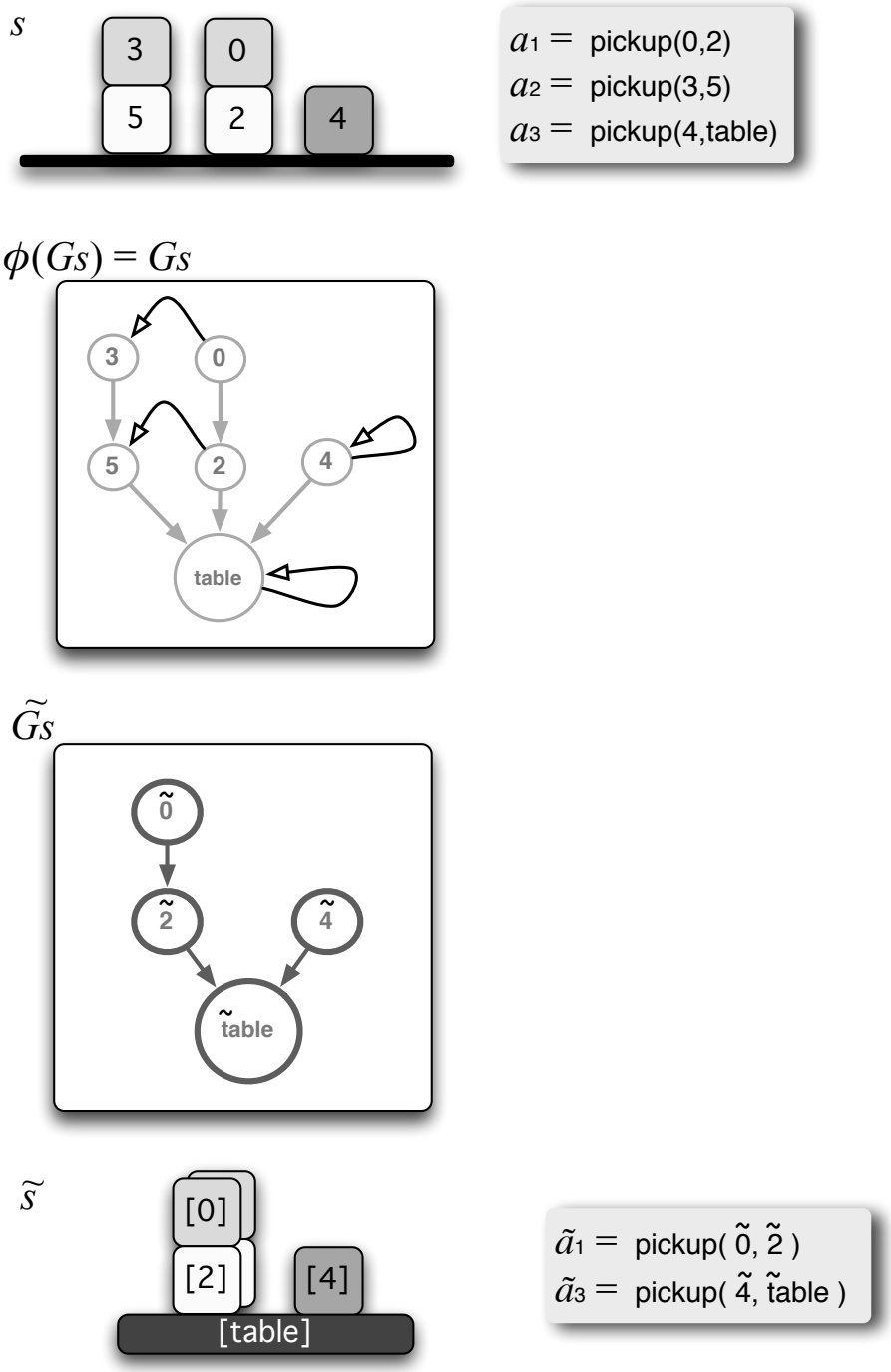


Figure 4-4: Action equivalence classes can also be found directly by computing equivalence classes among objects in the originating state. Each ground action applicable in the abstract state $[s]$ is a representative action for its equivalence class.

of isomorphism to apply to actions and develop a test on the starting state and actions directly, without propagation. In PPDDL and related formalisms, actions can be thought of as ground applications of predicates. Thus, each argument in a ground action will correspond to an object in the state, and, thus, to a node in the state relation graph. So, two actions applicable in a state s are provably equivalent if: (1) they are each ground instances of the same operator, and (2) there exists a mapping $\phi(s) = s$ that will map the arguments of one action to arguments of the other. In this case, since the isomorphism ϕ that we seek is a mapping between s and itself, it is called an *automorphism*. We compute action equivalence via the notion of action isomorphism, defined formally as follows:

Action isomorphism: *Two actions a_1 and a_2 are* isomorphic *in a state s, denoted $a_1 \sim_s a_2$, iff there exists an automorphism for s, $\phi(s) = s$, such that $\phi(a_1) = a_2$.*

This idea is illustrated in Figure 4-4: to compute the equivalence classes for the actions applicable in a state s, we first find the set Φ of automorphisms of the graph $\mathcal{G}_s$. This set of automorphisms induces a grouping into equivalence classes of the objects in s. An equivalence class containing an object consists of the set of objects to which it was mapped in one of the automorphisms $\phi \in \Phi$. This grouping on objects lets us construct a representative graph $\tilde{\mathcal{G}}_s$, with one node per representative object. When we compute the set of actions applicable in the corresponding representative, or *canonical*, state, $\tilde{s}$, we will have found exactly the set of representative actions for each equivalence class.

4.2 Consequences and main theorem

To show that the relations $\sim$ and $\sim_s$ defined in the definition of state equivalence and action isomorphism are in fact equivalence relations, we have to show that they are reflexive, symmetric, and transitive.

Lemma 1. *Relation-graph isomorphism defines an equivalence relation on states and actions.*

Proof. First, a state s produces a unique relation graph $\mathcal{G}_s$, and there always exists

the identity mapping from $\mathcal{G}_s$ to $\mathcal{G}_s$, so we conclude $s \sim s$. Next, if $s_1 \sim s_2$, then there exists ϕ such that $\phi(\mathcal{G}_{s_1}) = \mathcal{G}_{s_2}$. Since ϕ is bijective, it has an inverse, $\phi^{-1}(\mathcal{G}_{s_2}) = \mathcal{G}_{s_1}$, and so we conclude $s_2 \sim s_1$. Finally, if $s_1 \sim s_2$ and $s_2 \sim s_3$, then there exist ϕ_1 such that $\phi_1(\mathcal{G}_{s_1}) = \mathcal{G}_{s_2}$ and ϕ_2 such that $\phi_2(\mathcal{G}_{s_2}) = \mathcal{G}_{s_3}$. Thus $\phi_2(\phi_1(\mathcal{G}_{s_1})) = \mathcal{G}_{s_3}$, which implies $s_1 \sim s_3$. The argument for actions is analogous. □

Since the relation $\sim$ is an equivalence relation, we denote the equivalence class containing item x as $[x]$.

Next, we see that if a logical sentence is satisfied in a state s, then it can be satisfied in any state $\tilde{s} \in [s]$. We assume that a ground state is a fully ground list of facts (which we can treat as a conjunction of ground atoms). When we say that a state entails a sentence, we are speaking purely of syntactic entailment.

Lemma 2. *If a sentence ρ is syntactically entailed by a state s, then it is entailed by any $\tilde{s} \in [s]$*

Proof. Let $\phi(s) = \tilde{s}$. If sentence ρ is entailed by s, then there is some substitution ψ for the variables in ρ such that $\psi(\rho)$ is a subset of s. In other words, $s \models \rho$ if and only if $\exists\psi.\psi(\rho) \subseteq s$. Assume ρ is entailed by s, and let $\psi(\rho) \subseteq s$. We know by equivalence of s and $\tilde{s}$ that there exists a mapping ϕ^{-1} such that:

$$\begin{aligned}
\phi^{-1}(\tilde{s}) &= s. \\
\text{So, } \psi(\rho) &\subseteq \phi^{-1}(\tilde{s}), \\
\text{and, } (\phi^{-1})^{-1}(\psi(\rho)) &\subseteq \tilde{s}. \\
\text{Let } \psi' &= \phi \circ \psi. \\
\text{Then, } \psi'(\rho) &\subseteq \tilde{s}, \text{ and, thus} \\
\tilde{s} &\vdash \rho.
\end{aligned}$$

□

What we are saying in the above proof is this: if we have a substitution (ψ) that makes a sentence (ρ) true in a state (s), then, we can make that sentence true in a

second state ($\tilde{s}$) by composing the mapping between the two states (Ψ) along with our original substitution (ψ) to make a new, satisfying, substitution (ψ').

As an example, consider the states s_1 and s_2 (in Figure 4-2) and a sentence, $\rho : on(A, B), on(B, D)$. Applying the substitution, $\psi = \{A/\texttt{0}, B/\texttt{2}, D/\texttt{table}\}$ to ρ yields the ground sentence

$$\texttt{on}(\texttt{0}, \texttt{2}), \texttt{on}(\texttt{2}, \texttt{table}),$$

which is a subset of the complete state in s_1:

$$\texttt{on}(\texttt{0}, \texttt{2}), \texttt{on}(\texttt{2}, \texttt{table}), \texttt{on}(\texttt{4}, \texttt{table}), \texttt{is} - \texttt{table}(\texttt{table}).$$

Now, previously, we found that there exists a ϕ such that $\phi(s_1) = s_2$, meaning that, each object v in s_1 corresponds uniquely to $\phi(v)$ in s_2:

v	$\phi(v)$
0	1
2	3
4	5
table	table

To get the substitution that makes ρ true in s_2, we compose ψ with ϕ:

$$\{\phi(\texttt{0})/A, \phi(\texttt{2})/B, \phi(\texttt{table})/D\},$$

which yields the substitution

$$\{\texttt{1}/A, \texttt{3}/B, \texttt{table}/D\}$$

and, thus, the ground sentence

$$\texttt{on}(\texttt{1}, \texttt{3}), \texttt{on}(\texttt{3}, \texttt{table}),$$

which is a subset of the complete state in s_2:

$$\texttt{on}(1,3), \texttt{on}(3,\texttt{table}), \texttt{on}(5,\texttt{table}), \texttt{is}-\texttt{table}(\texttt{table}).$$

The next lemma establishes the equivalence of the states produced by taking isomorphic ground actions in equivalent states.

Lemma 3. *Let s_1 and s_2 be equivalent states. If two actions $a_1 \in z|_{s_1}$ and $a_2 \in z|_{s_2}$ are isomorphic according to the definition of isomorphic actions, then the successor states $\gamma_i(a_1, s_1)$ and $\gamma_i(a_2, s_2)$ determined by their respective outcomes are equivalent.*

Proof. By definition, for a given outcome i of z, $\gamma_i(a_1, s_1) = s_1 \cup add_i(a_1) \setminus del_i(a_1)$, so:

$$\begin{aligned}
\phi(\gamma_i(a_1, s_1)) &= \phi(s_1 \cup add_i(a_1) \setminus del_i(a_1)) \\
&= \phi(s_1) \cup \phi(add_i(a_1)) \setminus \phi(del_i(a_1)) \\
&= s_2 \cup add_i(a_2) \setminus del_i(a_2) \\
&= \gamma_i(a_2, s_2) \\
\text{thus, } \gamma_i(a_1, s_1) &\sim \gamma_i(a_2, s_2)
\end{aligned}$$

□

Now we almost have all the pieces to state the main theorem. We know that equivalent schema applications, in equivalent current states, produce equivalent successor states. Now, we must show that a sequence of schema applications can be replaced by an equivalent sequence to produce equivalent ending states.

Definition 1 (Equivalent Planning Procedures)**.** *Let P be a planning procedure such at at each state s, P selects an action a. Consider a planning procedure P' such that at each state $\tilde{s} \sim s$, P' chooses an action $\tilde{a} \sim a$. Then P and P' are defined to be* equivalent planning procedures.

Theorem 1. *Let P be a complete planning procedure.*[1]*. Any planning procedure P'* equivalent *to P is also a complete planning procedure. That is,*

$$\forall \tilde{a_i} \in [a_i], \gamma(a_1, \ldots, a_n, s_0) \rightarrow g \Rightarrow \gamma(\tilde{a_1}, \ldots, \tilde{a_n}, s_0) \rightarrow g \ .$$

Proof. We prove the theorem by induction. First, consider the initial step. If a_1 in s_0 is equivalent to $\tilde{a_1}$ in s_0, then $\gamma(a_1, s_0) \sim \gamma(\tilde{a_1}, s_0)$ (by Lemma 3). Next, we need to show that if a_{i+1} in $\gamma(a_1, \ldots, a_i, s_0)$ is equivalent to $\tilde{a}_{i+1}$ in $\gamma(\tilde{a_1}, \ldots, \tilde{a_i}, s_0)$, then $\gamma(a_1, \ldots, a_{i+1}, s_0) \sim \gamma(\tilde{a_1}, \ldots, \tilde{a}_{i+1}, s_0)$. Again, Lemma 3 guarantees that

$$\begin{aligned} \gamma(a_{i+1}, \gamma(a_1, \ldots, a_i, s_0)) &\sim \gamma(\tilde{a}_{i+1}, \gamma(\tilde{a_1}, \ldots, \tilde{a_i}, s_0)), \text{thus,} \\ \gamma(a_1, \ldots, a_{i+1}, s_0)) &\sim \gamma(\tilde{a_1}, \ldots, \tilde{a}_{i+1}, s_0)). \\ \text{Hence, } \gamma(a_1, \ldots, a_n, s_0) &\sim \gamma(\tilde{a_1}, \ldots, \tilde{a_n}, s_0), \end{aligned}$$

and by Lemma 2, $\gamma(\tilde{a_1}, \ldots, \tilde{a_n}, s_0) \rightarrow g$. □

Thus, any plan that existed before in the full action space will have an equivalent version in the new, partitioned action space.

Planning in the reduced action space consisting of representatives from each equivalence class preserves completeness. It does, however, have an effect on plan parallelism. Since we are limited to only one action of each class on each step, a planning procedure that might have used two instances of the same class in parallel would have to serialize them.

4.3 Example of computing equivalence classes

To summarize the process, we consider an example. In Figure 4-5, an example problem instance contains 7 blocks, colored red and blue. Given the state s in the figure, our

[1] *A* complete *planning procedure is one which is guaranteed to find a path to the goal if one exists.*

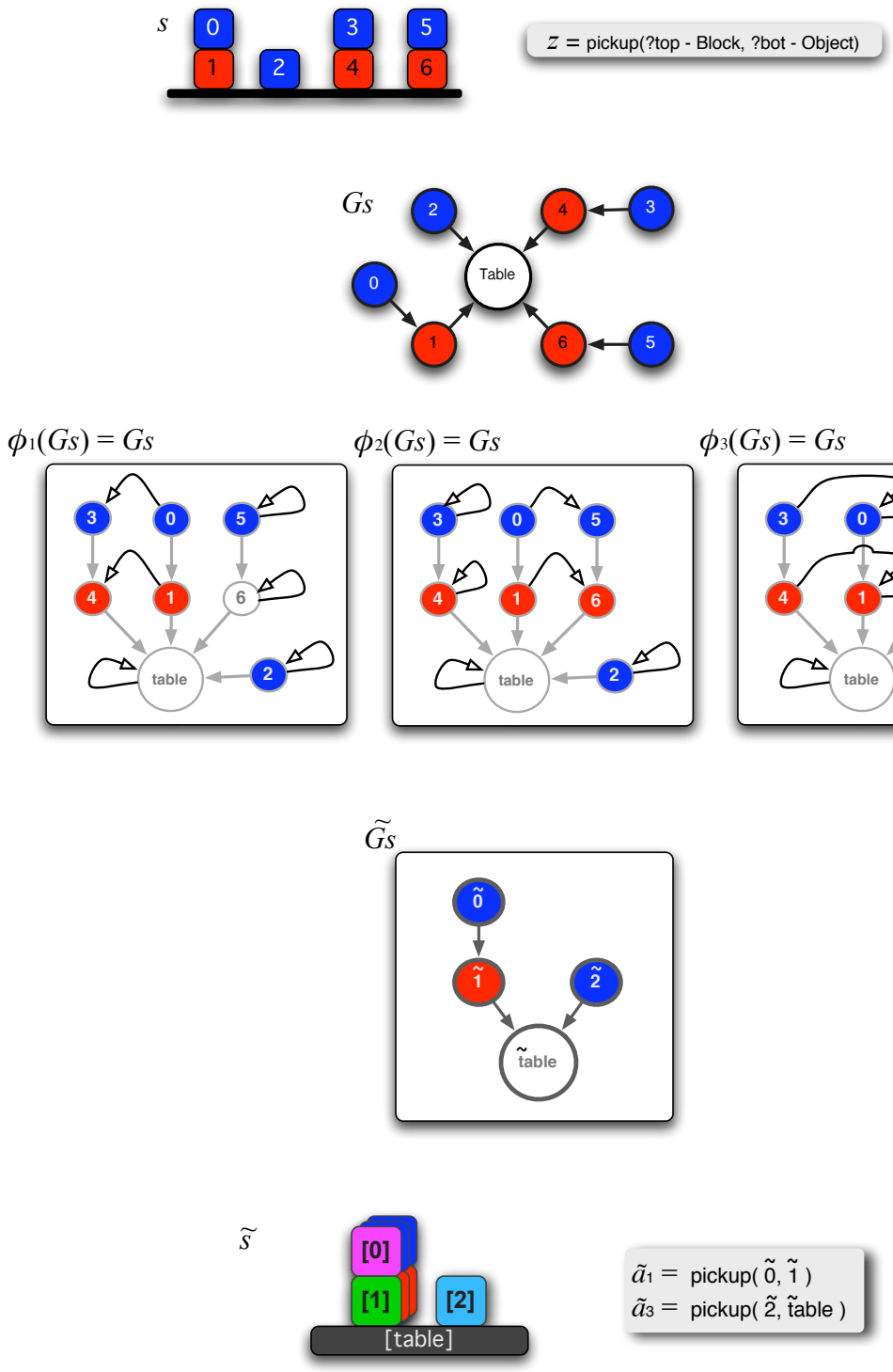


Figure 4-5: The steps involved in computing action equivalence classes in a 7-block domain. We start with the state s and the pickup operator z. The state relation graph G_s yields the set automorphisms $\Phi = \{\phi_1, \phi_2, \phi_3\}$. Grouping the objects together according to the mappings in Φ, produces the *canonical* state relation graph $\tilde{G}_s$, and the *canonical* state $\tilde{s}$. The set of action equivalence classes is then represented by the set of actions applicable in $\tilde{s}$, in which different colors are used to denote the object equivalent classes.

task is to compute which actions, if any, fall into the same equivalence class. We explain the procedure in detail below, using the figure for reference.

1. *Compute the set of automorphisms, Φ for the state relation graph G_s of s.* In Figure 4-5, we have drawn the state relation graph for the state s. There exist three unique automorphisms of G_s: $\Phi = \{\phi_1, \phi_2, \phi_3\}$.

2. *Compute the equivalence classes of objects.* The set Φ induces an equivalence partition among the objects. The equivalence class for each object $o \in s$ is

$$[o] = \{o_i | \exists \phi_i . \phi_i(o) = o_i\}.$$

   If we let $\tilde{o}$ stand for the representative object of this equivalence class, we can construct the *canonical* graph $\tilde{G}_s$, which consists of one node per representative object. The *canonical* state $\tilde{s}$ is constructed analogously: each object in s is a representative of each class, and each relation between canonical objects represents the relationship between corresponding objects of each class.

3. *The set of actions applicable in the canonical state constitutes the set equivalence actions.* In this case, there are two classes of actions: one consisting of actions similar to a_1, which pick up one block from another block; and, one consisting of actions similar to a_2, which pick up one block from the table.

4. *Return a reduced action set consisting of one representative from each equivalence class.* We return the set $\{\tilde{a_1}, \tilde{a_3}\}$.

Chapter 5

Equivalence-based planning

Having formally defined equivalences of actions and objects, we now discuss how they can be used to speed up planning. Recall that since we have formulated our MDP problem relationally, our approach will to take advantage of techniques from classical planning to produce a straight-line plan as as efficiently as possible and then use the state sequence induced by this plan to seed our envelope MDP.

We will first look briefly at a planning approach that does not work well but has instructive shortcomings. Then, we will look at a more successful approach, one that uses the action equivalence classes described above in the context of a heuristic-based forward-search planner. We will see that the action equivalence classes are not only useful in the main forward search, but also have the potential to significantly improve the heuristic evaluation. In our experience, computing the heuristic can often be a considerable portion of the overall computational effort; thus, any gains here are important.

5.1 First approach with TGraphplan

Since transitions in an MDP setting are stochastic, our first approach was to look for classical planning approaches that can handle stochastic actions. TGraphplan is one simple, established technique. It is based on the well-known Graphplan algorithm, which finds the shortest straight-line plan by iteratively growing a forward-chaining

structure called a *plangraph* and testing for the presence of goal conditions at each step. Blum and Langford [4] describe the probabilistic extension, TGraphplan, (TGP) which works by returning a plan's probability of success rather than a just a boolean flag. TGP can find straight-line plans fairly quickly from start to goal that satisfy a minimum probability. Given TGP's success in probabilistic STRIPS domains, our initial idea was to use the trajectory found by TGP to populate our initial envelope.

Of course, our relational MDP describes a large underlying MDP. TGP and other Graphplan descendants work by grounding out the rules and chaining them forward to construct the plangraph. Large numbers of actions cause severe problems for Graphplan-based planners [49] since the branching factor quickly chokes the forward-chaining plangraph construction.

In order to mitigate the potentially large branching factor, we would like to gain leverage from any equivalence classes among actions that we might be able to identify.

When we apply this definition into the TGraphplan setting, however, the following issue arises: in each layer of the plan graph, there is no notion of "current state." In the Graphplan algorithm, the first level in the graph contains the propositions corresponding to the facts in the initial state. Each level beyond the first contains two layers: one layer for all the actions that could possibly be enabled based on the propositions on the previous level, and a layer for all of the possible effects of those actions. Thus, each level of the plan graph simply contains a list of all propositions that could possibly be true. The only information at our disposal is that of which propositions are mutually exclusive from one another.

In order to partition actions into equivalence classes, we adopt the following criterion. We define the *extended state* of an action to be all those propositions in the current layer that are not mutually exclusive with each other nor with any of the action's preconditions. Thus, we group two actions together if the ground objects in each argument list are isomorphic to each other with respect to each action's *extended state*.

Computing equivalence based on extended states will create a finer set of equivalence classes as compared to ground states: the set of propositions that could be

possibly true is greater than or equal to the set that will actually become true. Thus, the equivalence classes produced by this criterion will be at least as fine (and probably more fine-grained) as those that exist in any actual state reachable at that layer.

While this is a reasonable first step, it quickly becomes clear that even in moderately-sized domains, the extended-state-based approach isn't aggressive enough and that much more leverage is needed. Thus, we move on to formulate the planning problem as a state-based search.

5.2 State-based approach

In a state-based search, we start with a ground state and apply the operators in our domain to explore resulting states. We look for some sequence of operations that will produce a ground state that satisfies the goal. Since we manipulate the states directly, this means that we can use our definition of equivalence classes on states to partition the action space as compactly as possible. This also means that we need to define some kind of heuristic to guide our search. Recall that our state-based search, to keep things as simple as possible, will take place in deterministic approximation of the original model: the planner will return a plan by considering only the *most-likely* outcome for each action. We'll see later how this plan serves as a starting point for the envelope-expansion phase.

For now, will divide the discussion of the state-based approach into two parts. First, we will discuss the "outer loop": that is, the main forward-chaining search for a plan. Then, we will discuss the "inner loop": how to compute a heuristic efficiently.

5.2.1 Outer loop: forward search

A typical forward-search planner has the basic structure:

Input: Initial state s_0, goal condition g, set of rules Z

Output: Sequence of actions from s_0 to a goal state

```
1 Initialize agenda with s_0
2 while agenda is not empty do
3     Select and remove a state s from the agenda
4     if s satisfies goal condition g then
5         return path from root of search tree to s
6     else
7         Find set A of ground actions applicable in s
8         foreach a ∈ A do
9             Add the successor of s under a to the agenda
```

Algorithm 1: Basic forward-search algorithm.

Our approach will make some modifications to this basic outline. First, we include a step before Line 1 that takes the initial ground state and reformulates it as an *canonical* state, as introduced in the previous chapter. This means that, rather than containing a list of relations among ground objects, the abstract state only contains relations among *canonical* objects: those objects that are representatives of an equivalence class.

The steps at Lines 7 and 9 also undergo a change. Instead of generating successors from each ground action, we find the set $\mathcal{A}$ of *equivalence classes of actions* applicable in s. Only a single successor state will be generated for each equivalence class of actions.

Reformulating the initial state

We were introduced to this idea in the previous chapter. We want to construct a *canonical* observation, or state, that captures the relationships between equivalence classes of objects.

