APPSSAT: Approximate Probabilistic Planning Using Stochastic Satisfiability

Stephen M. Majercik

Bowdoin College, Brunswick ME 04011, USA,
`smajerci@bowdoin.edu`,
`http://www.bowdoin.edu/~smajerci`

Abstract. We describe APPSSAT, an anytime probabilistic contingent planner based on ZANDER, a probabilistic contingent planner that operates by converting the planning problem to a stochastic satisfiability (Ssat) problem and solving that problem instead [1]. The values of some of the variables in an Ssat instance are probabilistically determined; APPSSAT considers the most likely instantiations of these variables (the most probable situations facing the agent) and attempts to construct an approximation of the optimal plan that succeeds under those circumstances, improving that plan as time permits. Given more time, less likely instantiations/situations are considered and the plan is revised as necessary. In some cases, a plan constructed to address a relatively low percentage of possible situations will succeed for situations not explicitly considered as well, and may return an optimal or near-optimal plan. We describe experimental results showing that APPSSAT can find suboptimal plans in cases in which ZANDER is unable to find the optimal (or any) plan. Although the test problems are small, the anytime quality of APPSSAT means that it has the potential to efficiently derive suboptimal plans in larger, time-critical domains in which ZANDER might not have sufficient time to calculate *any* plan. We also suggest further work needed to bring APPSSAT closer to attacking real-world problems.

1 Introduction

Previous research has extended the planning-as-satisfiability paradigm to support probabilistic contingent planning; in [1], it was shown that a probabilistic, partially observable, finite-horizon, contingent planning problem can be encoded as a stochastic satisfiability (Ssat) [2] instance such that the solution to the Ssat instance yields a contingent plan with the highest probability of reaching a goal state. This has been used to construct ZANDER, a probabilistic contingent planner [1] that is competitive with three other leading approaches: techniques based on partially observable Markov decision processes, graphplan-based approaches, and partial-order planning approaches [1]. Notably, ZANDER achieves this competitive level using a relatively naïve Ssat solver that does not attempt to adapt many of the advanced features that make today's state-of-the-art SAT solvers so efficient. APPSSAT is a probabilistic contingent planner based on ZANDER that produces an approximate contingent plan and improves that plan as time permits. APPSSAT does this by considering the most probable situations facing the agent and constructing a plan, if possible, that succeeds under those circumstances. Given more time, less likely situations are considered and the plan is revised as necessary.

Other researchers have explored the possibility of using approximation to speed the planning process. In "anytime synthetic projection" a set of control rules establishes a base plan which has a certain probability of achieving the goal [3]. Time permitting, the probability of achieving the goal is incrementally increased by identifying failure situations that are likely to be encountered by the current plan and synthesizing additional control rules to handle these situations. Similarly, MAHINUR is a probabilistic partial-order planner that also creates a base plan with some probability of success and then improves that plan [4]. MAHINUR identifies those contingencies whose failure would have the greatest negative impact on the plan's success and focuses its planning efforts on generating plan branches to deal with those contingencies. Several domain assumptions (including a type of subgoal decomposability) that underlie the design of MAHINUR are identified in [4], and there are no guarantees on the correctness of MAHINUR for domains in which these assumptions are violated.

Exploring approximation techniques in Markov decision processes (MDPs) and partially observable Markov decision processes (POMDPs) has been a very active area of research in recent years, making an exhaustive survey impossible. Evidence that the value function of a factored MDP can often be well approximated using a factored value function, i.e. a linear combination of restricted basis functions, each of which refers to only a small subset of variables in the MDP, has been presented in [5], and it is shown that this approximation technique can be used as a subroutine in a policy iteration process to solve factored MDPs [6]. [7] builds on this work by using factored value functions to create the first approximate MDP solution techniques that use max-norm projection. And [8] explores how to choose a good basis set or improve an existing basis set.

In [9] value functions are represented using decision trees and these decision trees are pruned so that the leaves represent ranges of values, thereby approximating the value function. In [10] that work was extended by applying it to algebraic decision diagrams (ADDs), which can represent functions more compactly than decision trees. ADDs have been used by others as well [11] as a basis for approximation techniques. [12] show how to combine value directed compression with bounded policy iteration that can deal with artificially generated network problems with millions of states. [13] explored the possibility of generating a value function for a first-order MDP (an MDP with existential and universal quantification) by representing them as a combination of first-order basis functions and using linear programming to find the weights these basis functions.

A method for choosing, with high probability, approximately optimal actions in an infinite-horizon discounted Markov decision process using truncated action sequences and random sampling is described in [14, 15]. They show that it is sufficient to consider the first H actions in the sequence, for an appropriate choice of H, and then evaluate the space of all H-step action sequences using random sampling. [16] build on this work to create an approximate planning algorithm using belief state simplification.

In [17] the authors transform a POMDP into a simpler *region observable* POMDP in which it is assumed an oracle tells the agent what region its current state is in. This POMDP is easier to solve and they use its solution to construct an approximate solution for the original POMDP. [18] use temporally extended actions and Monte Carlo updates on a set of grid points in belief space to create a new reinforcement learning algorithm that allows a robot to navigate to a goal in an extremely large state space starting with no knowledge of where it is. In [19], learning steps are used in the policy space of a very large relational MDP along with policy-language biases to produce high-quality planners. And [20] describe A-learning, a variant of Q-learning that they use to construct an approximate planning algorithm.