From here onward, we will often use interchangeably the terms *node* and *object*, and also *state* and *state relation graph*, since there is a one-to-one correspondence

between objects and nodes, and states and graphs.

This means finding, first, the set Φ of automorphisms in the state relation graph $\mathcal{G}_{s_0}$ of s_0. We will compute the set of object equivalence classes of s, which starts out empty, as we go. We proceed as follows: for each node $n \in \mathcal{G}_{s_0}$ and equivalence class C of s, determine if there exists an isomorphism ϕ that maps n and the representative node of the class C. If so, then we know that n belongs in C. Otherwise, we create a new class C' and add n as first member. Which node gets chosen as a representative node for a class is arbitrary. In our implementation, we just use the first node that was added to the class. Pseudo code for this procedure is shown in Algorithm 2, below:

```
Input: Canonical state s̃
/* initialize classes of s̃ with an empty list */
s̃.objectEqClasses = {}
foreach CanonicalObject o ∈ s̃ do
    foreach EquivalenceClass e ∈ s̃.objectEqClasses do
        õ = representative element of e
        /* in the context of state s̃, determine if there is an
           isomorphism φ that maps o to õ */
        if isomorphic( o, õ ) then
            foundClass = True
            break
    /* if none of the object equivalence classes we have so far
       contain o, create a new one */
    if not foundClass then
        newClass = { }
        newClass.add( o )
        s̃.objectEqClasses.add( newClass )
```

Algorithm 2: The updateClasses() function: pseudo code for computing the equivalence classes of a state s.

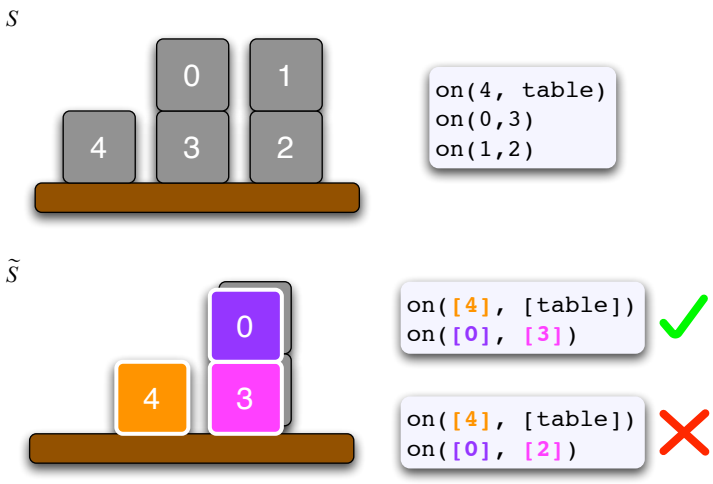


Figure 5-1: When we choose representatives from each equivalence class, we must be careful to conserve the relationships that the underlying objects have in the underlying world.

Now that we know which objects belong to which equivalence class, we must add to the state $\tilde{s}$ the *canonical propositions* describing the relationship between the classes. But here we must stop and think carefully: if a relation exists between two objects of different classes, we cannot simply create a proposition between the representative objects of each class. Consider Figure 5-1, in which we show a ground state s and its corresponding canonical state, $\tilde{s}$. In this case, we know that blocks 0 and 1 belong to the same class, as do blocks 3 and 2. But only the proposition `on-top-of([block0],[block3])`, is consistent with the interpretation associated with s, but not `on-top-of([block0],[block2])`. It is important to keep this straight because, in the next step, we will be looking for applicable actions in $\tilde{s}$. The action `pick-up(block0,block3)` is a legal action, and `pick-up(block0,block2)` is not.

To construct a consistent set of canonical propositions, we follow the procedure in Algorithm 3, below. The basic idea is to pick a representative object o for the first object equivalence class arbitrarily. Then, the remaining object equivalence classes will be represented by those objects related to o in the underlying ground state. We do this until we run out of objects related to o. If there are object equivalence classes remaining to be represented in the canonical state, we start the process again with a new arbitrarily chosen o.

Input: Canonical state $\tilde{s}$

`/* Keep track of which classes have been computed so far in` $\tilde{s}$ `*/`

seenEqClasses = {}

foreach *EquivalenceClass* $e \in \tilde{s}.objectEqClasses$ **do**

  $\tilde{o}$ = representative element of e

  addRelatedObjects($\tilde{s}$, $\tilde{o}$, seenEqClasses)

Define *addRelatedObjects()*

Input: Canonical state $\tilde{s}$, Canonical object $\tilde{o}$, List seenEqClasses

`/* recursively collect all canonical objects related to` $\tilde{o}$`, and each relation we collect is added to the canonical state,` $\tilde{s}$ `*/`

foreach *Object o in relatedTo(* $\tilde{o}$ *)* **do**

  Predicate p = nameOfRelationBetween($\tilde{o}$, o)

  $\tilde{s}$.addProposition(p, $\tilde{o}$, o)

  seenClasses.add(equivalenceClassOf(o))

  `/* recursive step */`

  addRelatedObjects($\tilde{s}$, o, seenEqClasses)

Algorithm 3: Pseudo code for computing the propositions for a canonical state $\tilde{s}$ once the object equivalence classes for $\tilde{s}$ have been computed.

At this point, we have constructed a canonical state consisting only of the relationships between canonical objects. This, then, is our initial observation, and we are ready to find out which actions apply in this state.

Find the set of action equivalence classes

Given an abstract state $\tilde{s}$, the equivalence classes of actions can be induced directly. For each operator $\mathcal{Z}$, we determine which actions are applicable in $\tilde{s}$. This is done analogously to determining which ground actions are applicable in a ground state, s, but here is the difference: in the ground case, the universe of discourse consisted of the set of domain objects; in the canonical case, the universe of discourse has been reduced to the set of representative objects of each object equivalence class. So, we

need only consider ground actions whose preconditions are true given the canonical propositions in $\tilde{s}$. To extend the terminology we have been using, we call the ground actions in the canonical case *canonical ground actions*. For example, in Figure 5-1, yields two possible canonical ground actions: `pick-up-block-from([4],[table])` and `pick-up-block-from([0],[block3])`.

We're not quite done, however. What if we had a variation of the basic blocks world in which we have an operator *move(top, bottom, target)* that moves a block *top* from *bottom* onto *target*, in one step? As we can see in Figure 5-2 (left side), [1] moving block 0 to block 1 is a legal operation in state s. How could we permit the same operation in $\tilde{s}$, since blocks 0 and 1 are in the same equivalence class and are represented by a single canonical literal [0] ?

We introduce a *one-of()* function on canonical literals. That is to say, when building up a candidate binding list, ψ, we bind the result of the *one-of()* function to the corresponding variable. By default, *one-of*(`[0]`) simply returns the representative object for the class `[0]`. However, its power comes in when we come across an inequality clause in the sentence we are trying to unify. An inequality clause, such as `(not (= ?a ?b))` constrains the legal unifications of the clause to those that bind a and b to distinct domain objects.[2] If such an inequality constraint is encountered, the *one-of()* function, which keeps track of which elements of the class it has doled out to the binding list so far, looks to see if it can provide another distinct element of the class. If not, the binding fails.

Let's see how this works in Figure 5-2 (right side). By default, any time the function *one-of*(`[0]`) is bound in the list ψ, it will return `0` . However, in order to satisfy the constraint `(not (=` *one-of*(`[0]`) *one-of*(`[0]`)`))`, *one-of*(`[0]`) must be able to return a domain object distinct from block `0`. In this case, block `1` is available and will be bound to the variable `?target`.

[1] Note that the *move* operator as shown in Figure 5-2 is a simplification so as not to encumber the discussion at hand with excessive detail. It will not, as written, behave as expected; but, we can just make a note of that and carry on.

[2] Otherwise, no such constraint is assumed, and it would be perfectly legal to try to move block 0 onto itself.

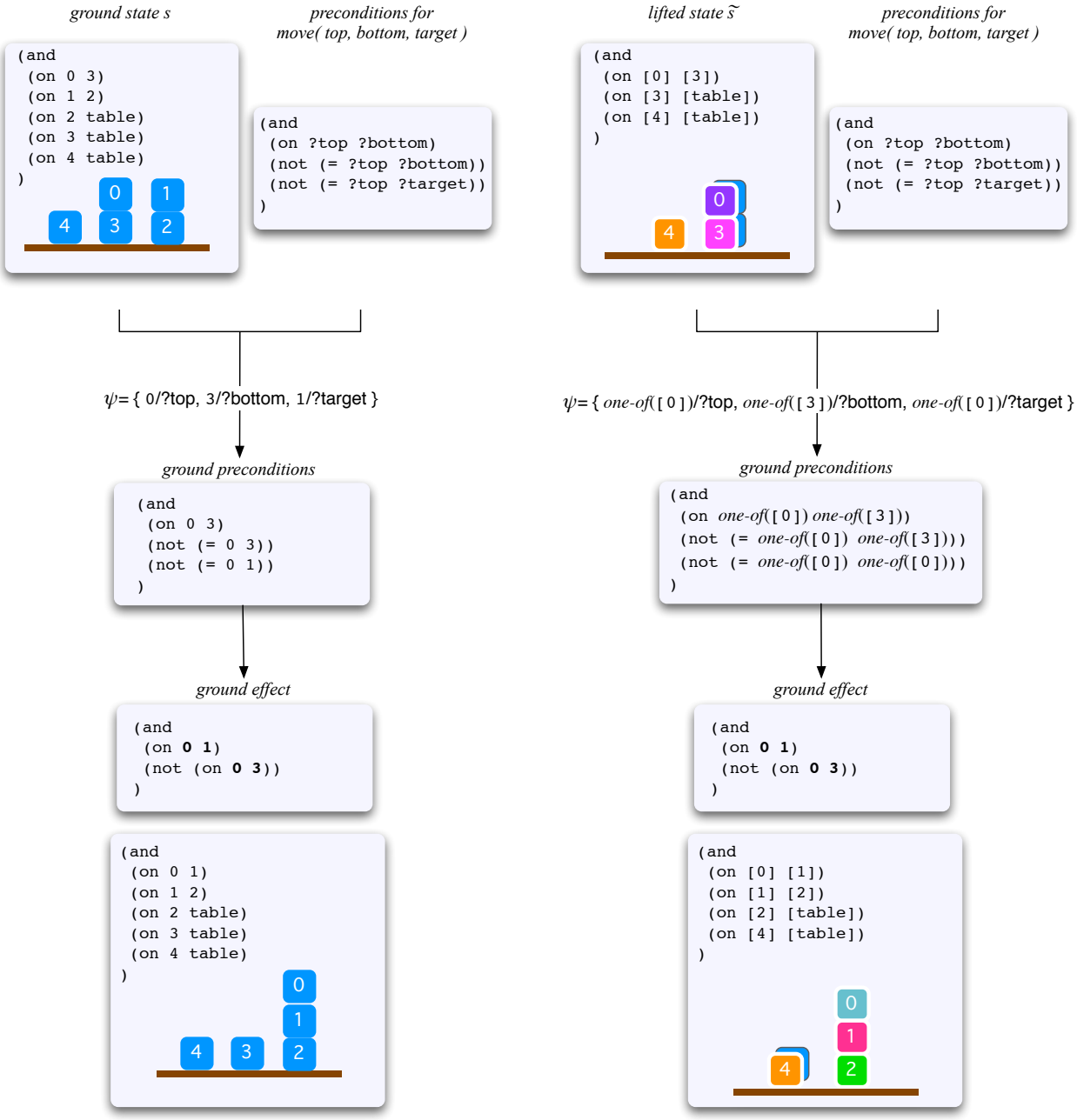


Figure 5-2: On the left side, we see how the *move* operator works in a ground state. We introduce a *one-of()* function on canonical literals to ensure analogous behavior in a canonical state, on the right side.

Find the successor states

Given that we started from an abstract state $\tilde{s}$ and applied an action to it, we would like to avoid repeating this work when calculating the classes in the successor state. In order to conserve as much work as possible, we notice that each action only manipulates a subset of the objects, and, thus, that only a subset of the equivalence classes will change in $\tilde{s}'$.

Let's look at the left side of Figure 5-2 again. The effect of the *move* operator changes the *on* relationship between blocks `0` and `1`, and blocks `0` and `3`. We want only recalculate the object equivalence classes necessary to reflect the change. This can be done simply by initializing $\tilde{s}'$ to be a copy of $\tilde{s}$ and removing the manipulated objects, blocks `0`, `1`, and `3` from their equivalence classes before copying the classes over to $\tilde{s}'$. To be completely correct, we must also remove from their equivalence classes any objects *related to* the manipulated objects: this is because an object is only a member of an equivalence class by virtue of sharing similar relationships with the other objects in its class. Once its relationships change, it may belong to a different class. In our example of Figure 5-2, this results in blocks `0`, `1`, `2`, and `3` being removed from their equivalence classes.

Once we have pulled out all the manipulated, and indirectly manipulated, objects from their equivalence classes, we simply run the procedure of Algorithm 2 on $\tilde{s}'$. This updates the classes for $\tilde{s}'$. Then, we can follow with the procedures of Figure 3 to calculate the canonical propositions for $\tilde{s}'$. The result for our example is seen at the bottom of the left side of Figure 5-2.

Having computed each successor state in this way, we will need to evaluate each state with some heuristic (which we'll describe later in this section) and insert them accordingly into the search agenda.

5.2.2 Inner loop: heuristic computation

The most expensive part of the algorithm is in the heuristic evaluation of a candidate action. This is because we adapt the well-known FF heuristic in order to make it

admissible. The Fast-Forward (FF) algorithm is well-established and known to produce an admissible heuristic efficiently [36]. FF works by using a Graphplan-style structure to solve a *relaxed* planning problem. A relaxed plan is one that ignores mutual-exclusion interactions between propositions in the graph. It can quickly estimate a lower bound on the number of steps necessary to achieve a goal; this estimate is inadmissible, however, since it is not necessarily the shortest relaxed plan. We can turn this into an admissible heuristic by searching in the plan graph for the shortest relaxed plan. Taking the time to compute an admissible heuristic guarantees an optimal, shortest-length plan. We know from our previous experience with TGraphplan, however, that managing a large branching factor will be crucial to getting a plangraph-based approach off the ground.

We have developed a technique for solving relaxed planning problems that takes advantage of the computation of equivalence classes occurring in the main search loop. The result is a heuristic computation, based on the FF idea, that is considerably more efficient than simply using ground actions.

Let's work with our running example. Imagine we are given the state in Figure 5-1, and that we have to evaluate how far this state is from our goal of three blocks in a stack. That is, we would like to underestimate how many steps it would take, from this state, to reach the goal.

A straightforward calculation of the heuristic would proceed as in Figure 5-3. The initial proposition layer in the plan graph is built up with the facts from the initial state. The next layer, an action layer, contains all of the possible actions that might be applicable given the previous layer, ignoring any mutual exclusion effects. [3] The resulting proposition layer includes all of the possible effects of the preceding actions. We test our goal condition over the union of the layers, since this includes

[3]We take advantage of one trick here to reduce branching factor: because we know that the initial layer represents the true state of the world, we compute the actions in the first action layer in accordance with any pre-condition requirements. For example, the *pick(2)* action, which is not legally executable in the ground state, is not added in this layer since we know block 2 is not clear. For subsequent layers, however, actions are added "optimistically": as long as the positive atoms in the precondition exist in the previous layer, the action can be added without regard to negated or interacting precondition atoms.

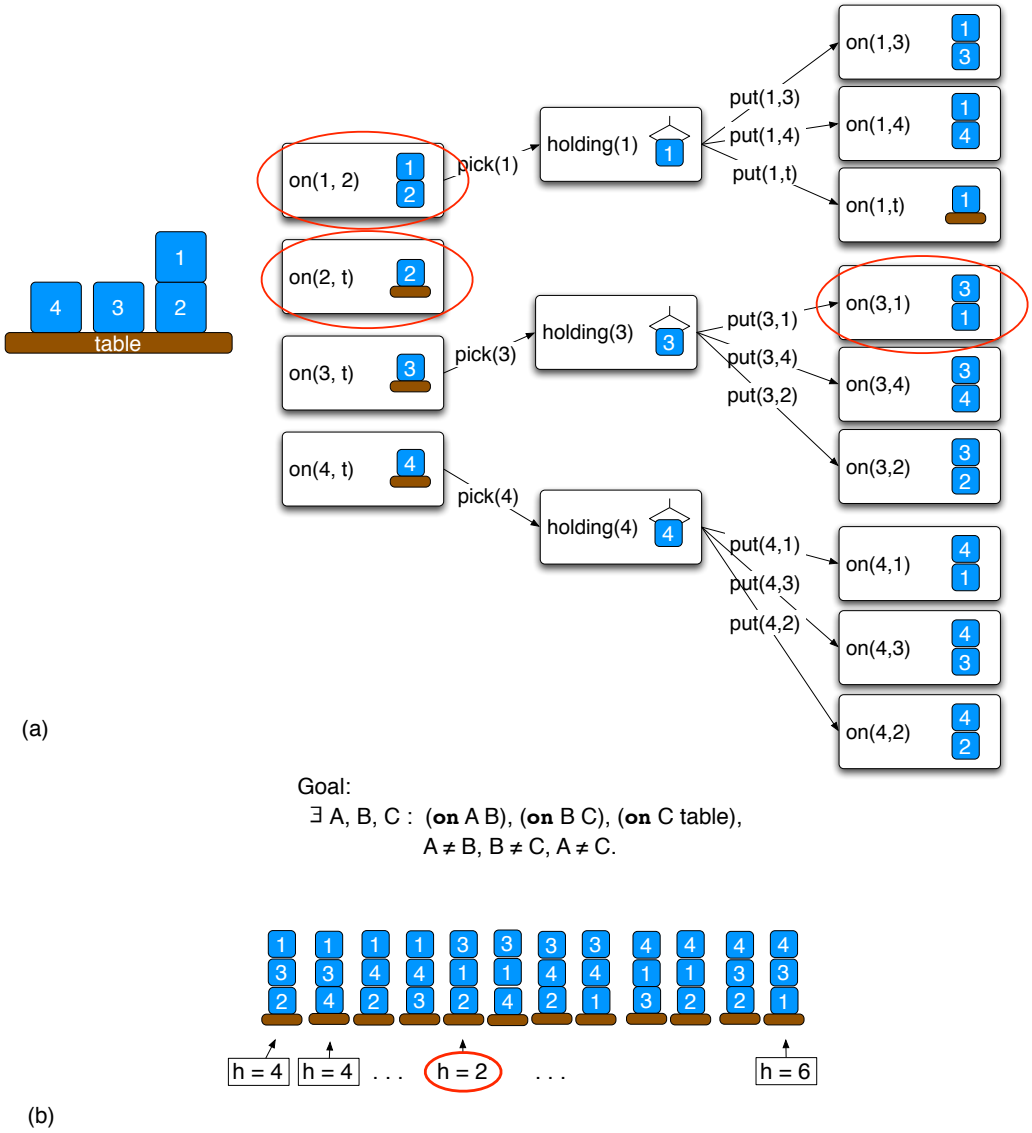
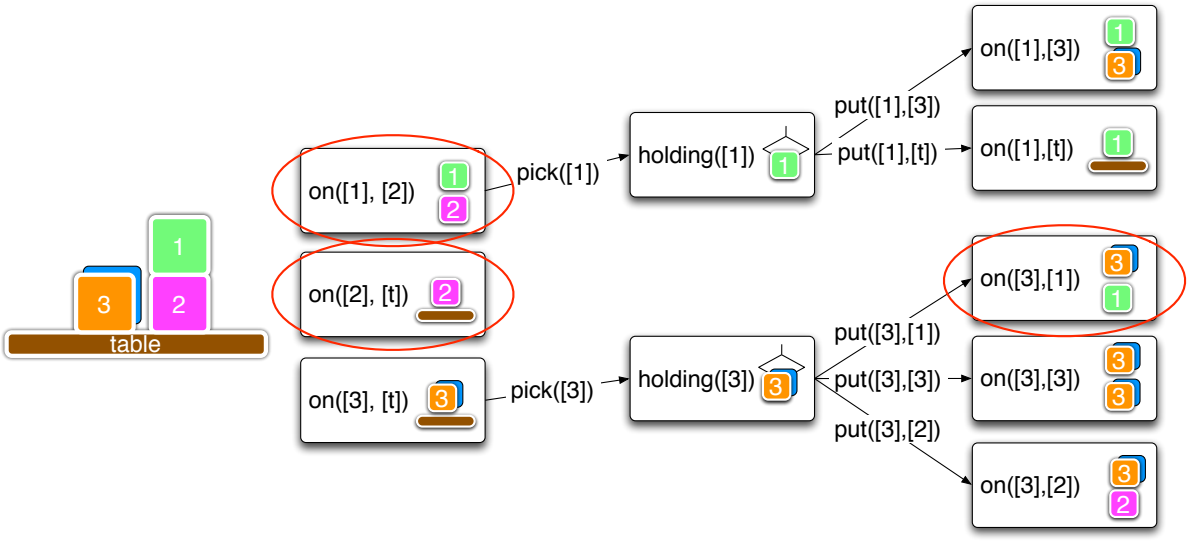


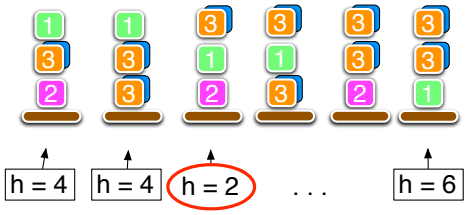
Figure 5-3: Calculation of the heuristic with no equivalence classes. We have to find the smallest heuristic value amongst all the possible bindings for the goal: the possible bindings that the plan graph represents as being possibly true at this stage are shown at the bottom of the figure. There are twelve of them. The set of proposition nodes given by the fifth binding in the list result in the lowest heuristic estimate, $h = 2$. These proposition nodes are encircled in the graph.



(a)

Goal:

$\exists$ A, B, C : (**on** A B), (**on** B C), (**on** C table),
A $\neq$ B, B $\neq$ C, A $\neq$ C.



(b)

Figure 5-4: Calculation of the heuristic, this time with equivalence classes. Now there are only six bindings amongst which to search. The *one-of()* operator is also in effect in the heuristic computation, which ensures that we correctly determine the set of propositions that may legally satisfy the goal.

every proposition that could possibly be true. If a satisfying assignment is found, we return the number of actions required to produce the propositions needed for the goal. In order to be an admissible heuristic, we must look for the *smallest* number of actions. The only way to do this is to consider all possible satisfying assignments for the goal and to return the smallest action count out of all of them. In the example of Figure 5-1, the heuristic estimate is $h = 2$.

However, we should to take advantage of the fact that we know about some equivalence relationships in our initial state. We can certainly start off the initial layer of the plan graph with the set of canonical propositions in our initial state instead of the whole list of ground facts. That in itself seems simple enough. The next action layer is simply the set of ground actions applicable given the previous set of facts, just as before. However, when we calculate the effects this action layer, we are presented with a problem: how should we represent the effect of this action correctly? Consider, for example, the *pick-up([3])* action in Figure 5-4. The class *[3]* represents the class of blocks equivalent to block 3. As soon as we add the fact *holding([3])* to the planning graph, we represent the fact that we could be holding *any* block that was originally equivalent to block 3. This is consistent with the idea of the relaxed planning graph: it represents the superset of propositions achievable at a given plan-graph length.

This technique results in overall fewer propositions being created in the plan graph. This is potentially a large savings on the computational cost of the goal test. The fact that we are searching for the shortest relaxed plan (because we need an admissible heuristic) means that we must search for all possible bindings for the goal formula. This is a search that is, in the worst case, is exponential in the number of propositions that satisfy a clause in the formula.

5.2.3 Being more aggressive: minimal predicate set

A key component of dealing with potentially "imperfect" representations is not only grouping together similar objects, but, adapting the representation to eliminate further unnecessary distinctions. Imagine, for example, that we would like to move nails from one box to another. Suppose that as part of the initial domain description we

are given the metal composition of each nail. But, for the nail transfer task, knowing the composition of the nails doesn't matter. Thus, we would like to start out with some minimal set of predicates that depends on the goal, in order to group together as many nails as possible.

The basic idea is this: two objects in a state can be considered equivalent if there is an automorphism in the state that maps one object to the other. The more complex the graph, the fewer automorphisms there will be. Thus, if we can be aggressive about keeping the graph as simple as possible, by restricting the set of predicates for which we add edges, then, we will be able to group more objects together. We will use the term *basis set* to refer to the set of predicates used to construct the state relation graphs for a problem instance. In order to be as efficient as possible, we want to determine the *minimal* basis set that will let us solve our problem.

We compute this minimal set as follows. At the beginning of the planning process, we make a call to the heuristic solver from the initial state. The heuristic solver returns a plan to solve the relaxed planning problem using the original representation. We know that this plan is the most optimistic underestimate of the sequence of actions needed to achieve the goal. We compute the minimal basis β by starting from the starting from the goal and working backwards: for every proposition node used in satisfying the goal, called a *subgoal*, add the corresponding predicate to β. Then, work backwards through each action: take as the next "subgoal" the precondition nodes of the action in the last layer.

By the time we reach the first layer in the graph, we will have included all the predicates necessary to express the satisfaction of the goal and the satisfaction of a set of actions that can achieve the goal.

Thus, our complete forward-search algorithm is as follows:

Input: Initial state s_0, goal condition g, set of rules Z

Output: Sequence of actions from s_0 to a goal state

Find a minimal representational basis, β

Canonicalize $s_0 \rightarrow \tilde{s_0}$

Initialize agenda with $\tilde{s_0}$

while *agenda is not empty* **do**
    Select and remove a state s from the agenda
    if *s satisfies goal condition g* **then**
        return *path from root of search tree to s*
    else
        Find representative set $\mathcal{A}'$ of actions applicable in s
        foreach $a \in \mathcal{A}'$ **do**
            Add the successor of s under a to the agenda

Algorithm 4: REBP forward-search algorithm.

As an illustrating example of this algorithm, we again turn to a small blocksworld problem in Figure 5-5. The objective is to find three blocks that form a stack. The initial, fully-ground state contains two blocks on the table, and two blocks in a two-block stack. The relations in our domain are `on-top-of(?b - block,?x - object)` and `holding(?b - block)`, and the operators are `pick-up(?b - block,?x - object)` and `put-down(?b - block,?x - object)`.

Our final system is described schematically in Figure 5-6.

5.2.4 Planning experiments

In this section, we show some simple experiments intended to illustrate the computational advantages possible with the equivalence-based approach. We compare four algorithms in three domains.

The first algorithm we call the *baseline*: it is a simple state-based, best-first, forward-search planner that uses our adapted FF heuristic to guide its search. In the event that the heuristic evaluates two or more actions with the same heuristic value,

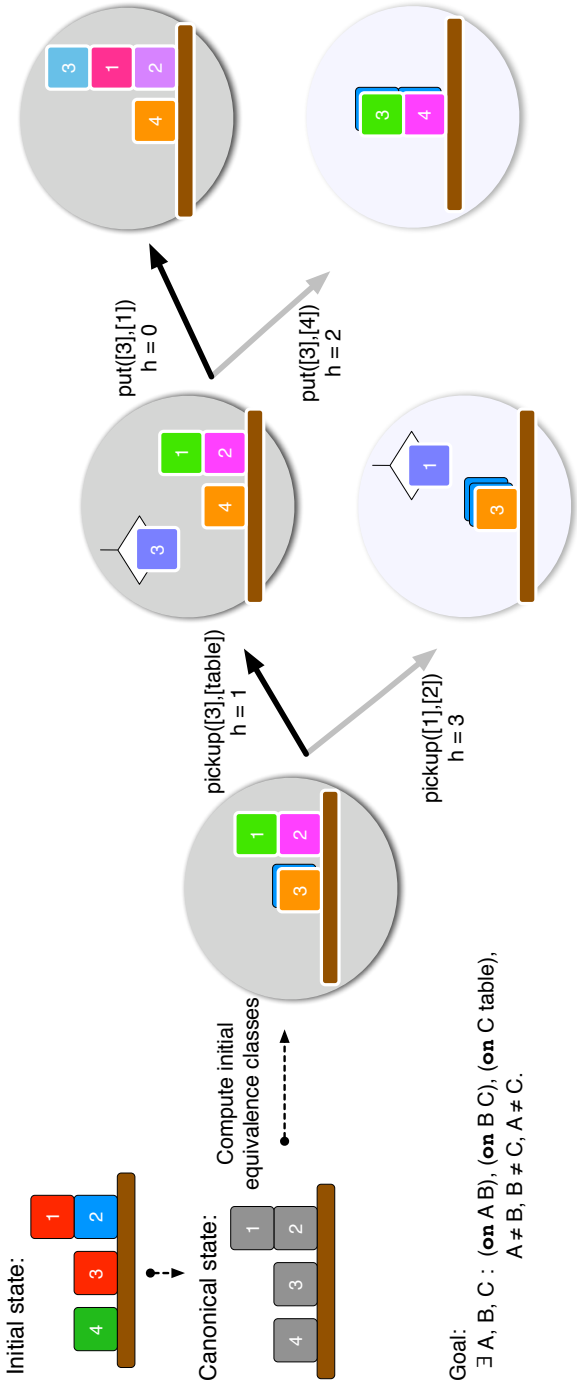


Figure 5-5: Illustration of the REPB search algorithm. The task is to achieve a stack of any three blocks. We start with the initial state as shown. The object equivalence classes are then computed, resulting in an abstract version of the initial state. Then, as we search for the goal in successive states, we update the equivalence classes of the objects in the domain. We stop when we determine that the goal can be satisfied; in this case, after a sequence of pick-up and put-down actions.

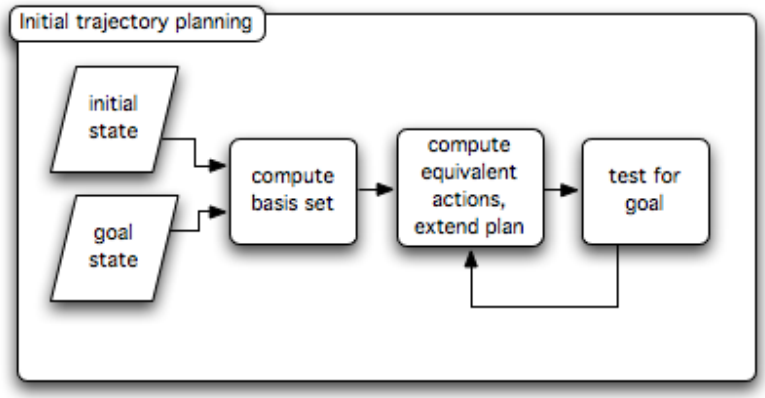


Figure 5-6: The REBP system. Given a problem description with an initial state and a goal, the system first attempts to determine automatically the minimal basis set of predicates it needs to reach the given goal. Then, given this basis set, the system executes a forward-chaining, heuristically guided search, until the goal is reached.

it selects one of those actions at random. The baseline algorithm does not compute equivalence classes nor try to reduce the basis set of predicates — it works in the original, propositional, state space.

Second, we have REBP-full: this is the baseline planner with the ability to compute equivalence classes among actions and objects. It does not try to reduce the basis set, however.

Third is REBP-min, which extends REBP-full by approximating the original representation with a minimal basis set.

These planners are implemented in Java with an eye to correctness rather than speed, and are thus not as fast as optimized or well-tuned implementations. Therefore, we also compare against a freely available, highly efficient, implementation of FF itself [57]. Also, we only compare planning time. Solution length for FF may vary, since it is not an optimal planner.

The first set of experiments explores the most simplest setting possible. We used the blocks world domain of the 2004 ICAPS planning competition for the dynamics, and we set up a series of very simple problem instances: starting with three blocks on the table, and growing up to 100 blocks on the table, the goal is to place three blocks in a stack. A plot of planning time vs. domain size is shown in Figure 5-7. In this

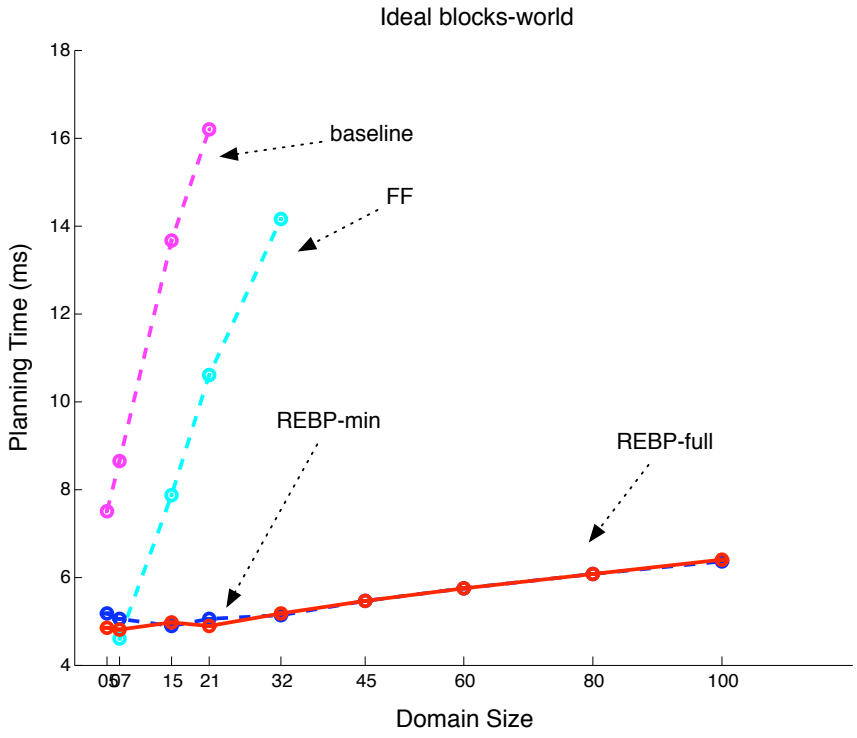


Figure 5-7: Planning time vs. domain size for best-case blocksworld. Equivalence classes are helpful as the domain gets bigger.

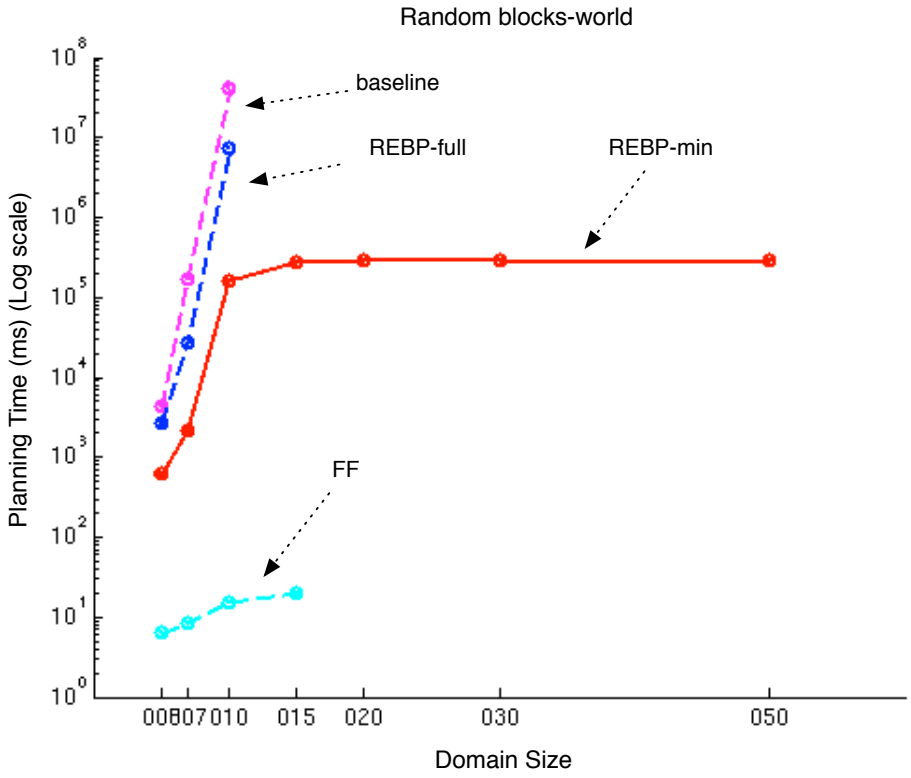


Figure 5-8: Planning time vs. domain size for random blocks-world domain. Equivalence classes are helpful as the domain gets bigger, and reducing the basis set of predicates yields computational savings.

experiment, all the blocks are the same color, so, nothing is gained by ignoring the color predicates, which are not necessary to satisfy the precondition of the *pick-up* or *put-down* actions, or the goal. Thus, as we expect, REBP-full and REBP-min perform exactly the same.

The second set of experiments also explores planning time as a function of domain size, but in a slightly less straightforward way. We picked a random 5-block problem instance from the 2004 planning competition archives. Then, we produce problem instances of increasing size by replicating this initial state. A plot of planning time vs. domain size is shown in Figure 5-8. The initial problem instance contained blocks of three different colors, so, REBP-a can reduce its computational effort by reducing its basis set to just *on-top-of* and *holding*, which it does.

The third set of experiments was done in an adaptation of the AIPS 2002 "depot"

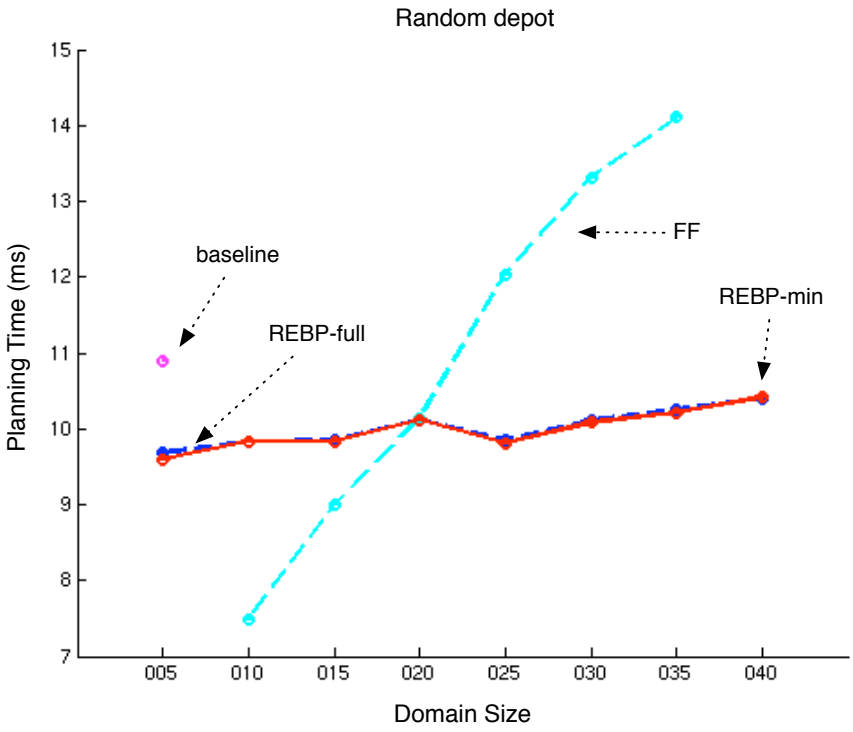


Figure 5-9: Planning time vs. domain size for random depot domain. As in the blocks world, equivalence classes are helpful as the domain gets bigger, and reducing the basis set of predicates yields computational savings.

domain [58], a logistics domain. A problem instance in this domain consists of a set of trucks, hoists, pallets, crates, distributors, and depots. The trucks and crates are initialized randomly among the distributors, and the objective is to move any two crates to the target pallet at the depot. As in the blocks-world domain above, increasing the number of crates does not lengthen the solution; but, the increased number of objects proves difficult to handle for the FF and baseline algorithms. The plot of planning time vs. domain size is shown in Figure 5-9.

5.3 Complexity issues

A question that immediately arises is whether it is wise to embed the computation of graph automorphisms in our search loop. The difficulty of the graph isomorphism a long-standing open question in the field of complexity theory, and it is currently unknown whether the problem is NP-complete. One can construct instances in which even a well-regarded algorithm such as *nauty* [46] is forced to do an exponential-time search for an isomorphism. However, for a broad class of graphs, there also always exist conditions under which *nauty* can run in polynomial time [47]. Our experience has shown that searching for isomorphisms can be fast when extremely constrained by labeling and typing, as in our case. In our empirical studies, the amount of time spent computing isomorphisms remains a small fraction of the total execution time.

The computation time of the algorithm is more severely affected by the heuristic evaluation, since it is potentially an exponential operation for each action that must be evaluated.

Chapter 6

Computing an abstract envelope

In this chapter, we will see how to use the output of our deterministic planning process to bootstrap an MDP and directly address uncertainty in the domain.

The output of the planning phase is a sequence of canonical actions, which corresponds to a sequence of canonical states. The canonical states are represented in a *basis set* of predicates that may be smaller than or equal to the set of predicates originally given in our domain description.

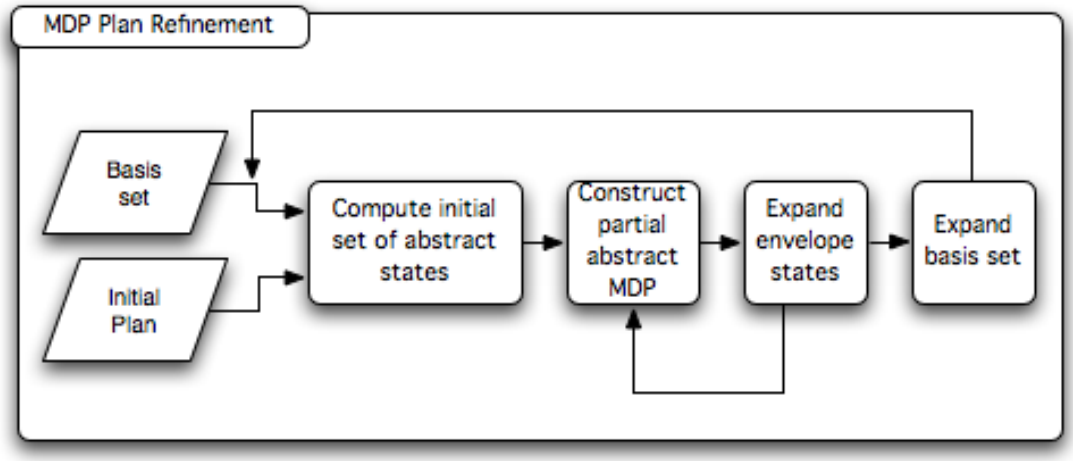


Figure 6-1: In the second part of the REBP system, an envelope MDP is constructed from the output of the planning process. The envelope MDP, and the basis set of predicates used to express the MDP, will be expanded and refined in subsequent steps of the algorihtm.

We will use this abstract state sequence to initialize an *envelope* MDP, which was defined in earlier sections. We will manipulate this envelope MDP in two ways: first, as in the original Plexus algorithm, we will sample from our policy and incorporate deviations from the initial path; second, new to this work, we will incorporate modifications to the representation to increase the accuracy in our value estimate. The overall system is described schematically in Figure 6-1.

6.1 Interval envelope MDP

To have it fresh in our minds, we recall the basic steps of the original Plexus algorithm for atomic-state MDPs:

Determine initial sequence of states

Compute transition probabilities between all states for all applicable actions

while *Have time to compute* **do**

- Compute policy (value iteration)
- Sample deviations from envelope and expand

Algorithm 5: Basic envelope algorithm for atomic-state MDPs.

However, now, each state in our MDP is an abstract state, and it stands for a set of underlying ground states as illustrated in Figure 6-2. Thus, we cannot simply represent a transition between such states with a point probability. We must allow for the possibility that the dynamics driving the transition may be different depending on which underlying ground states are participating in the transition.

Instead, we represent each transition probability as an interval, as depicted in Figure 6-3. Interval MDPs, and the corresponding Interval Value Iteration algorithm, were first presented by Givan *et al.* [28, 29].

Let us formally define an abstract-state, interval envelope MDP (called an AMDP for short) as an extension of the basic relational MDP we saw in Section 1.3.3. An AMDP M is a tuple $\langle \mathcal{Q}, \mathcal{P}, \mathcal{Z}, \mathcal{O}, \mathcal{T}, R \rangle$, where:

States: The abstract state space, $\mathcal{Q}^*$, is defined by a basis set $\mathcal{P}$ of relational pred-

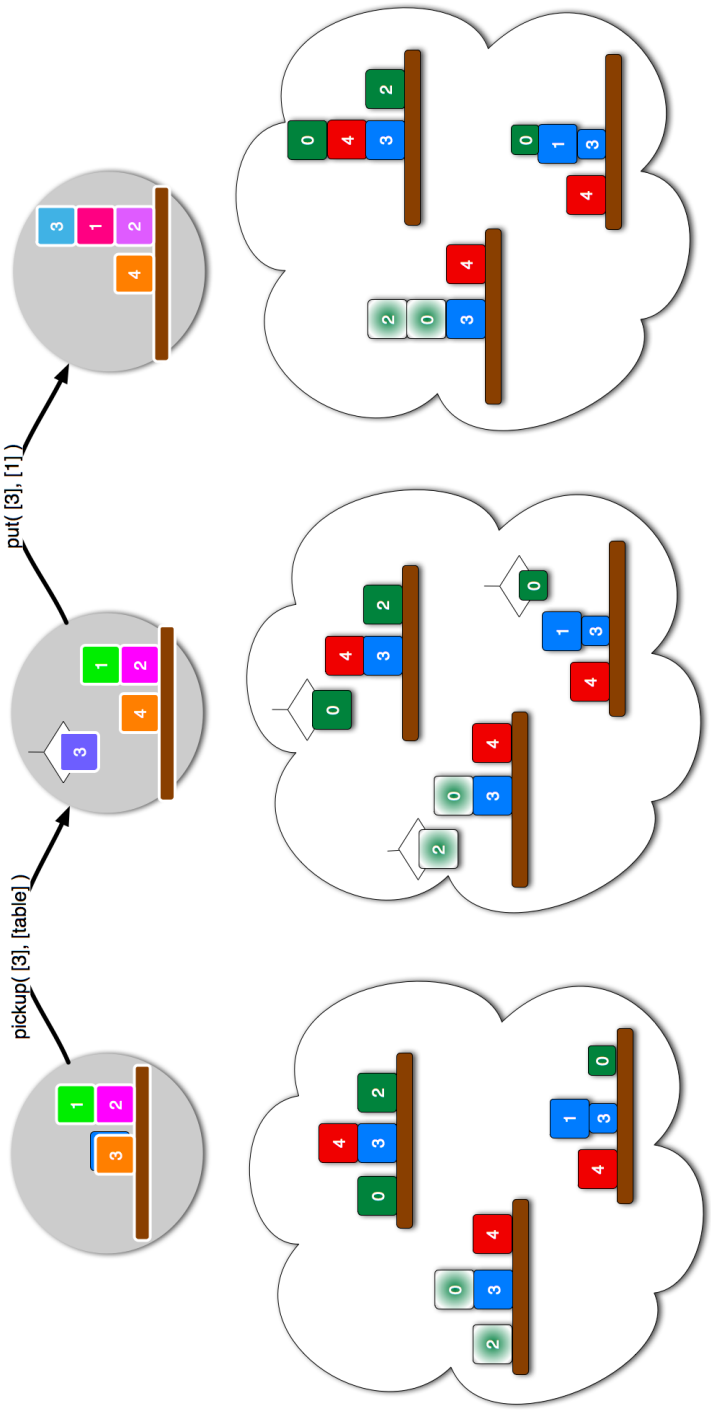


Figure 6-2: The sequence of canonical states may actually represent a collection of underlying state transitions. Each canonical state represents the set of underlying ground states consistent with the basis set of predicates used to express the canonical state. The objects in the underlying states may have relationships and properties not represented in the canonical state, such as color, texture, or size, for example.

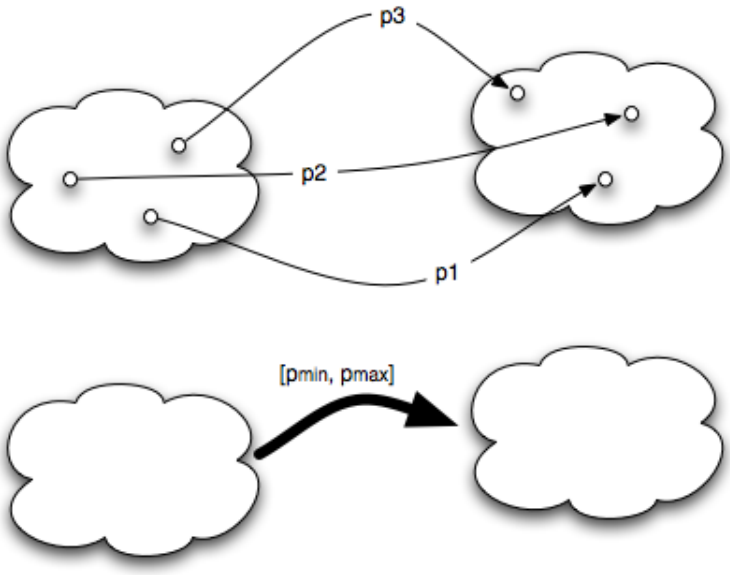


Figure 6-3: Each canonical, or abstract, state in the MDP describes a set of underlying ground states. Transitions between abstract states correspond to a collection of underlying ground transitions, denoted above by scalar probabilities p_1, p_2, and p_3. Thus, we will represent the transition between two abstract states as an interval, whose upper bound is the largest underlying probability and whose lower bound is the smallest underlying probability.

icates, representing the relations that hold among the equivalence classes of domain objects, $\mathcal{O}$. The set of states $\mathcal{Q}$ of M, is the union of the set $\mathcal{Q}' \subseteq \mathcal{Q}^*$ and a special state q_{out}. That is, $\mathcal{Q} = \mathcal{Q}' \cup \{q_{out}\}$. The set $\mathcal{Q}'$, also called the *envelope*, is a subset of the entire abstract state space, and q_{out} is an additional special state that captures transitions from any $q \in \mathcal{Q}'$ to a state outside the envelope. Through the process of envelope expansion, the set of states $\mathcal{Q}$ will change over time.

Actions: The set of actions, $\mathcal{A}$, of M is composed of the ground instances of set of rules $\mathcal{Z}$ applicable in the envelope states of M.

Transition Dynamics: In an interval MDP, $\mathcal{T}$ gives the interval of probabilities that a state and action pair will transition to another state: $\mathcal{T} : \mathcal{Q} \times \mathcal{A} \times \mathcal{Q} \rightarrow [\Re, \Re]$. We will see how to compute this transition function in the sections below.

6.1.1 Initializing the abstract-state envelope

In this section, we look at how to compute initial set of states $\mathcal{Q}$ corresponding to the plan produced by the planning phase.

Each state $q_i \in \mathcal{Q}'$ of our AMDP M is a composite structure consisting of:

- $\tilde{s}_i$: a canonical state, in which we represent only the relations among the representatives of each object equivalence class.
- S_i: a set of underlying ground states consistent with the above canonical state.

The first state, q_0, of M is computed from the initial state of the planning problem straightforwardly: the set S_0 is initialized to contain the ground initial state of the planning problem, and the canonical state $\tilde{s}_0$ is the canonical version, with respect to basis $\mathcal{P}$, of the initial state.

We compute the second state, q_1, by taking the first action, a_0, in our plan. The next canonical state $\tilde{s}_1$ is computed by propagating $\tilde{s}_0$ through a_0 by the procedure described in Section 5.2.1. The ground state of q_0 can be efficiently propagated as well, and, we add the result to S_1. This procedure is repeated until we've processed the last action.

More formally, the procedure to compute the envelope, $\mathcal{Q}'$, from a plan p is:

Input: Canonical Initial State $\tilde{s_0}$, Plan p, Basis β

Output: Set of envelope MDP states $\mathcal{Q}'$

Initialize q_0 with $\tilde{s}_0$ and with $S_0 = \{s_0\}$

Initialize $\mathcal{Q}' = \{q_0\}$

foreach *action* a_i *in* p, $i = 0 \ldots n$ **do**

- Propagate $\tilde{s}_i$ to obtain $\tilde{s}_{i+1}$
- Propagate the s_i in S_i to obtain s_{i+1}
- Initialize q_{i+1} with $\tilde{s}_{i+1}$ and with $S_{i+1} = \{s_{i+1}\}$
- $\mathcal{Q}' = \mathcal{Q}' \cup \{q_{i+1}\}$

Algorithm 6: Procedure to compute a set of envelope states given a plan.

At this point, we have a set of MDP states $\mathcal{Q}' = \cup_{i=0}^{n+1}\{q_i\}$. To complete the set of states, $\mathcal{Q}$, we add the special state q_{out}.

This procedure lets us keep a record of the true ground state sequence, the s_i's, as we construct our model. Why do this, when we've gone through so much trouble to compute the canonical states? The problem is not that any individual ground state is too large or difficult to represent, but, that the combined search space over all the ground states is combinatorially much larger. If we do not keep the ground information around in some form, it will be impossible to determine how to modify the basis set later.

While each MDP state keeps around its underlying ground state for this purpose, it is only the canonical state that is used for determining behavior. Since a canonical state represents a collection of underlying ground states, the policy we compute using this approach actually encompasses more of the state space than we actually physically visit during the planning process.

6.1.2 Computing transition probabilities

Now that we have a set of states $\mathcal{Q}$, we need to determine the transitions, and the probability of those transitions, between the states in $\mathcal{Q}$. This computation proceeds in two phases. First, we compute the nominal interval probabilities of transitioning between canonical states. Second, we sample from our underlying state space to flesh out the *interval* of probabilities describing each transition. Below, we will speak of *updating* a probability interval $P = [a, b]$ with the probability p, which means: if $p < a$, then P becomes $[p, b]$; if $p > b$, then P becomes $[a, p]$

The computation of the nominal interval probabilities proceeds as follows:

1. For each state q_i, we find the set of actions A_i applicable in $\tilde{s}_i$.

2. For each action $a_k \in A_i$, and each state $q_j \in \mathcal{Q}'$ we compute the transition probability between q_i and q_j:

   (a) Initialize the ground transition probability. That is, take the first ground state in S_i and propagate it through action a_k. If the resulting ground state is equivalent to q_j with respect to the basis $\mathcal{P}$, and p is the probability of the outcome of a_k corresponding to that transition, then set the probability of transitioning from q_i to q_j via action k as the interval $P_{ijk} = [p, p]$.

   (b) For each remaining ground state $s \in S_i$, compute the probability, p', of transitioning to q_j via action a_k.[1] *Update* the interval P_{ijk} with p'.

3. Compute the probability of transitioning to q_{out} from q_i and a_k. This involves keeping a list, as we execute the above steps, of the out-of-envelope probability for each ground application of the action a_k. More precisely: for each $s \in S_i$, when we apply a_k and detect a transition of probability p to a state within the envelope, we update a_k's out-of-envelope probability with $1 - p$. This ensures

[1] Strictly speaking, we will need to use the inverse of the mapping ϕ between s and $\tilde{s}_i$ to translate the action a_k into the analogous action applicable in s. This is because, while s may belong to the equivalence class of states represented by $\tilde{s}_i$, it may have objects of different actual identities belonging to each object equivalence class.

that the out-of-envelope probabilities are consistent for each representative action, a_k.

The above procedure computes the basic transitions and transition probabilities for our MDP.

Next, in order to improve our interval estimates, we do a round of sampling from our model. The idea is to see if we can uncover, via this sampling, any ground states that yield transition probabilities outside of our current interval estimate. This is done as follows:

1. For each state $q_i \in \mathcal{Q}'$, action $a_k \in A_i$, and state $q_{j \neq i} \in \mathcal{Q}'$: let the ground state s' be the result of propagating a state $s \in S_i$ through a_k.

   (a) If there exists a state q_k such that the probability of transitioning from s' to q_k under an action is outside of the current interval for transition of q_j to q_k for that action, add s' to S_j.

Let us work through an example. To do that, we introduce one of our experiment domains, called the *slippery blocks* world. This domain is an extension of the standard blocks world, except that in this domain, our ability to successfully pick up or put down a block is modified when the block is "slippery." Green blocks are slipperier than red or blue blocks. In Figure 6-4 we see the PPDDL description of this domain. Note the *conditional effect* in each action: if the block is green, then the probability of a successful pick-up outcome changes from 0.9 to 0.6; likewise, the probability of a successful put-down outcome changes from 0.9 to 0.6. Please see Figures 6-5 through 6-9 for a detailed example of the procedures just described.

Once we have our transition probabilities, we compute the policy on the MDP by following the interval value iteration algorithm of Givan *et al.* [28, 29]. Interval transition probabilities result in interval estimates of value. In our implementation, a state's value is reported as *average* value in the interval.

```
(define (domain slipperyblocks)

 (:types block table - object)
 (:constants table - table)

 (:predicates
  (is-red ?block - block)
  (is-blue ?block - block)
  (is-green ?block - block)
  (holding ?block - block)
  (on-top-of ?block - block ?obj - object)
 )

 (:action pick-up-block-from
  :parameters (?top - block ?bottom - object)
  :precondition
    (and (on-top-of ?top ?bottom)
         (not (= ?top ?bottom))
         (forall (?b - block) (not (holding ?b)))
         (on-top-of ?top ?bottom)
         (forall (?b - block) (not (on-top-of ?b ?top))))
  :effect
  (and
    (when (is-green ?top)
       (probabilistic
        0.6 (and (holding ?top) (not (on-top-of ?top ?bottom)))
        0.4 (and (on-top-of ?top table) (forall (?b - block) (not (on-top-of ?top ?b))))))
    (probabilistic
     0.9 (and (holding ?top) (not (on-top-of ?top ?bottom)))
     0.1 (and (on-top-of ?top table) (forall (?b - block) (not (on-top-of ?top ?b)))) ))
  )


 (:action put-down-block-on
  :parameters (?top - block ?bottom - object)
  :precondition
   (and (not (= ?top ?bottom))
        (holding ?top)
       (not (holding ?bottom))
        (or (= ?bottom table)
            (and (forall (?b - block) (not (on-top-of ?b ?bottom))))))
  :effect
  (and
    (not (holding ?top))
    (when (is-green ?top)
        (probabilistic 0.6 (on-top-of ?top ?bottom)
                       0.4 (on-top-of ?top table)))
    (probabilistic 0.9 (on-top-of ?top ?bottom)
                   0.1 (on-top-of ?top table)))
  )
)
```

conditional effect

conditional effect

Figure 6-4: The slippery blocks domain. This is an extension of the standard blocks world in which green blocks are "slippery" and are thus more likely to be dropped on the table.

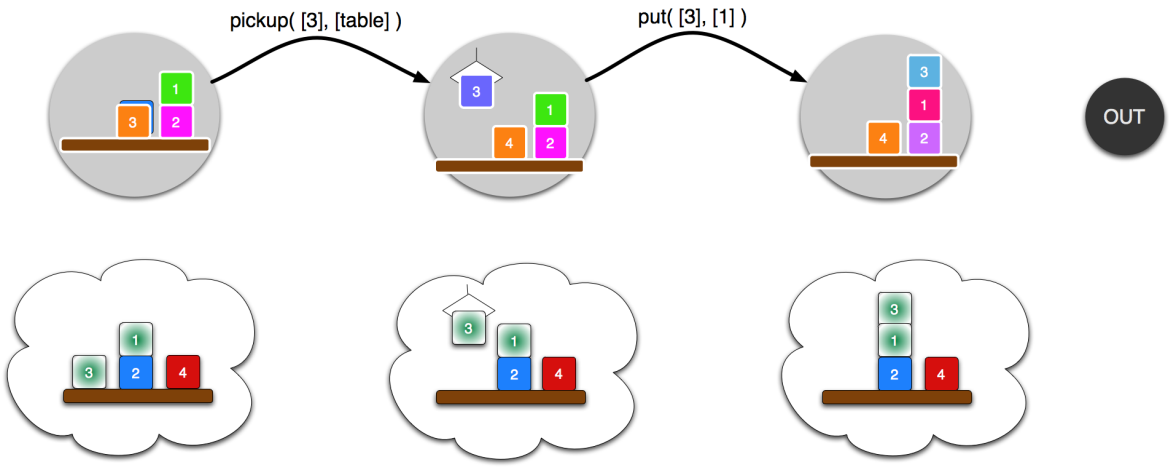


Figure 6-5: First, start with a newly initialized envelope corresponding to the example planning task of Figure 5-5. At this point, we have created the set of states $\mathcal{Q}$, consisting of each canonical state, its ground version, and the state q_{out}.

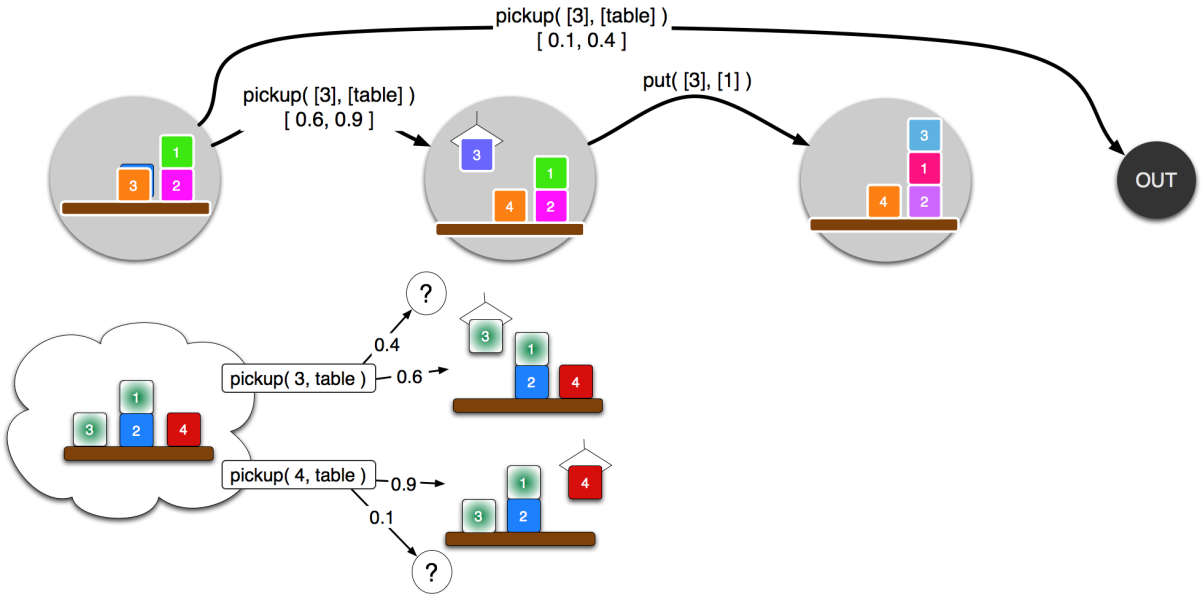


Figure 6-6: Second, we compute the nominal transition probabilities. In this case, there are two ground actions equivalent to the `pickup([3],[table])` action applicable in the canonical state. These actions yield an interval probability of $[0.6, 0.9]$ of transitioning to the second state, and an interval probability of $[0.1, 0.4]$ of falling out of the envelope.

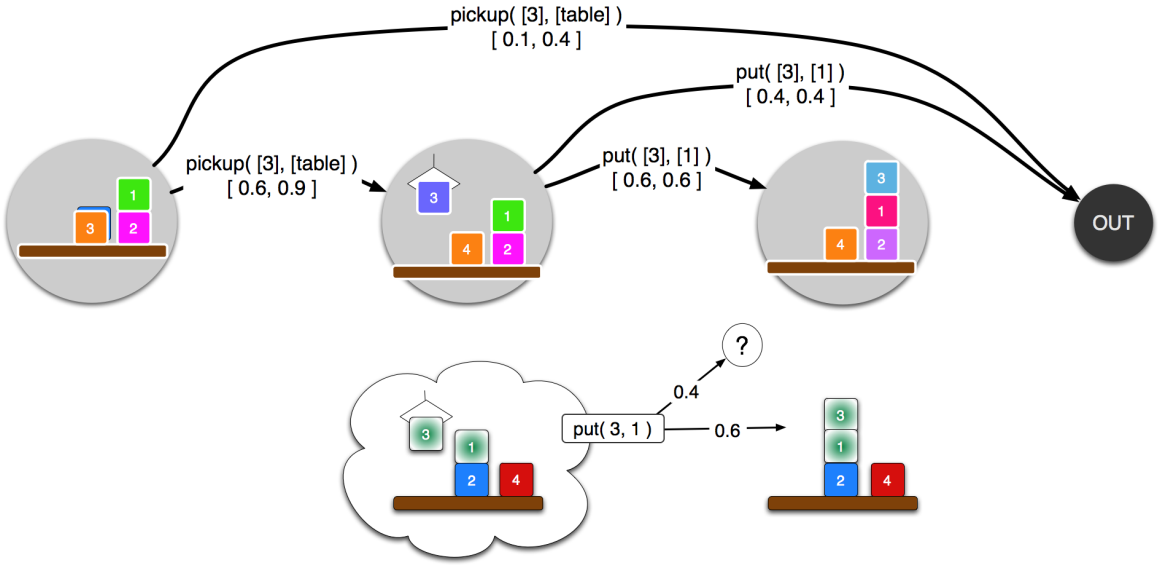


Figure 6-7: Third, we do the same for the second state: the ground actions (only one in this case) yield a probability interval of $[0.6, 0.6]$ of transitioning to the third state, and a probability interval $[0.4, 0.4]$ of transitioning out of the envelope.

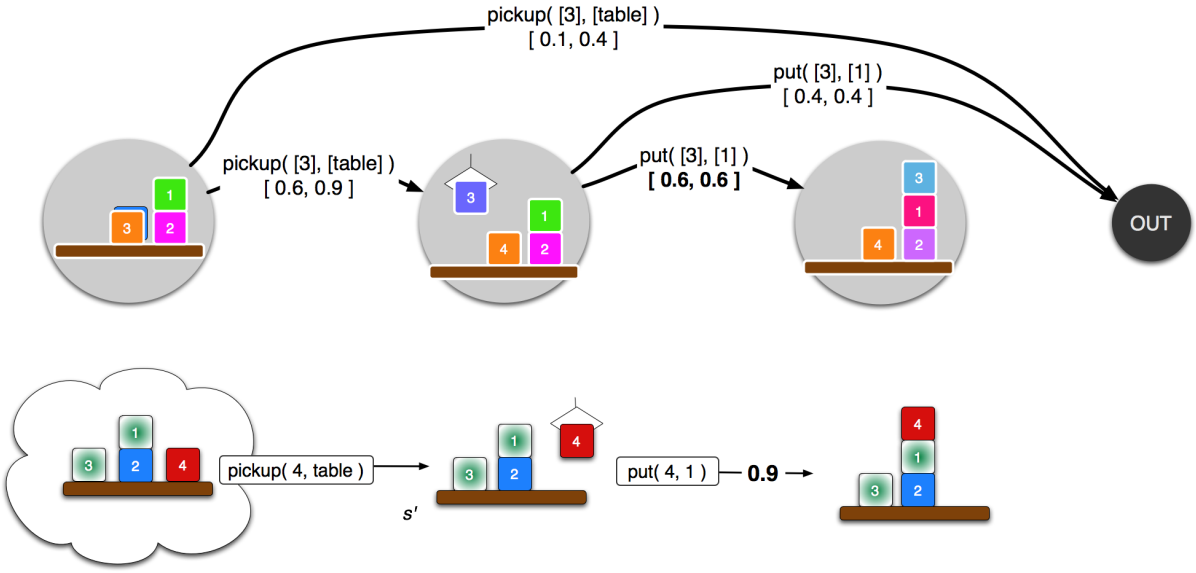


Figure 6-8: Fourth, we sample from our model in order to improve our interval probability estimates. We see that adding the ground state s' into S_1 changes our estimate of the types of transitions that can occur between the second and third canonical states.

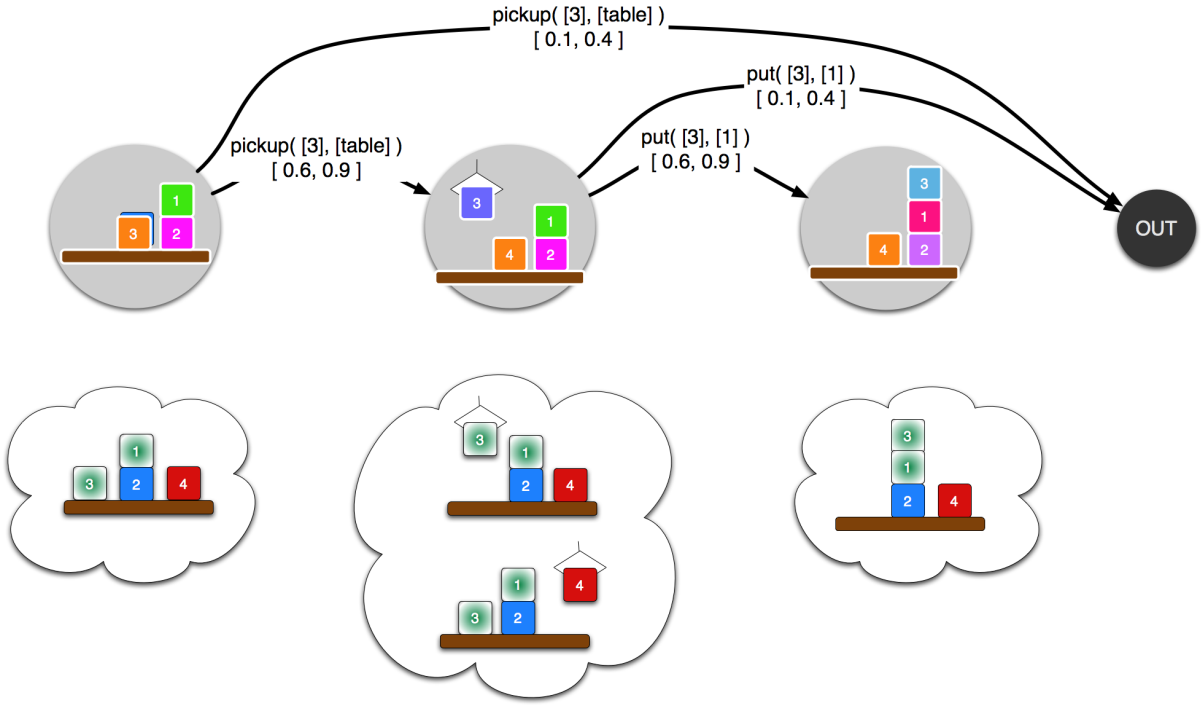


Figure 6-9: Fifth, and finally, is our completed abstract envelope MDP. From here, we are ready to do a round of policy improvement (via value iteration) and envelope expansion.

6.2 Proposing a change to the representation

Having formulated our MDP in this way, with an abstract state space and with probability intervals represented as intervals, we are confronted very naturally with a type of *structure search* problem. We may want to add a predicate, or set of predicates, into the representation basis in order to tighten these intervals, and, consequently, to lessen the uncertainty in our value estimates.

The point of augmenting the basis set is to be able to express transition probabilities, and thus the expected value of a policy, more precisely. The next logical question is how to frame a procedure for representation. We begin by noting that the transition probabilities are encoded in the rule schemas given as part of our planning description. Therefore, in our case, a representation that is missing some potentially useful predicates suffers from an inability to apply a necessary action, or, in an inability to effect a particular conditional outcome of that action. For example, in the slippery blocks world case, the minimal basis ignores color completely. While this representation speeds up the planning, by allowing blocks of different colors to be put into the same equivalence class, it does not allow modeling the fact that blocks of color green will experience a transition via the conditional outcome of the pick-up and put-down actions, and thus a different transition probability.

The basic mechanism is to add a function, called `proposeBasis()`, which takes as an argument a rule and the current basis, and returns a list of candidate predicate sets to be added to the basis. What does it mean for an operator to "suggest", or propose, a predicate set? Consider an operator with a condition w on an effect. If we are interested enabling this effect, then the operator must propose the set (which may simply be a singleton) of required predicates missing from our representation. If more than one additional predicate is required to express a condition, then no benefit will be observed until all required predicates have been added, one at a time. This is a classic structure-search issue. Because we know we are dealing with rule schemas whose conditions have this characteristic, we can take the shortcut of proposing sets of predicates at a time.

There are two places in the algorithm in which to refine the representation. The first is as a part of the existing envelope-refinement loop. As part of that, we keep a sorted list of the transitions in our MDP. Currently, we sort transitions in descending order by width of the interval; i.e., the maximally uncertain transitions are at the top of the list.[2] Then, when we need to suggest a refinement, we start with the transition at the top of the list and request its proposal.

The second opportunity comes when we reach a representational "failure" point. In the process of sampling from actions that were not originally in our optimistic plan, and, thus, made no contribution the original choice of basis, computing their effects might have unexpected results. This becomes obvious when we produce an outcome state that has no applicable actions. We call this a "failure" point, and we deal with it as follows. First, we remove, as much as possible, the trajectory that leads to this state. We do this by iterating backward from the failed state until we either reach the initial state, or, a state that has more than one incoming transition. At this point, we re-route that single outgoing transition to the OUT state. We set a flag that disallows any future sampling from that action. Then, starting from the offending state, we work our way backwards through the transitions until we find a transition with that has a non-empty predicate set to propose. If we do find one, we add this proposal to a high-priority list of candidate predicate sets. Then, the next time the MDP considers a new proposal, it selects from this list.

Once we have determined a proposal to try, we initialize a new MDP using the original plan and the *new* basis. Then, the regular phases of policy improvement and envelope expansion happen for both of them in parallel. We can add as many parallel MDPs as desired. In our current implementation, we keep no more than 5 interval MDPs in parallel.

At any given time, the policy of the system is that of the MDP with the highest policy value.

[2]We could imagine sorting this list by other metrics. For example, we could be risk-averse and sort them by the lower value bound.

So, the general REBP algorithm is:

Input: Initial state s_0, Goal condition g, Set of rules Z

Compute minimal basis representation, β

Let plan P = REBPForwardSearch(s_0, g, Z) `/* Algorithm 4 */`

begin Initialize envelope MDP M with P and β :
- Compute transitions and transition probabilities for M
- Do interval value iteration in M until convergence

end

Initialize a list of MDPs $m = \{M\}$

while *have time to deliberate* **do**
- **foreach** MDP M_i *in* m **do**
  - Do a round of envelope expansion in M_i
  - **if** *failure to find applicable action in a state* q' **then**
    - Remove the q' from M_i
    - Select the first non-empty proposal basis, β', corresponding to the sequence of actions between q' and q_0
    - **if** β' *not empty* **then** append to the front of the list of proposals, l_i
  - **else**
    - Sort transitions of M_i in descending order
    - Compute a proposal basis β' from the top transition
    - **if** β' *not empty* **then** append β to end of list l_i
  - Do interval value iteration in M_i until convergence
  - **if** l_i *not empty* **then**
    - Select a basis β' from the list
    - Construct a new MDP M' with plan P and basis β'.
    - Append M' to list m of MDPs.
- Sort the list m by decreasing average policy value

Algorithm 7: Overall REPB algorithm.

	Basis type			
	Adaptive	Fixed, minimal	Fixed, full	Propositional (no classes)
Initial plan	`adap-init`	`minb-init`	`fulb-init`	`prop-init`
Start state	`adap-null`	`minb-null`	`fulb-null`	`prop-null`

Table 6.1: The matrix of experiments. We compared two ways of initializing the envelope MDP — with the output of the planning phase ("Initial plan"), and with only the initial state ("Start state") — with four ways of working with the basis representation. The last column, ("Propositional"), does no equivalence class computations.

6.3 Experiments

In this section we examine a set of experiments done in four different domains. The objective in each domain is to compute a high-valued policy with as compact a model as possible. We will look at the various ways of combining the techniques described in this work, and we'll try to identify the impact of each on the behavior we observe. We describe the different algorithms below.

Complete Basis + Initial Plan (`fulb-init`): This is the basic relational envelope-based planning algorithm. A plan is found in the original representation basis, and this plan initializes a scalar-valued envelope MDP.

Minimal Basis + Initial Plan (`minb-init`): This is an extension of REBP that first computes a minimal basis set for the representation. Because it uses a scalar-valued MDP, no basis modification is done.

Adaptive Basis + Initial Plan (`adap-init`): This is the full technique: computation of a minimal basis plus an interval MDP for basis and envelope expansion.

No initial plan (`fulb-null`, `minb-null`, `adap-null`): We also control for the impact of the initial plan by combining each style of equivalence-class representation with a trivial initial envelope consisting of just the initial planning state.

Propositional (`prop-init`, `prop-null`): Finally, to control for the impact of the equivalence classes, we initialize a scalar-valued MDP in the full, propositional

representation (this is the "baseline" algorithm of Chapter 4, which does not compute equivalence classes) with an initial plan, and with the initial state, respectively.

The four domains are:

Blocksworld: this is simply the standard blocks world, the same one we saw in the second set of experiments in the last chapter. We include this to get a baseline for the algorithms' behavior. The first problem instance contains 5 blocks, and the goal is to put five of them in a stack. The largest domain contains 50 blocks; all have the same goal. The PPDDL description of this domain was given in Figure 1-2 on page 23.

Slippery blocksworld: an extension of blocks world in which some blocks (the green ones) are "slipperier" than the other block. While color may be ignored for the purposes of getting a solution quickly, higher quality policies result from detecting that the color green is informative. The PPDDL description of this domain was given in Figure 6-4 on page 96.

Zoom blocksworld: a different extension of blocks world in which the action set is augmented by a one-step *move* action. This action gets things done quickly, but, is less reliable than a sequence of *pick-up* and *put-down*. However, in order to switch to using the *pick-up* action, the "holding" predicate must be in the representation. The PPDDL description of this domain can be seen in Figure A-10 on page 126.

MadRTS world: this domain is an adaptation of a real-time military logistics planning problem. [3] The world consists of a map (of varying size), some soldiers, and some enemies. The goal is to move the soldiers so as to outnumber the enemies at their location (the enemies don't move). However, the success of

[3] Our PPDDL planning problem was adapted from a scenario originally described by the Mad Doc Software company of Andover, MA in a proposal to address the Naval Research Lab's TIELT military challenge problem [48]. While no longer taking place in a real-time system, we call this planning domain the MadRTS domain to signal this origin.

a *move* action depends on the health of the soldier. A soldier can transfer, collect, and consume food resources in order to regain good health. The PPDDL description of this domain is in Figure A-15 on page 130.

We plot the results in two ways. We plot expected value (take as the expected value of state q_0 in the MDP, or the average of the interval, in the case of an interval MDP) vs. the number of states in the MDP, as a way of showing the value of the policy as a function of the size of the model. Next to each such graph, we also plot the same expected value as a function of the computation time (as measured by a CPU-cycle monitoring package). In the experiments which required no computation of a plan first, time is measured from the beginning of the construction of the initial MDP; for those that did, the first data point is plotted also after construction of the MDP plus the amount of time spent planning. To compute the accumulated reward, we ran 900 steps of simulation in each domain (corresponding to about 8 successful trials in the blocks worlds), selecting actions according to the policy, and selecting an action randomly %15 of the time. This was to force the policy to react to a situation which it might not have expected. In the interval MDPs, action selection is done by choosing the action with the highest *average* value. A reward of 1.0 was given upon attainment to the goal, and we report the average accumulated reward per step.

Let us examine some representative results from the experiments. More complete results can be found in Appendix A. Figure 6-10, contains a plot of value vs. number of MDP states for the adaptive and the minimal (fixed) basis algorithms in the 7-block slippery blocks world. The PPDDL description of this problem instance is in Figure A-1 on page 118 The interesting thing to note is that, even though the minimal-basis approaches get a similar expected value to the adaptive basis (0.4 for `adap-init` and `minb-null`), the adaptive-basis approaches are able to accumulate more reward in during execution. This is because not representing the `green` predicate results in a model that is slightly optimistic, and the MDP is unable to distinguish green blocks nor formulate a policy to avoid them. Thus, the adaptive-basis approach yields a more accurate model.

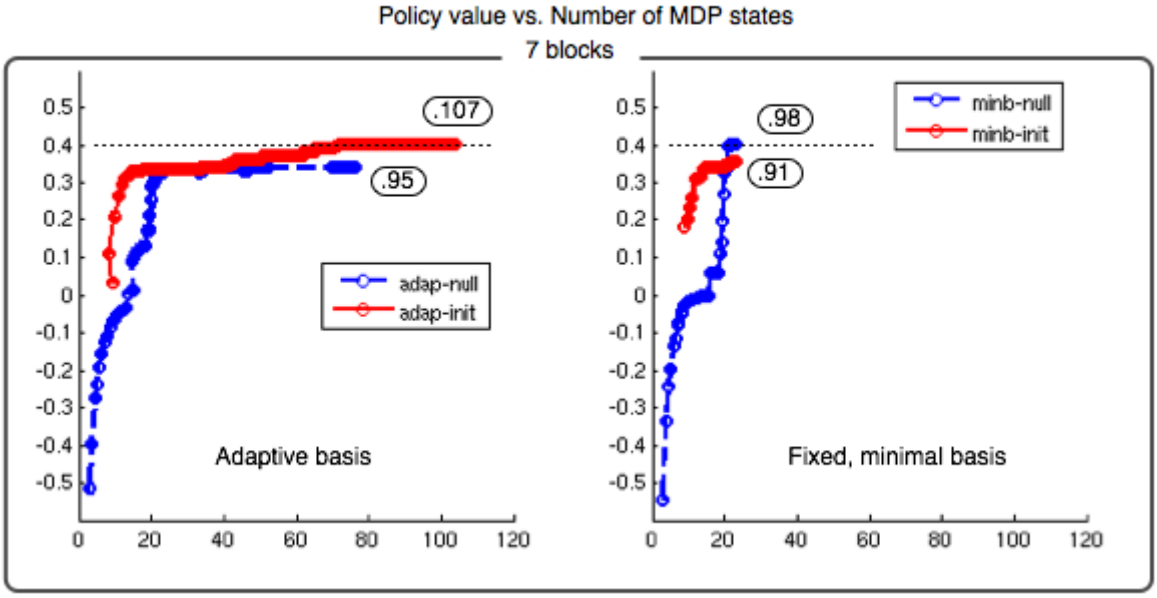


Figure 6-10: A plot of expected value vs. number of states in the MDP in the 7-block instance of the slippery blocks world domain. The dotted line is provided for reference across the two graphs. The average reward-per-step accrued by each algorithm is encircled near the corresponding curve.

Figure 6-11 contains a plot of value vs. number of MDP states for the adaptive and the full (fixed) basis algorithms in the 7- and 50-block slippery blocks worlds. The observation here is that the envelope expansion benefits greatly from the compaction of the state space resulting from the smaller basis. The `fulb-null` algorithm, which uses the complete representation (that is, it distinguishes all the colors of the blocks) is unable to get off the ground in the 7-block domain; while the `adap-null` algorithm follows closely behind `adap-init`. In the much larger domain, `adap-null` is still able to produce a reasonable model, while `fulb-null` produces nothing. [4]

Figure 6-12 shows a plot of value vs. number of MDP states for the all algorithms in the $b1$ instance of the MadRTS world. The PPDDL description of this problem is in Figure A-16 on page 131. Of note here is simply that, in general, the adaptive-basis algorithms are able to provide the highest expected value for a given model size.

[4]When no data is shown in a graph, it is because the the preceding experiments in the suite — that is, the smaller ones — had already exceeded the time-limit, which was approximately two hours for any given problem instance.

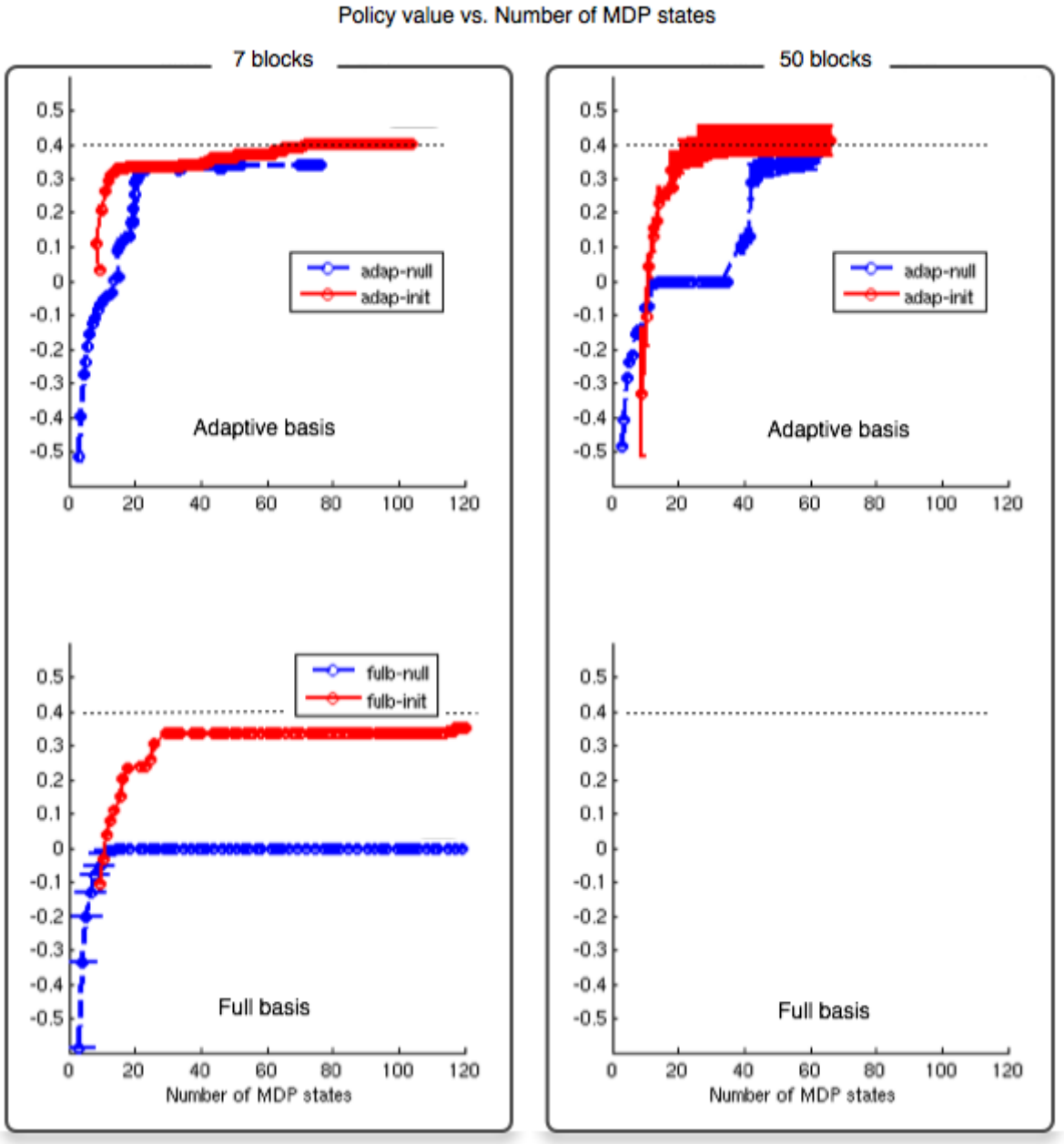


Figure 6-11: A plot of expected value vs. number of states in the MDP in the 7- and 50-block instances of the slippery blocks world domain. The dotted line is provided for reference across the graphs.

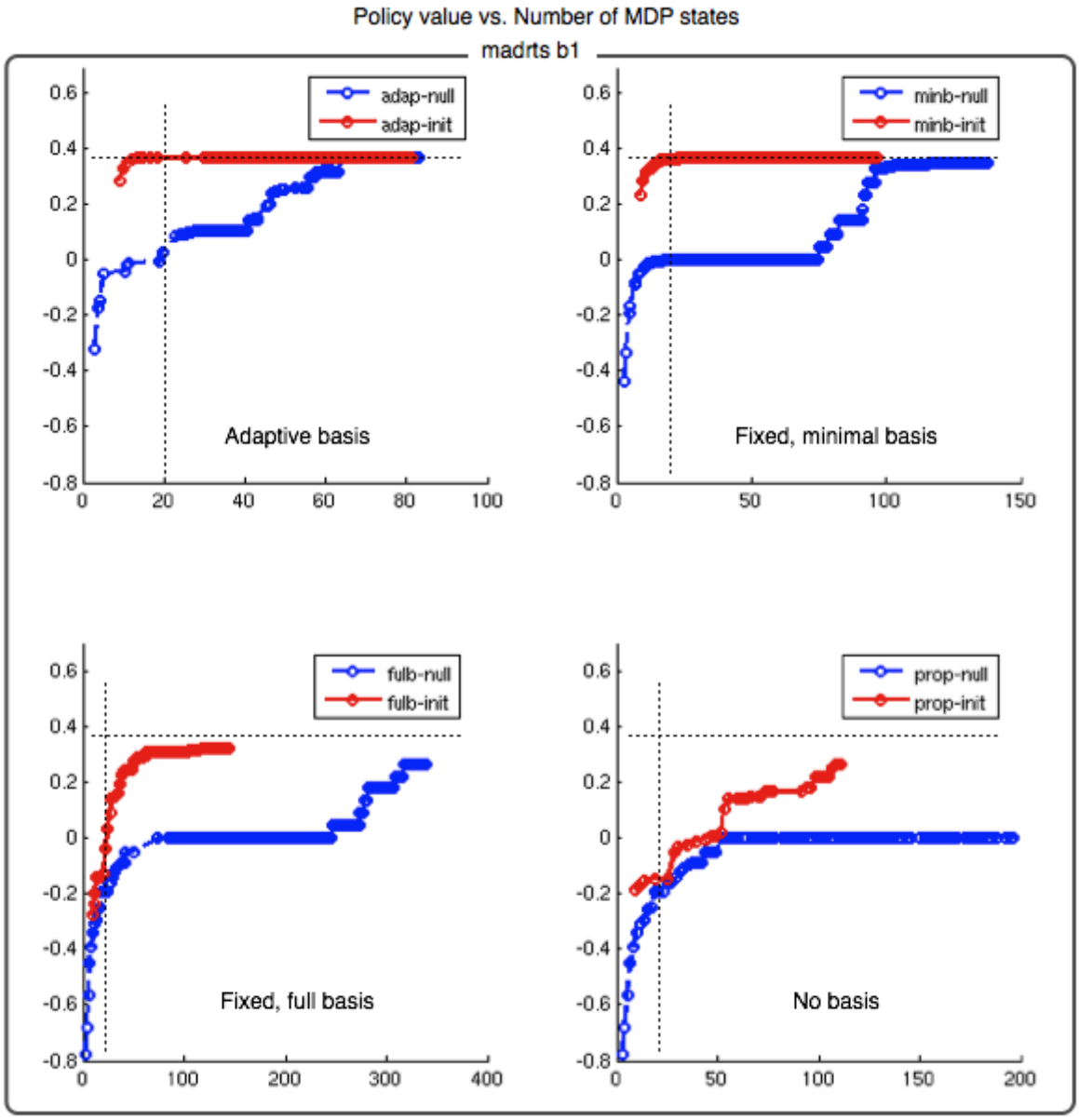


Figure 6-12: A plot of expected value vs. number of states in the MDP in the $b1$ instance of the MadRTS domain. The crossed dotted lines provide an invariant reference point across all the graphs.

The essential message that can be draw from these experiments consists of these three points:

1. Adjusting basis representation can yield more accurate model.

2. Equivalence classes help in envelope expansion.

3. Finding minimal basis representation, in conjunction with an initial plan, produces the highest value per number of states in MDP.

In general, better policies are found when gradually elaborating an initial solution than are found by trying to solve a problem all at once. The equivalence classes further aid this elaboration because they constrain the sampling done in the envelope MDP during envelope expansion.

Chapter 7

Conclusions and future directions

In this thesis, we have described a formalism for planning with equivalence classes of objects which is dynamic, domain-indpendent, and works under arbitrarily complex relational structure. We have shown the results of some experiments that demonstrate efficiency gains as problems become large in several domains.

Furthermore, we have used this idea to bootstrap the solution of planning problems in uncertain domains by implementing envelope-based planning as an interval MDP. We have also presented some experiments that show the advantage of this anytime approach to refinement of policy.

However, since the work described in this thesis is an initial step towards planners of this kind, there are many ways in which the approach could be improved.

7.1 Improving the planning

As our implementation currently stands, the biggest bottleneck in the planning algorithm, which is implemented as a best-first search with random tie-breaking, is the heuristic evaluation of states. This is because we have adapted the FF heuristic to be an admissible heuristic, which ultimately yields solutions of optimal length, but which involves searching the relaxed plan graph for the *shortest* relaxed plan. As a result, depending on the order of the search, this is a potentially exponential operation. We discuss these ramifications next.

7.1.1 Impact of action commutativity

In the logistics-style domains, there is a greater potential for actions that could be executed in parallel, or, irrespective of order. For example, in a sequential plan, it may not matter which we do first: either hoist a crate at the first depot, or move a truck towards the depot. While our equivalence-class analysis eliminates such permutations among objects of the same class (and thereby realizes considerable efficiency gains), it does not do so for structurally distinct objects, nor does it eliminate permutations in order among parallelizable action sequences. As a result, the heuristic computation in these domains, because it seeks to optimize for the shortest path, becomes more costly.

This issue is discussed by Haslum and Geffner [33] and Korf [43], who suggest a solution based on imposing a fixed ordering on such actions. If the REBP approach is to be extended efficiently into a greater variety of domains, this is one issue that must be addressed.

7.1.2 Other admissible heuristics

There are other admissible heuristics that may be more efficient to compute than our adaptation of the FF heuristic. For instance, Haslum and Geffner describe a family of heuristics that trades off informativeness with efficiency of computation [33]. Edelkamp [14] surveys this approach and others but finds that, in heuristic search planning, the heuristics are either not admissible, or, admissible but too weak. Edelkamp proposes by contrast an approach based on pattern databases [14, 43]. Pattern databases are pre-computed tables of distances between abstractions of states. Edelkamp's approach is appealing in that it finds optimal plans if possible, and approximates the optimal solution in more challenging planning problems (in propositional, deterministic settings); however, space consumption grows rapidly — for example, searching for solutions in benchmark blocks-world domains of more than 13 blocks grinds to a halt the various optimal, general planners studied.

Thus, how to improve the efficiency of the heuristic for a larger variety of domains

while preserving its informativeness is a key open question. It is also important to recognize that no one heuristic is best suited for all types of domains — some may be more effective in logistics-style domains, others for puzzle-style domains, and so on.

7.1.3 Considering non-optimal planning

The impact of REBP seems largest in the area of optimal planning, in which the shortest solution must be found, since REBP provides a way to reduce branching factor without losing optimality.

However, in Edelkamp's study [14], as well as in our own experience and that of other researchers, FF's approach to finding approximate (non-optimal) solutions via hill-climbing is a time-effective approach in large problem instances. The experiments presented in this work were constructed to illustrate the properties of our algorithm specifically when planning difficulty is a function of increased problem size and not of increased solution length; but, there are vast numbers of planning problems out there who are not necessarily guaranteed to scale in this way.

One way to move in this direction may well be to use object equivalence classes in conjunction with the non-admissible FF heuristic. This would produce a plan faster, but it could be a longer solution. However, after discovering this initial solution, REBP can then invoke the anytime envelope expansion phase, which may proceed to discover a shorter solution given more computational resources. This is a tactic we have not yet explored.

7.2 More aggressive approximations

The idea behind minimizing the set of predicates used to represent the planning problem was to force more objects into the same equivalence class and, thus, approximate the original planning problem with a smaller one. This approach turned out to be effective in our experiments, but it is only a rudimentary start based on a simple syntactic analysis of the goal sentence and action preconditions. It may be profitable to investigate other ways of calculating approximate representations. Learning may

play an important role here, as discussed in a later section.

Furthermore, as problems increase in size and complexity, it will be necessary to consider approximations to the isomorphism-based equivalence we have developed for objects. For example, approximate graph isomorphism is an idea investigated in a recent paper by Fox and Long [22]. Additionally, at the risk of including an even more complex problem into the inner loop of our algorithm, it may be possible to find a more generalizable notion of equivalence in considering isomorphism or approximate isomorphism over *substructures* of graphs rather than whole graphs. Finally, one might consider other types of distance metrics on graphs, such as kernel functions of structured data [56, 35, 23, 24]. While a distance function on states might put more actions into the same equivalence class, it is less obvious how to use it to produce object equivalence classes, or whether it would be compatible with the incremental computation of object equivalence classes we have described.

7.3 Improving the envelope expansion

In the original paper on the Plexus algorithm [13], Dean *et al.* describe various techniques for estimating the value of incorporating a new state into the envelope MDP. If a candidate state is not expected to improve the expected value much, then it is not added to the envelope. Our implementation is simpler: all candidate states are accepted during envelope expansion. Thus, there is nothing to stop the envelope from growing, in the limit, to the size of the abstract state space. Obviously, this is a concern in large domains, and it would be beneficial to incorporate some kind of value analysis to this step.

Furthermore, now that we are dealing with relational domains — instead of the atomic-state domains of the original Plexus algorithm — it may be that a factored or hierarchical approach to the the envelope and basis expansion would be worth considering. For example, it may make little sense to search for refinements between trucks, routes, and crate-contents all at the same time, as is currently done.

The basis expansion, in particular, may benefit from a sensitivity analysis. In

the current implementation, we base our search for the refinement of the predicate set based on the width of the transition probability intervals. However, if this wide interval is in a part of the state space with relatively low risk or reward, why bother refining it? It would be more worthwhile to refine instead those intervals with the greatest impact on the value estimate.

Finally, there is the question of how to most efficiently compute the transition probability intervals. We have chosen a method based on sampling from the underlying state space, which, while having the advantage of simplicity and fidelity to the state space in question, comes with a computational cost. It may be worth considering computing these intervals analytically, by syntactic analysis of the rule schemas.

7.4 The role of learning

Learning is entirely absent from REBP, but there are many aspects which might benefit from it. For example, the idea of reducing the number of actions under consideration by minimizing the basis predicate set brings to mind the idea of *affordances* [25]; that is, the types of actions that are possible to effect on an object. There is the idea that people are able to restrict the number of affordances they consider for an object as a function of their motives [2]. Would it be possible to learn to "see" objects as equivalent, given the role that they play in eventually achieving a goal? This may yield more adaptive approaches than our current idea of computing a minimal basis set of predicates before beginning to plan.

7.5 Completeness, correctness, convergence, and complexity

Finally, what can we say about the computational characteristics of the full REBP algorithm?

We have theoretical guarantees on the planning side as long as we use the full predicate set. What happens when we use a reduced predicate set? The computation

of the reduced predicate set simply guarantees that we will be able to solve a relaxed plan, but it guarantees nothing about being able to solve the non-relaxed version. A mechanism for dealing with possible failures of this type will need to be investigated.

This approximation of the state space also impacts the MDP side. What are the convergence properties for an abstract, interval envelope MDP? It seems plausible to expect so, but, we have not proved whether Dean and Givan's analysis for bounded-parameter MDPs [28] holds this case.

Also missing is a thorough complexity analysis of the algorithm. Because it is a rather straightforward implementation, it is almost assuredly not as efficient in its current state as it might be. On the planning side: the algorithm is, in the worst case, exponential in the branching factor. On the MDP side, the greatest bottleneck is in testing for state equivalence when doing the sampling to improve the transition probability interval estimates. As we mentioned in Section 5.3, computing isomorphisms is not a limiting bottleneck in practice. What we have observed is that, if an isomorphism is not present, the computation tends to fail quickly. Thus, we only seem to pay the full cost of the computation in the case where the cost is able to be offset by its long-term benefits. Being able to characterize and guarantee this observation, however, is important and open future work.

Nonetheless, it seems hard to avoid computational complexity when dealing with inherently hard problems such as planning, which is at least PSPACE-complete even in the propositional, deterministic case [9]. The best we can hope for is not to let unnecessary complexity get the better of us. The work described in this thesis is one small step towards that goal.

Appendix A

Results

Below, we show a representative selection of the full set of experiments. In general, the experiments were carried out as follows. For the algorithms that required an initial plan, a corresponding MDP was initialized with the output of the planning system. Then, 150 rounds of deliberation were done. For the algorithms that required no initial plan, an MDP was initialized with a single state, the initial state as given by the planning problem description. Then, 200 rounds of deliberation were done. In some of the more memory-intensive domains, the number of deliberation rounds was reduced to 60. At the end of each deliberation round, value iteration was used to compute policies, with a discount factor of 0.9. Finally, to compute the accumulated reward, we ran 900 steps of simulation in each domain at the end of the deliberation rounds. This corresponds to about 8 successful trials in the blocks worlds. Actions were selected according to the policy, with random action selection occurring %15 of the time. This was to force the policy to react to an unexpected state. In the interval MDPs, action selection is done by choosing the action with the highest *average* value. A reward of 1.0 was given upon attainment to the goal, and we report the average accumulated reward per step.

We plot the results in two ways. We plot expected value (taken as the expected value of state q_0 in the MDP, or the average of the interval, in the case of an interval MDP) vs. the number of states in the MDP, as a way of showing the value of the policy as a function of the size of the model. In conjunction with each such graph, we also

plot the same expected value as a function of the computation time, as measured by a CPU-cycle monitoring package. In the experiments which required no computation of a plan first, time is measured from the beginning of the construction of the initial MDP; for those that did, the first data point is plotted after construction of the MDP, plus the amount of time spent planning. Error bars denote mean squared error above and below each data point.

A.1 Blocks world

This domain is simply intended as a control, as there is no difference expected between the fixed, minimal basis and the adaptive one. The results show that there is not significant overhead to the adaptive approach in this domain. The PPDDL description of this domain was given in Figure 1-2 on page 23.

```
(define (problem blocks-problem)
 (:domain blocks-domain)
 (:objects block0 block1 block2 block3 block4 block5 block8  - block)
 (:init
  (on-top-of block0 block3)
  (on-top-of block1 block2)
  (on-top-of block2 table)
  (on-top-of block3 table)
  (on-top-of block4 table)
  (is-green block0)
  (is-blue block1)
  (is-green block2)
  (is-blue block3)
  (is-red block4)
  (on-top-of block5 block8)
  (on-top-of block8 table)
  (is-green block5)
  (is-blue block8)
 )
 (:goal
  (and
   (exists (?fb0 - block)
    (and
   (exists (?fb1 - block)
    (and
     (not (= ?fb0 ?fb1))
     (on-top-of ?fb0 ?fb1)
   (exists (?fb2 - block)
    (and
     (not (= ?fb0 ?fb2))
     (not (= ?fb1 ?fb2))
     (on-top-of ?fb1 ?fb2)
   (exists (?fb3 - block)
    (and
     (not (= ?fb0 ?fb3))
     (not (= ?fb1 ?fb3))
     (not (= ?fb2 ?fb3))
     (on-top-of ?fb2 ?fb3)
   (exists (?fb4 - block)
    (and
     (not (= ?fb0 ?fb4))
     (not (= ?fb1 ?fb4))
     (not (= ?fb2 ?fb4))
     (not (= ?fb3 ?fb4))
     (on-top-of ?fb3 ?fb4)
     (on-top-of ?fb4 table)
))))))))))
  )
 )
)
```

Figure A-1: Blocksworld: sample PPDDL problem description, 7-block world. The same problem instances are used in the standard blocksworld, slippery blocksworld, and zoom blocksworld.

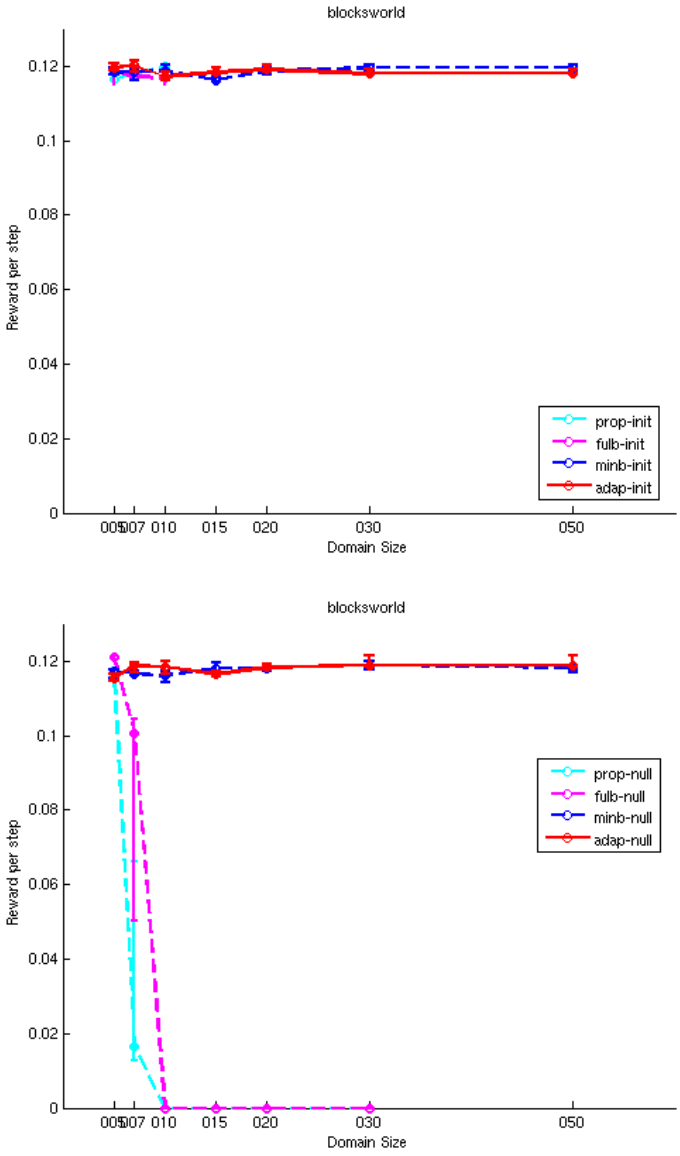


Figure A-2: Blocksworld: average reward per step in all problems. After 7 blocks, the approaches that do not minimize the basis have trouble: the ones which require a plan run out of memory or exceed the time limit, and those which sample the MDP space produce poor policies. The maximum score possible is 0.12.

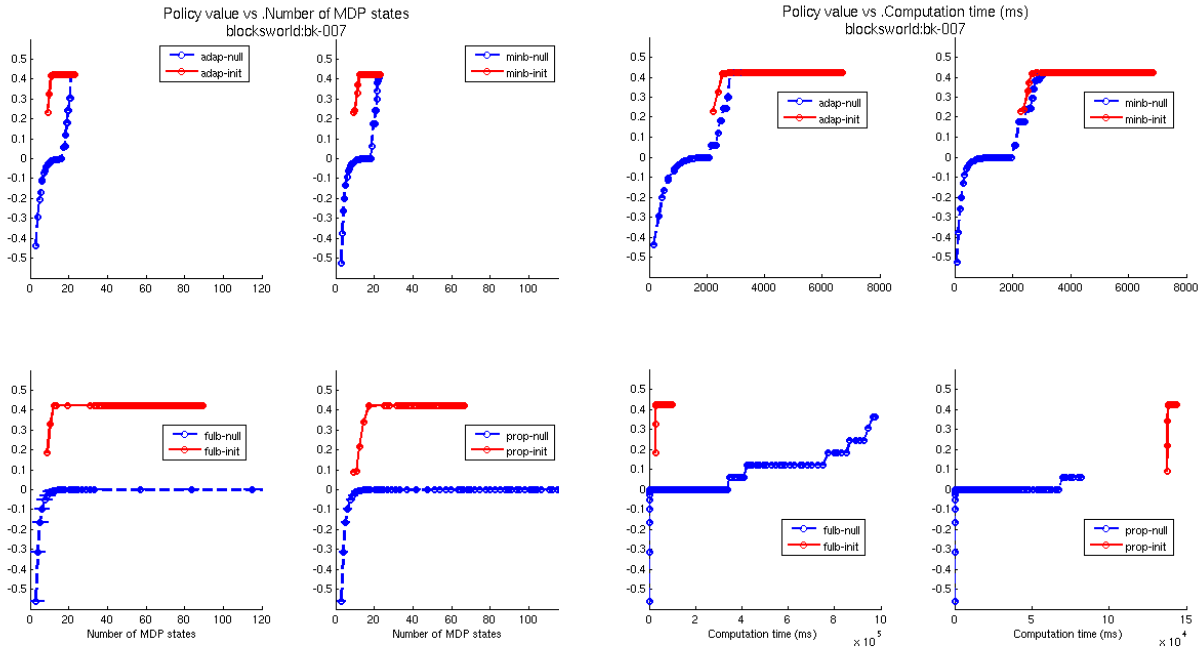


Figure A-3: Blocksworld: plot of expected value in the 7-block domain. Minimizing the basis produces good policies in less time.

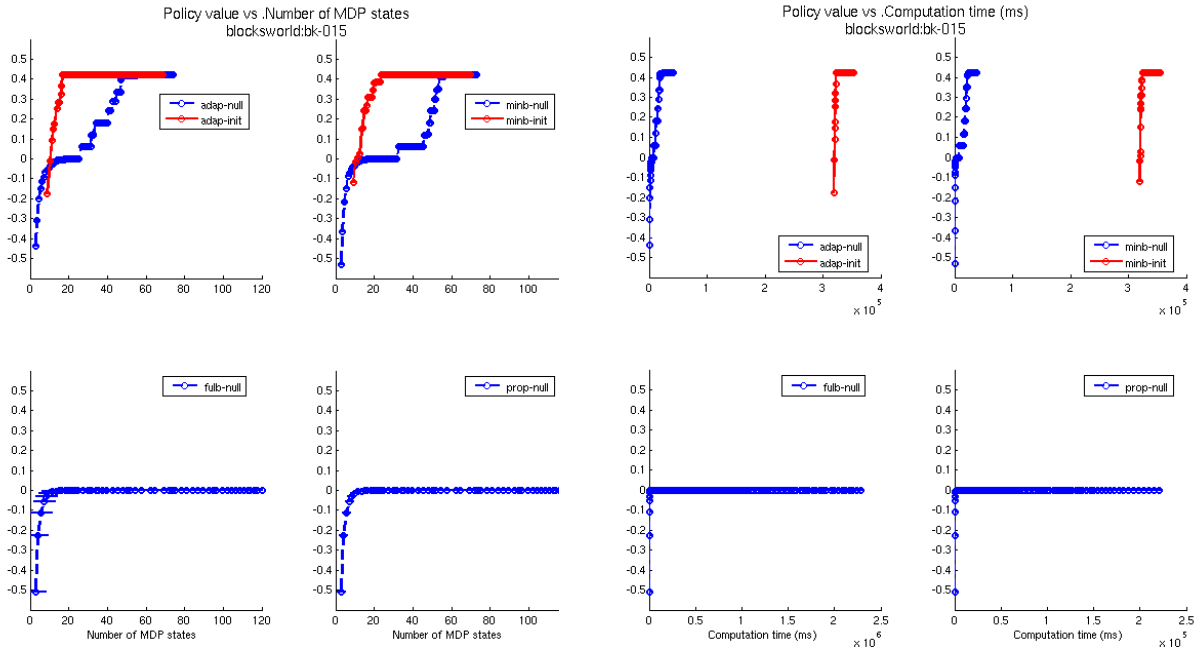


Figure A-4: Blocksworld: plot of expected value in the 15-block domain. Planning takes too long in the full-basis and propositional settings.

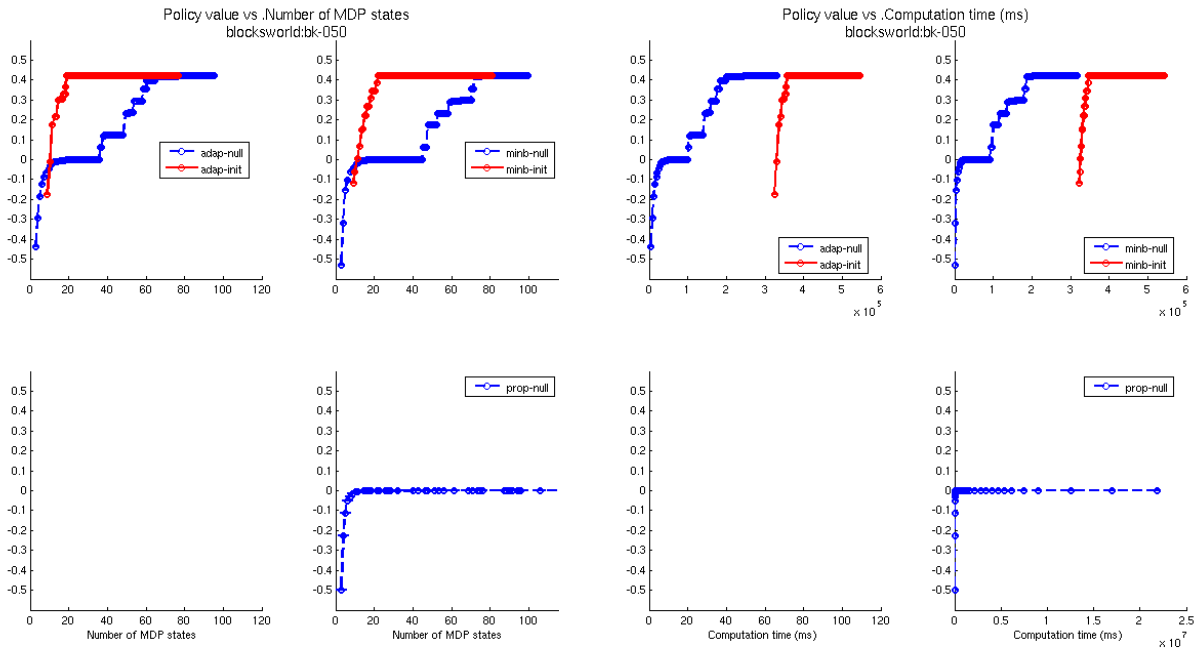


Figure A-5: Blocksworld: plot of expected value in the 50-block domain. Computing a plan first produces MDPs with higher expected value for a given model size.

A.2 Slippery blocks world

This is the slippery blocks domain. In general, being able to adapt the basis to detect the green predicate resulted in high-valued policies that also corresponded with good execution behavior. By contrast, being forced to look for a solution in the full-size basis resulted in higher computation and memory costs. The PPDDL description of this domain was given in Figure 6-4 on page 96.

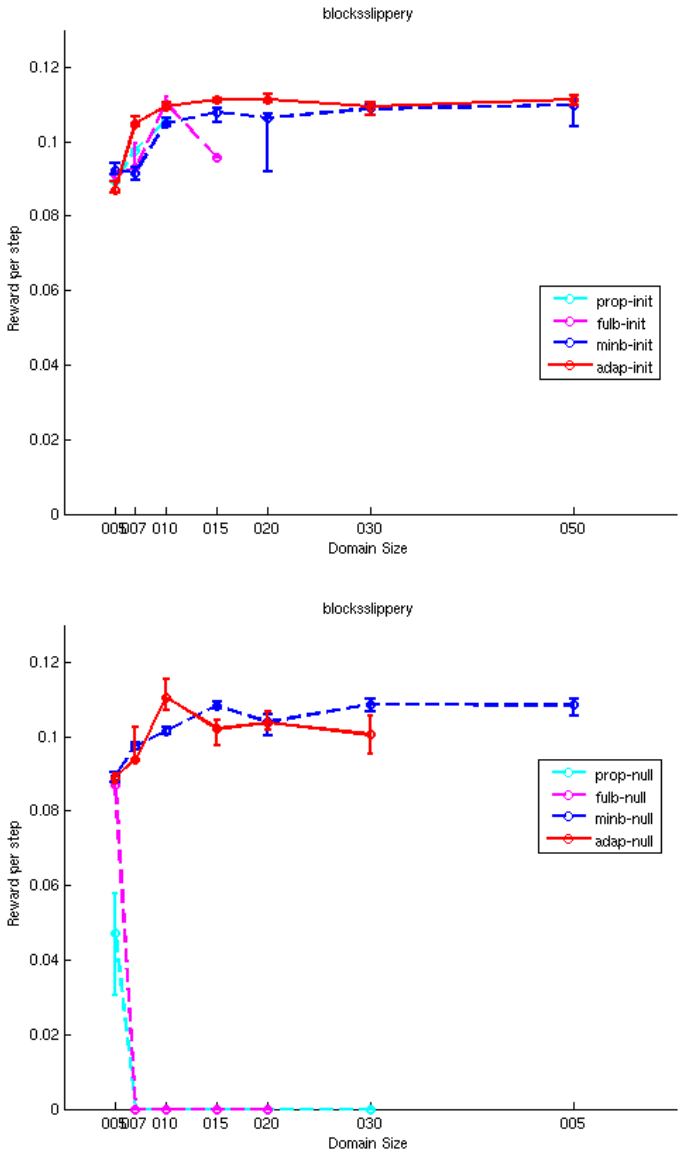


Figure A-6: Slippery blocksworld: average reward per step in all problems. The maximum score possible is 0.12.

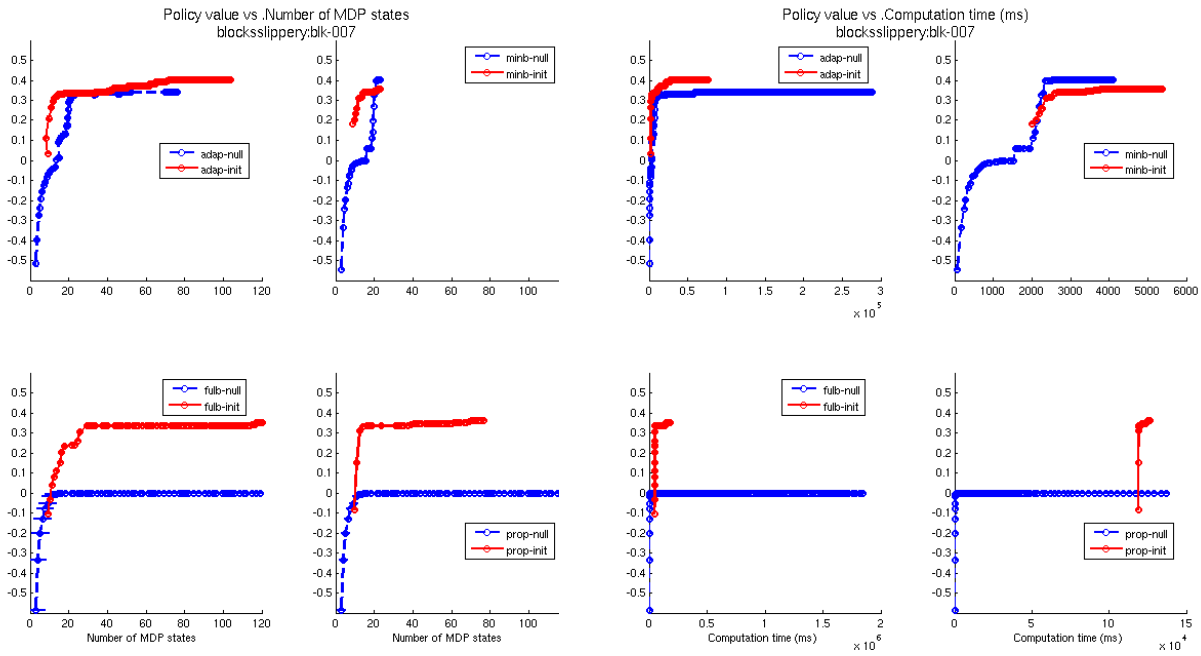


Figure A-7: Slippery blocksworld: 7 blocks. The fixed, minimal basis is fastest; but, it does not achieve as high a reward during execution as the adaptive-basis approach.

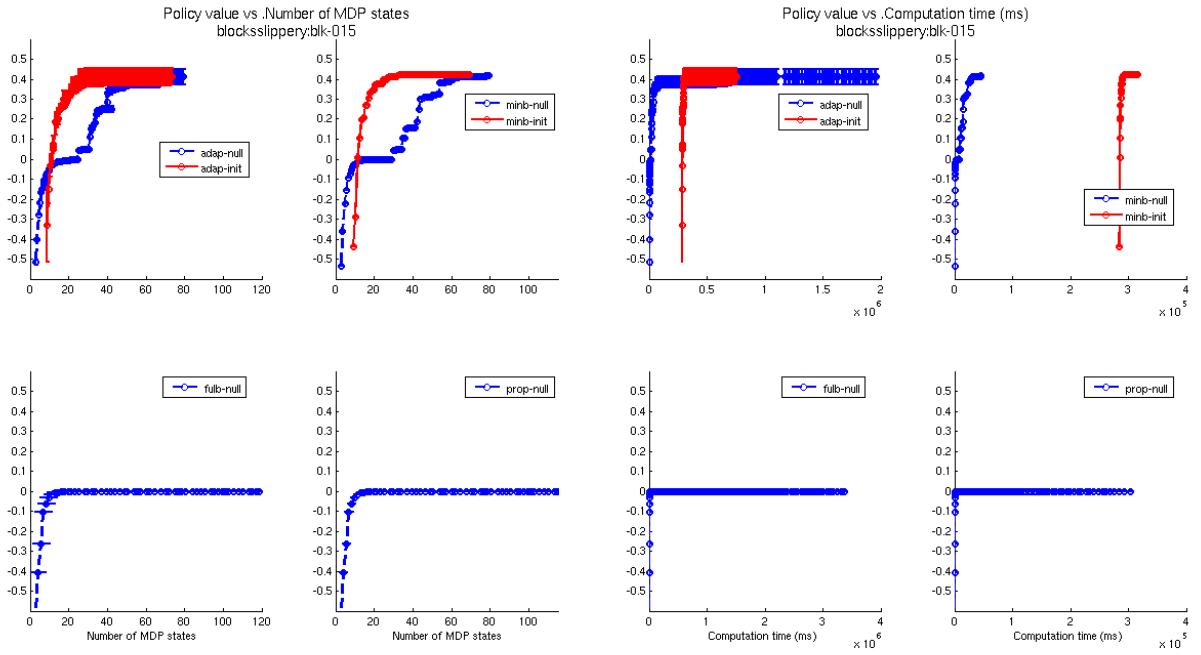


Figure A-8: Slippery blocksworld: 15 blocks. Again, the adaptive basis approach takes more computation time, but it is able to represent the interval of expected value and produces higher reward during execution.

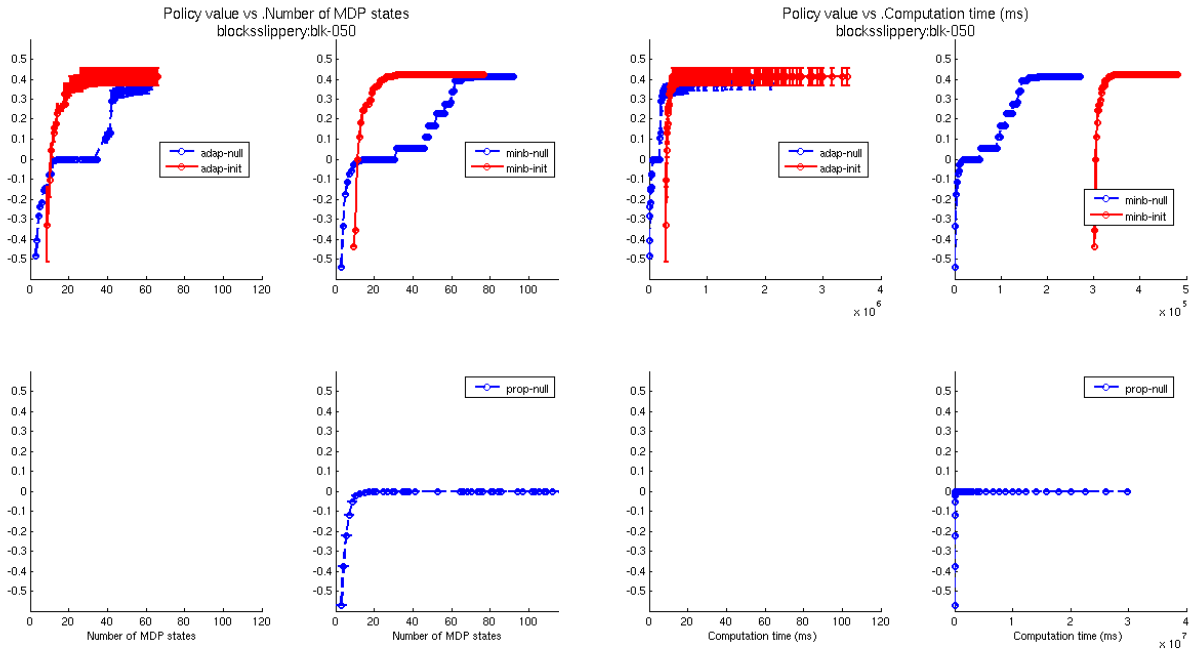


Figure A-9: Slippery blocksworld: 50 blocks. In the biggest domain, the adaptive-basis approach is able to model the range of expected values better.

A.3 Zoom blocks world

The PPDDL description of this domain can be seen in Figure A-10. This ended up being a trickier domain than anticipated. While a policy consisting of "zoom" actions reaches the goal faster, we expected to see the adaptive-basis approaches switch over to using the more reliable pick-up and put-down actions as they discovered the *holding* predicate. While this is indeed what happens, as evidenced by the higher rewards seen in Figure A-11, it ends up requiring considerable envelope exploration to uncover the new policy. More directed envelope exploration may be able to address this.

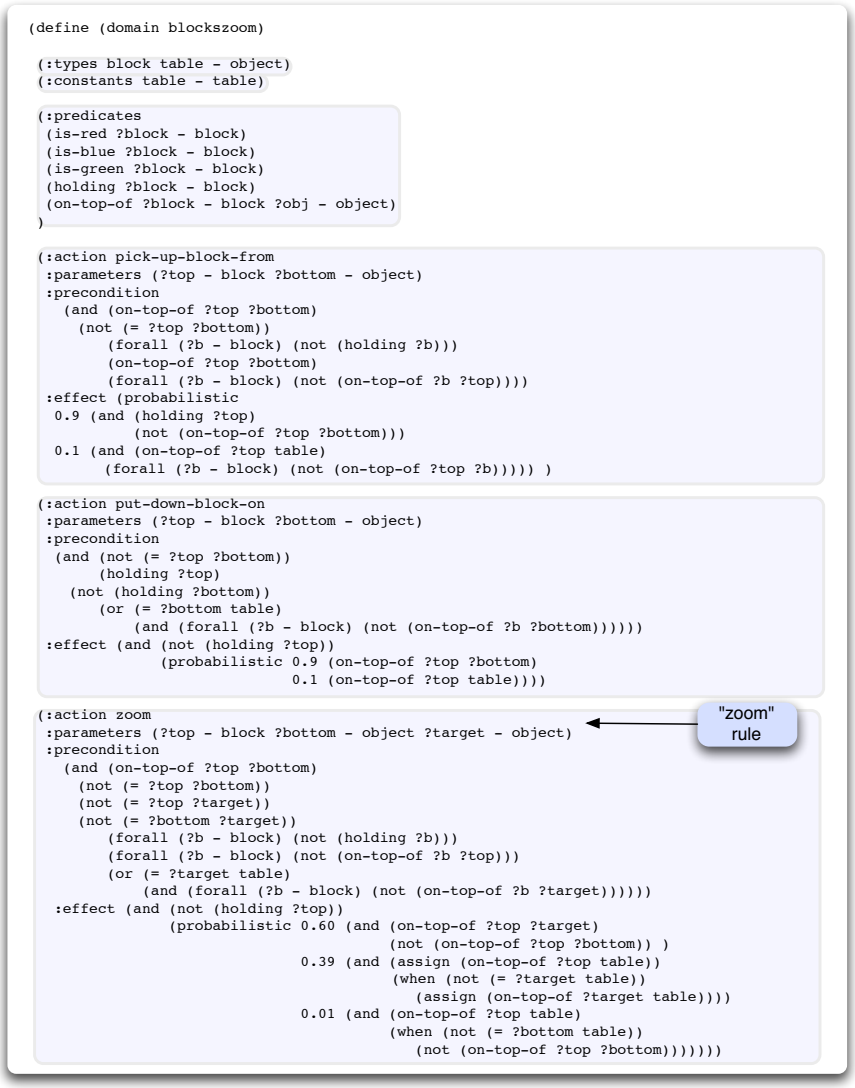


Figure A-10: Zoom blocksworld: PPDDL domain description.

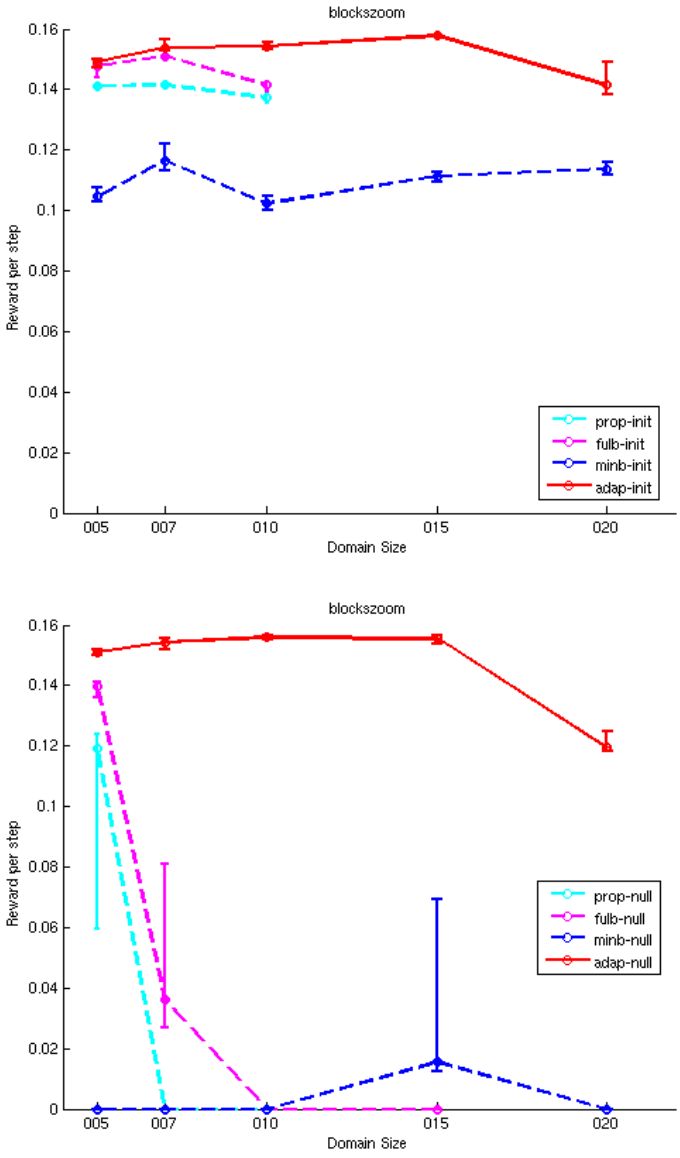


Figure A-11: Zoom blocksworld: average reward per step in all problems. The adaptive basis is able to discover a more rewarding policy.

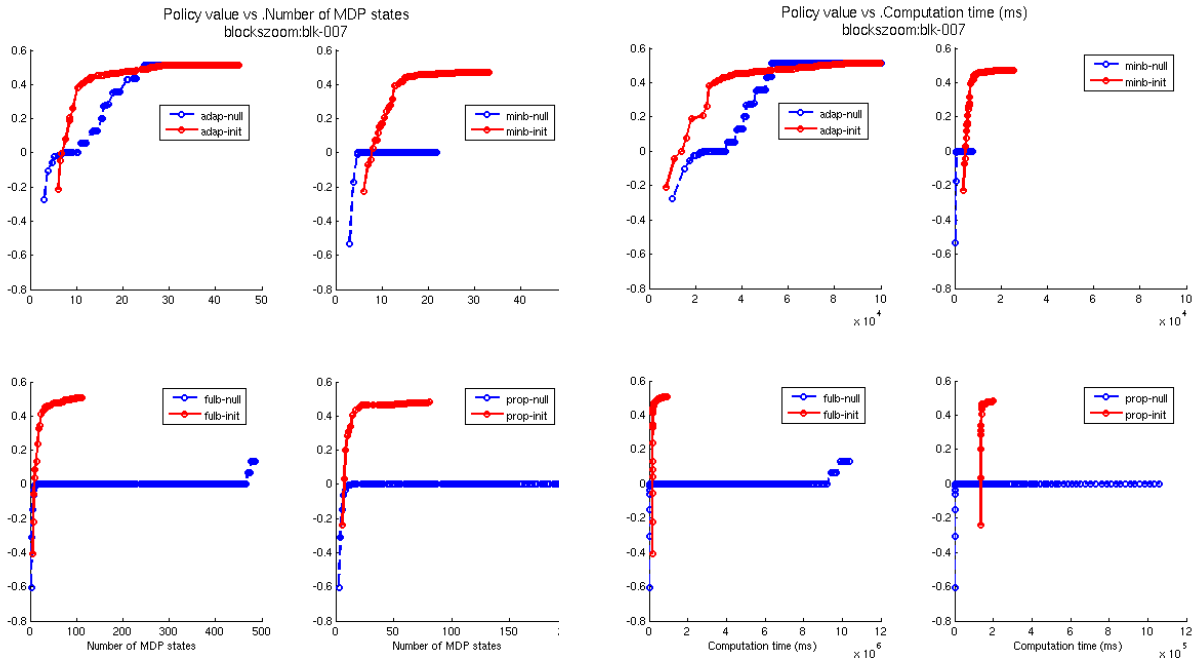


Figure A-12: Zoom blocksworld: 7 blocks.

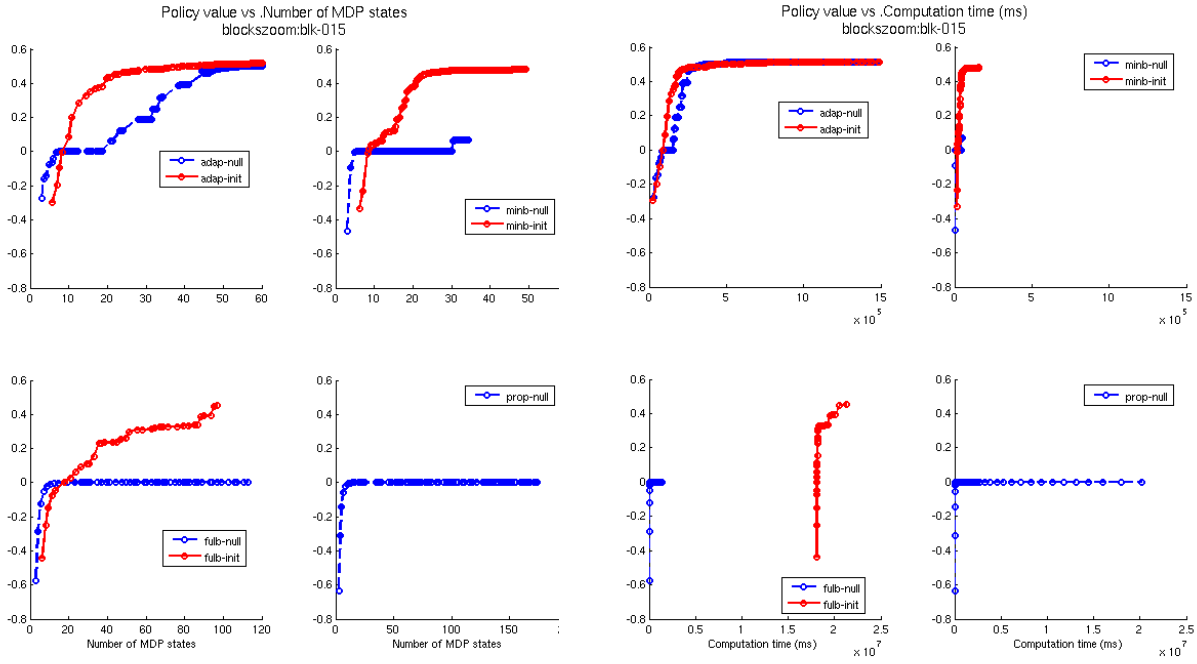


Figure A-13: Zoom blocksworld: 15 blocks.

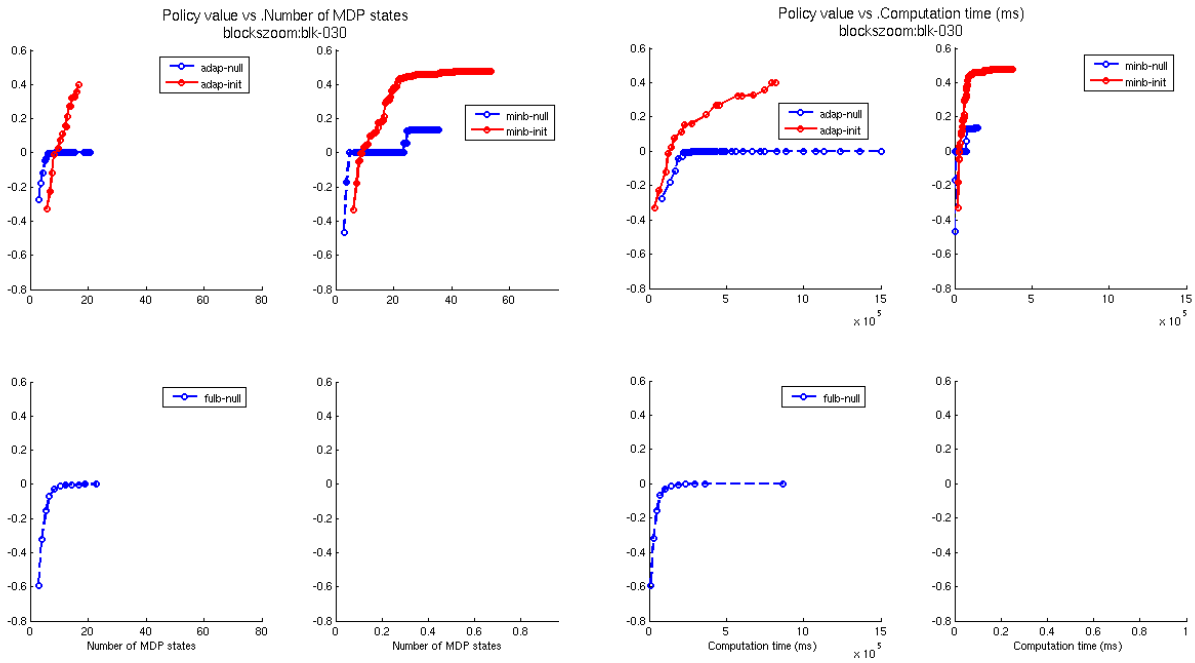


Figure A-14: Zoom blocksworld: 30 blocks. While high values are achieved, this larger problem instance exerts a greater computational burden on the adaptive-basis approaches than on the fixed, minimal-basis ones.

A.4 MadRTS

```
(define (domain maddomain)

 (:types  territory movable - object
       resource unit enemy squad - movable
       food - resource)

 (:predicates (at ?u - movable ?t - territory)
          (adj ?start - territory ?end - territory)
          (member ?u - unit ?squad - squad)
          (htop ?u - unit)
          (has ?u - unit ?r - resource)
 )

 (:action move
       :parameters (?u - unit ?terrOld - territory ?terrNew - territory)
       :precondition (and  (or (adj ?terrOld ?terrNew) (adj ?terrNew ?terrOld))
                   (at ?u ?terrOld))
       :effect
        (and (when (htop  ?u)
                    (probabilistic .99 (and (at ?u ?terrNew)
                                            (not (at ?u ?terrOld))
                                            (probabilistic .7 (not (htop  ?u))))))
             (probabilistic .51 (and (at ?u ?terrNew)
                                     (not (at ?u ?terrOld))
                                     (not (htop ?u))))
             )
 )

 (:action collect
    :parameters (?u - unit ?terr - territory ?f - resource)
    :precondition (and (at ?u ?terr) (at ?f ?terr))
    :effect (and (has ?u ?f) (not (at ?f ?terr)) )
 )

 (:action use
    :parameters (?u - unit ?f - resource)
    :precondition  (has ?u ?f)
    :effect (and (htop ?u))
 )

 (:action transfer
    :parameters (?u1 - unit ?u2 - unit ?f - resource)
    :precondition (has ?u1 ?f)
    :effect (and (not (has ?u1 ?f)) (has ?u2 ?f))
 )
)
```

Figure A-15: MadRTS: PPDDL domain description.

The general PPDDL description of this domain is in Figure A-15, and a sample domain description is in Figure A-16. These were challenging problems for all algorithms, likely due in part to the problem of *action commutativity* discussed in Section 7.1.1.

A.4.1 The b world

Schematics of the three problem instances are given in Figure A-17. These were challenging problems. Policies of slightly higher value are eventually found with the adaptive basis, but this produces no appreciable increase in the amount of accrued reward. In general, the adaptive basis with initial plan and the minimal basis with initial plan seem to perform about the same. A more sophisticated, or directed,

```
(define (problem madrts)
(:domain maddomain)
(:objects
  t3 t8 t9 t13 t12 t18 - territory
  squad1 - squad
  u1 u2 u3 u4 - unit
  f1 - food
  e1 - enemy)
(:init
 (adj t3 t8)
 (adj t3 t9)
 (adj t8 t13)
 (adj t9 t13)
 (adj t8 t12)
 (adj t12 t18)
 (adj t13 t18)
 (at u1 t3)
 (at u2 t3)
 (at u3 t3)
 (at u4 t3)
 (htop u1)
 (htop u2)
 (htop u3)
 (htop u4)
 (member u1 squad1)
 (member u2 squad1)
 (member u3 squad1)
 (member u4 squad1)
 (at f1 t9)
 (at e1 t18)
)
(:goal
  (forall (?e - enemy)
     (and    (at ?e ?loc)
        (exists (?u1 - unit)
             (exists (?u2 - unit) (and (not (= ?u1 ?u2))
                                  (at ?u1 ?loc)
                                  (at ?u2 ?loc))))))
 )
)
```

Figure A-16: MadRTS: sample PPDDL problem description, $b1$ world.

way of exploring the fringe may be needed to bring about a change in policy: even though the adaptive-basis approach can represent a policy to explicitly seek out and consume food resources, the many action changes may place it quite "far" from the initial envelope and make it hard to discover through random exploration.

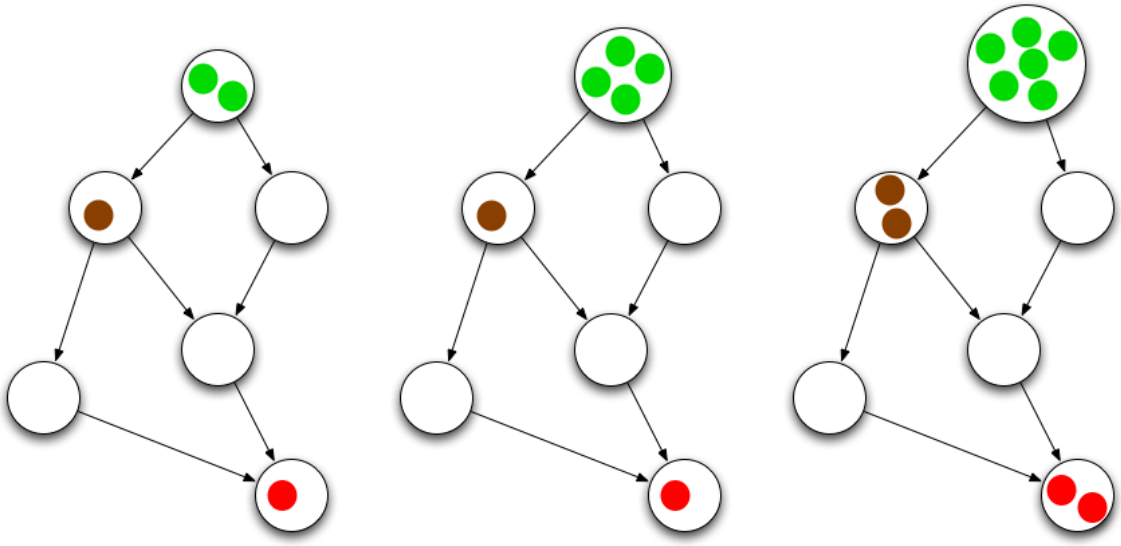


Figure A-17: MadRTS domain: schematics of the three b problems; $b0$ through $b2$, from left to right. In the first domain, there are two units (green), one food resource (brown) and one enemy (red). In the third domain, there are six units, two food resources, and two enemies.

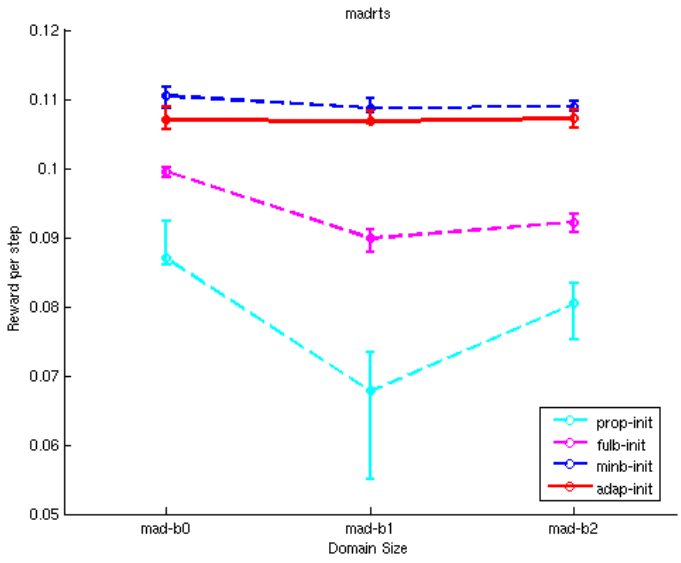


Figure A-18: MadRTS world b: average reward per step.

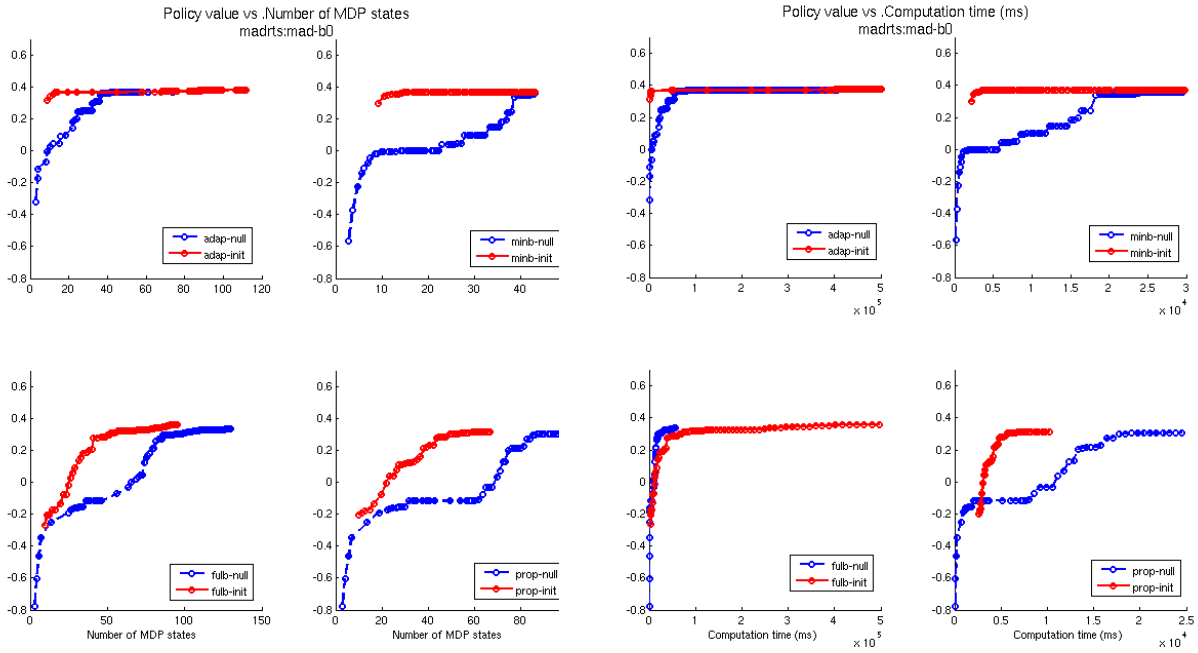


Figure A-19: MadRTS: expected value in world b0.

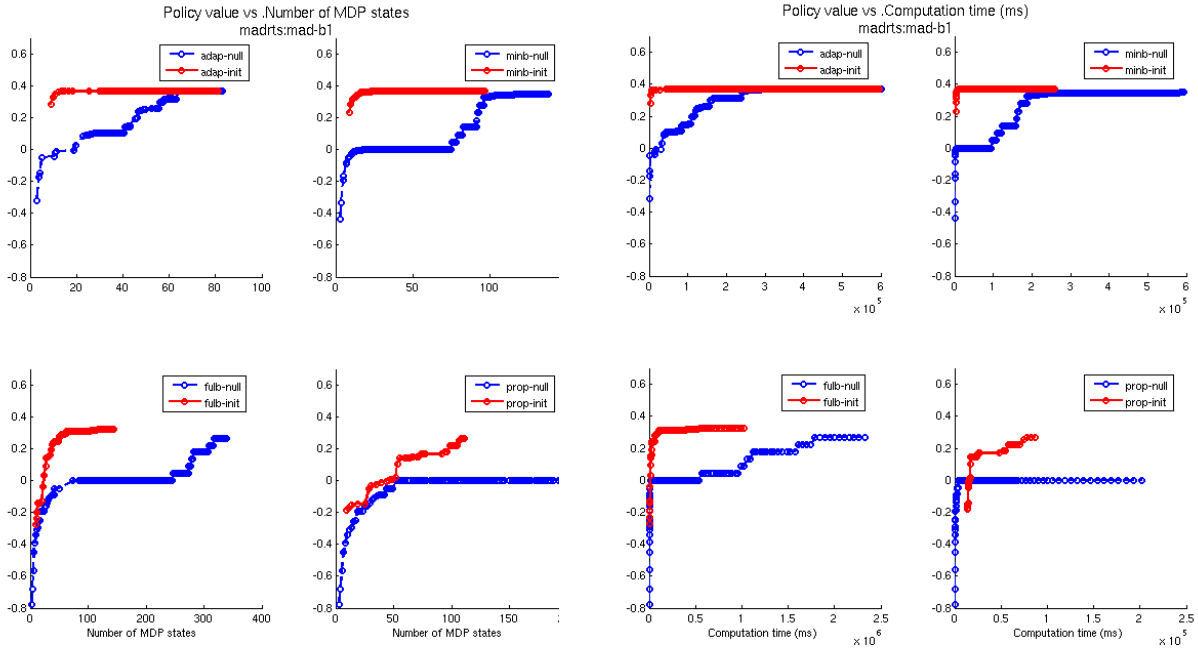


Figure A-20: MadRTS: expected value in world b1.

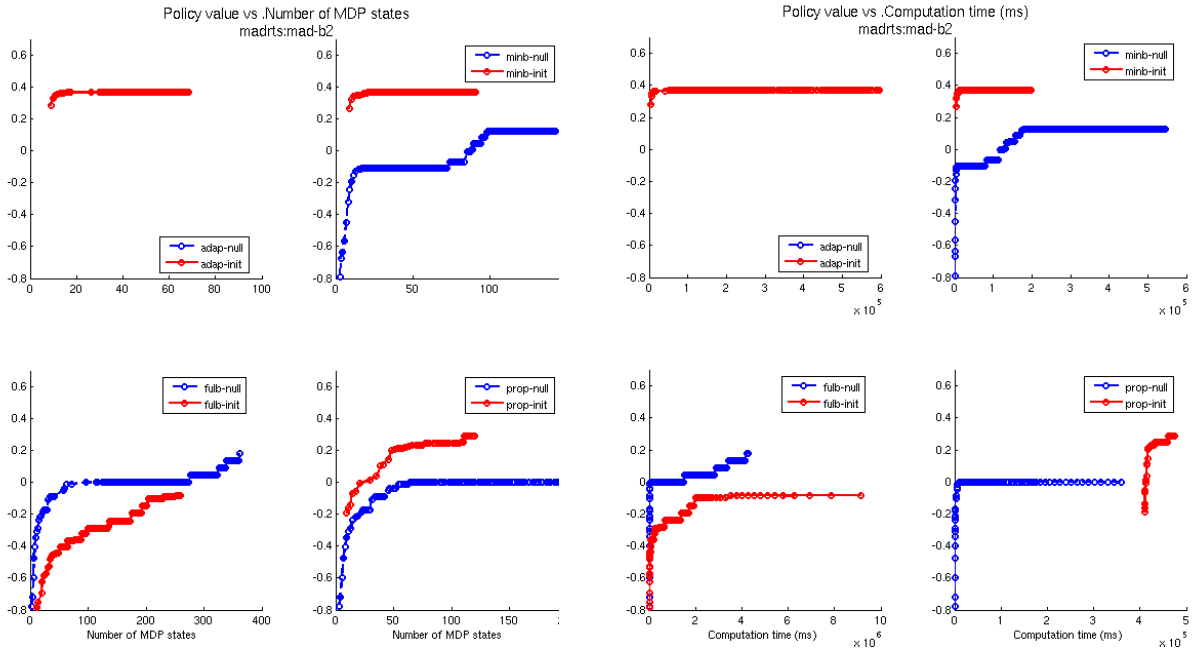


Figure A-21: MadRTS: expected value in world b2.

A.4.2 The c world

Schematics of the three problem instances are given in Figure A-22. As before, these were challenging problems. In the larger of the two instances, the `minb-init` method starts to run into memory problems, probably due to our implementation. The `adap-init` method does succeed in finding a good-valued policy, however.

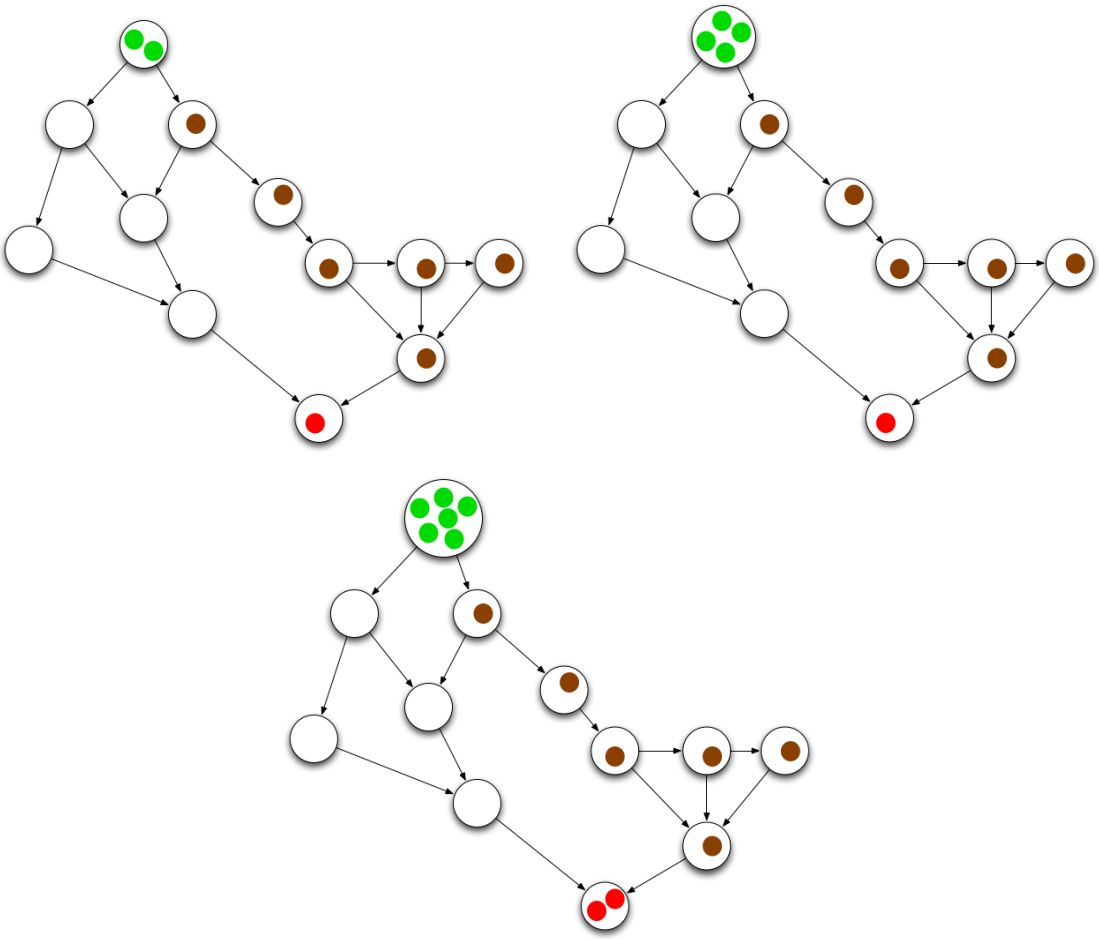


Figure A-22: MadRTS domain: schematics of the three c problems; $c0$ through $c2$, clockwise from top left.. The map is a replica of that given in the original Mad Doc proposal document; the placement of units, enemies, and food resources is our own. In the first domain, there are two units, one enemy, and a variety of food resources in one area of the map. In the third domain, there are six units and two enemies.

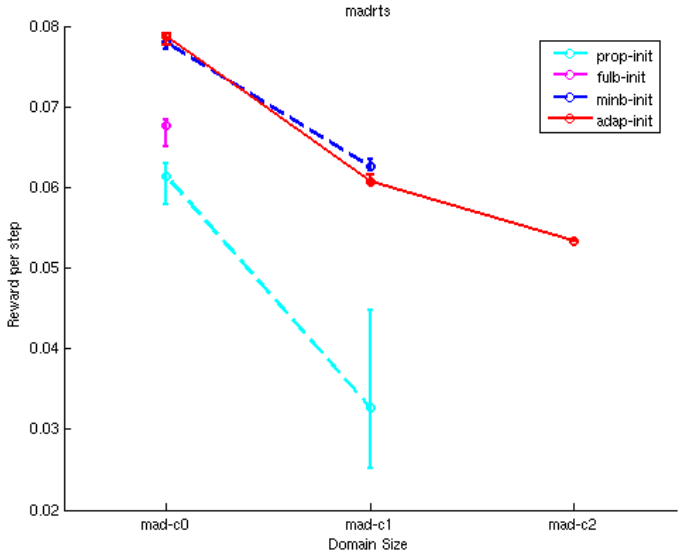


Figure A-23: MadRTS world *c*: average reward per step.

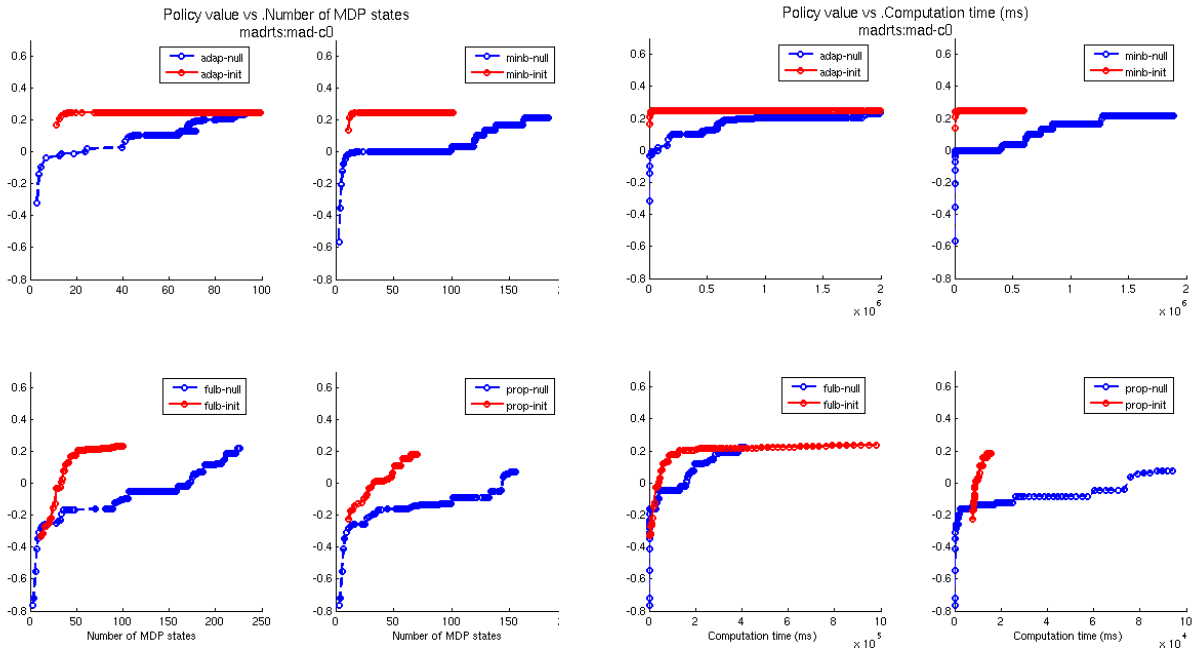


Figure A-24: MadRTS: expected value in world c0.

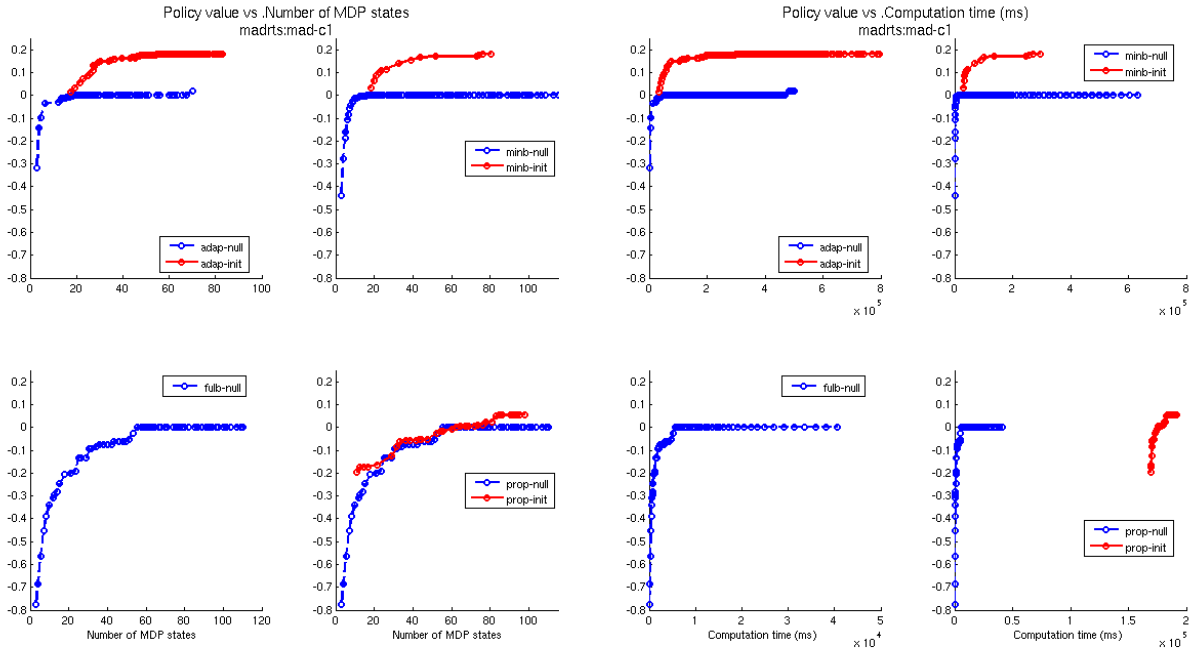


Figure A-25: MadRTS: expected value in world c1.

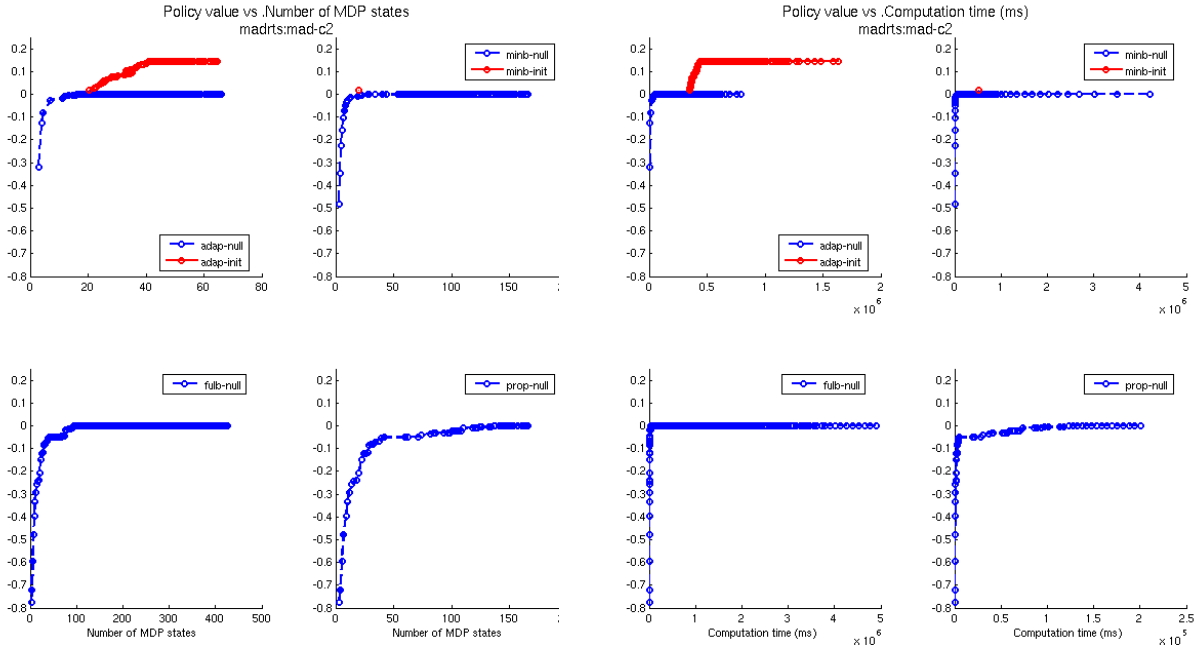


Figure A-26: MadRTS: expected value in world c2.

Bibliography

[1] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1), 1995.

[2] Gordon Bernedo-Schneider. Cognitive modeling of motivations for artificial agents. In E. Rome, P. Doherty, G. Dorffner, and J. Hertzberg, editors, *Towards Affordance-Based Robot Control*, number 06231 in Dagstuhl Seminar Proceedings, Abstracts Collection. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Dagstuhl, Germany, 2006.

[3] Avrim L. Blum and Merrick L. Furst. Fast plannning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[4] Avrim L. Blum and John C. Langford. Probabilistic planning in the graphplan framework. In *5th European Conference on Planning*, 1999.

[5] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence, Special issue on Heuristic Search*, 129, 2001.

[6] Blai Bonet and Hector Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *ICAPS-03*, 2003.

[7] Craig Boutilier and Richard Dearden. Using abstractions for decision-theoretic planning with time constraints. In *12th AAAI*, 1994.

[8] Craig Boutilier, Raymond Reiter, and Bob Price. Symbolic dynamic programming for first-order MDPs. In *17th International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

[9] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69, 1994.

[10] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *AAAI-88*, 1988.

[11] Thomas Dean and Robert Givan. Model minimization in Markov decision processes. In *AAAI*, 1997.

[12] Thomas Dean, Robert Givan, and Sonia Leach. Model reduction techniques for computing approximately optimal solutions for Markov decision processes. In *UAI*, 1997.

[13] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76, 1995.

[14] Stefan Edelkamp. Plannning with pattern databases. In *European Conference on Planning*, 2001.

[15] T. Ellman. Abstraction via approximate symmetry. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 1993.

[16] F. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design: An International Journal*, August 1996.

[17] David Ferguson and Anthony Stentz. Focussed dynamic programming: Extensive comparative results. Technical Report CMU-RI-TR-04-13, Robotics Institute, Carnegie Mellon University, March 2005.

[18] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[19] M. Fox and D. Long. The detection and exploitation of symmetry in planning problems. In *IJCAI*, 1999.

[20] M. Fox and D. Long. Extending the exploitation of symmetries in planning. In *AIPS*, 2002.

[21] M. Fox, D. Long, and J. Porteous. Abstraction-based action ordering in planning. In *IJCAI*, 2005.

[22] M. Fox, D. Long, and J. Porteous. Discovering near symmetry in graphs. In *Proceedings of AAAI*, 2007.

[23] Thomas Gartner, Kurt Driessens, and Jan Ramon. Graph kernels and gaussian processes for relational reinforcement learning. In *Thirteenth International Conference on Inductive Logic Programming (ILP-2003)*, 2003.

[24] Thomas Gartner, John W. Lloyd, and Peter A. Flach. Kernels and distances for structured data. *Machine Learning*, 3(57), 2004.

[25] James J. Gibson. The theory of affordances. In Robert Shaw and John Bransford, editors, *Perceiving, Acting, and Knowing*. Lawrence Erlbaum Associates, 1977.

[26] Robert Givan and Thomas Dean. Model minimization, regression, and propositional STRIPS planning. In *15th IJCAI*, 1997.

[27] Robert Givan, Tom Dean, and Matthew Greig. Equivalence notions and model minimization in Markov decision processes. *Artificial Intelligence*, 147:163–223, 2003.

[28] Robert Givan, Sonia Leach, and Thomas Dean. Bounded parameter Markov decision processes. In *Proceedings of the European Conference on Planning (ECP-97)*, 1997.

[29] Robert Givan, Sonia Leach, and Thomas Dean. Bounded parameter Markov decision processes. *Artificial Intelligence*, 2000.

[30] A. Grossman, S. Holldobler, and O. Skvortsova. Symbolic dynamic programming within the fluent calculus. In *Proceedings of the IASTED International conference on Artificial and Computational Intelligence*, 2002.

[31] E. Guere and R. Alami. One action is enough to plan. In *IJCAI*, 2001.

[32] Eric Hansen and Shlomo Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 2001.

[33] P. Haslum and H. Geffner. Admissible heuristics for optimal planning. In *AIPS*, 2000.

[34] P. Haslum and P. Jonsson. Planning with reduced operator sets. In *AIPS*, 2000.

[35] David Haussler. Convolution kernels on discrete structures. Technical Report UCSC-CRL-99-10, University of California at Santa Cruz, July 1999.

[36] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14, 2001.

[37] J.E. Hopcroft. An *n log n* algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, New York, 1971. Academic Press.

[38] David Joslin and Amitabha Roy. Exploiting symmetry in lifted CSPs. In *AAAI*, 1997.

[39] K. Kersting, M. van Otterlo, and L. de Raedt. Bellman goes relational. In *International Conference on Machine Learning (ICML-04)*, 2004.

[40] Sven Koenig and Maxim Likhachev. D* Lite. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI)*, 2002.

[41] Sven Koenig and Maxim Likhachev. Incremental A*. In *Advances in Neural Information Processing Systems 14 (NIPS 2001)*, 2002.

[42] Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong planning A*. *Artificial Intelligence*, 155:93–146, May 2004.

[43] R. Korf. Finding optimal solutions to Rubik's cube using pattern databases. In *AAAI*, 1998.

[44] Nicholas Kushmerick, Steve Hanks, and Daniel Weld. Algorithm for probabilistic planning. *Artificial Intelligence*, 76, July 1995.

[45] Stephen M. Majercik and Michael L. Littman. Contingent planning under uncertainty via stochastic satisfiability. *Artificial Intelligence*, 147, 2003.

[46] B. McKay. Practical graph isomorphism. *Congr. Numer.*, 30, 1981.

[47] T. Miyazaki. The complexity of McKay's canonical labeling algorithm. *Groups and Computation II*, 28, 1997.

[48] Matthew Molineaux and David W. Aha. TIELT: A testbed for gaming environments. In *AAAI*, 2005.

[49] B. Nebel, J. Koehler, and Y. Dimopoulos. Ignoring irrelevant facts and operators in plan generation. In *Proc. European Conference on Planning (ECP-97)*, 1997.

[50] Hanna M. Pasula, Luke S. Zettlemoyer, and Leslie Pack Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Inteligence Research*, 29, 2007.

[51] Carl Adam Petri. Fundamentals of a theory of asynchronous information flow. In *First IFIP World Computer Congress*, pages 386–390, publisher = North Holland, 1963.

[52] M. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 1994.

[53] J. Rintanen. Symmetry reduction for SAT representations of transition systems. In *ICAPS*, 2004.

[54] P. H. Starke. Reachability analysis of Petri nets using symmetries. *Source Systems Analysis Modelling Simulation*, 8, 1991.

[55] Anthony Stentz. The focussed D* algorithm for real-time replanning. In *14th Internationational Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

[56] S.V.N. Vishwanathan and A. Smola. Fast kernels for string and tree matching. In B. Schölkopf, K. Tsuda, and J.P. Vert, editors, *Kernel Methods in Computational Biology*. MIT Press, 2004.

[57] website. FF Homepage. http://members.deri.at/ joergh/ff.html.

[58] website. IPC-02 Homepage: 2nd International Planning Competition. http://planning.cis.strath.ak.uk/competition, 2002.

[59] H. Younes and M. Littman. PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects. In *In Proceedings of the 14th International Conference on Automated Planning and Scheduling*, 2003.

[60] Luke Zettlemoyer, Hanna Pasula, and Leslie Pack Kaelbling. Learning planning rules in noisy stochastic worlds. In *Twentieth AAAI (AAAI-05)*, 2005.