In Section 2, we describe stochastic satisfiability, the basis for both ZANDER and APPSSAT. In Section 3, we describe how ZANDER uses stochastic satisfiability to solve probabilistic planning problems. In Section 4, we describe the APPSSAT algorithm for approximate planning and in Section 5 we describe some experimental results. We conclude with a discussion of further work.

2 Stochastic Satisfiability

SSAT, suggested by Papadimitriou in [21] and explored further by Littman, Majercik & Pitassi in [2], is a generalization of satisfiability (SAT) that is similar to quantified Boolean formulae (QBF). The ordered variables of the Boolean formula in an SSAT problem, instead of being existentially or universally quantified, are existentially or *randomly* quantified. Randomly quantified variables are `true` with a certain probability, and an SSAT instance is satisfiable with some probability that depends on the ordering of and interplay between the existential and randomized variables. The goal is to choose values for the existentially quantified variables that maximize the probability of satisfying the formula.

More formally, an SSAT problem $\Phi = Q_1 v_1 \dots Q_n v_n \phi$ is specified by 1) a *prefix* $Q_1 v_1 \dots Q_n v_n$ that orders a set of n Boolean variables $V = \{v_1, \dots, v_n\}$ and specifies the quantifier Q_i associated with each variable v_i, and 2) a *matrix* ϕ that is a Boolean formula constructed from these variables. More specifically, the prefix $Q_1 v_1 \dots Q_n v_n$ associates a quantifier Q_i, either existential ($\exists_i$) or randomized ($\text{Ⅎ}_i^{\pi_i}$), with the variable v_i. The value of an existentially quantified variable can be set arbitrarily by a solver, but the value of a randomly quantified variable is determined stochastically by π_i, an arbitrary rational probability that specifies the probability that v_i will be `true`. (In the basic SSAT problem described in [2], every randomized variable is `true` with probability 0.5, but it is noted that the probabilities associated with randomized variables can be arbitrary rational numbers.) In this paper, we will use $x_1, x_2, \dots$ for existentially quantified variables and $y_1, y_2, \dots$ for randomly quantified variables.

The matrix ϕ is assumed to be in conjunctive normal form (CNF), i.e. a set of m conjuncted clauses, where each clause is a set of distinct disjuncted literals. A *literal* l is either a variable v (a *positive* literal) or its negation $\overline{v}$ (a *negative* literal). For a literal l, $|l|$ is the variable v underlying that literal and $\bar{l}$ is the "opposite" of l, i.e. if l is v, $\bar{l}$ is $\overline{v}$; if l is $\overline{v}$, $\bar{l}$ is v; A literal l is `true` if it is positive and $|l|$ has the value `true`, or if it is negative and $|l|$ has the value `false`. A literal is *existential* (*randomized*) if $|l|$ is existentially (randomly) quantified. The probability that a randomly quantified variable v has the value `true` (`false`) is denoted $Pr[v]$ ($Pr[\overline{v}]$). The probability that a randomized literal l is `true` is denoted $Pr[l]$. As in a SAT problem, a clause is satisfied if at least one literal is `true`, and unsatisfied, or *empty*, if all its literals are `false`. The formula is satisfied if all its clauses are satisfied.

The solution of an SSAT instance is an assignment of truth values to the existentially quantified variables that yields the maximum probability of satisfaction, denoted $Pr[\Phi]$. (The decision version of the problem asks if there is an assignment such that $Pr[\Phi]$ meets or exceeds a specified threshold θ.) Since the values of existentially quantified variables can be made contingent on the values of randomly quantified variables that appear earlier in the prefix, the solution is, in general, a *tree* that specifies the optimal assignment to each existentially quantified variable x_i for each possible instantiation of the randomly quantified variables that precede x_i in the prefix. A simple example will help clarify this idea before we define $Pr[\Phi]$ formally. Suppose we have the following SSAT problem:

$$\exists x_1, Ⅎ^{0.7} y_1, \exists x_2 \{\{x_1, y_1\}, \{x_1, \overline{y_1}\}, \{y_1, x_2\}, \{\overline{y_1}, \overline{x_2}\}\} \ . \tag{1}$$

The form of the solution is a noncontingent assignment for x_1 plus two contingent assignments for x_2, one for the case when y_1 is `true` and one for the case when y_1 is `false`. In this problem, x_1 should be set to `true` (if x_1 is `false`, the first two clauses become $\{\{y_1\}, \{\overline{y_1}\}\}$, which specify that y_1 must be both `true` *and* `false`), and x_2 should be set to `true` (`false`) if y_1 is `false` (`true`). Since it is possible to satisfy the formula for both values of y_1, $Pr[\Phi] = 1.0$. If we add the clause $\{\overline{y_1}, x_2\}$ to this instance, however, the maximum probability of satisfaction drops to 0.3: x_1 should still be set to `true`, and when y_1 is `false`, x_2 should still be set to `true`. When y_1 is `true`, however, we have the clauses $\{\{\overline{x_2}\}, \{x_2\}\}$, which insist on contradictory values for x_2. Hence, it is possible to satisfy the formula only when y_1 is `false`, and, since $Pr[\overline{y_1}] = 0.3$, the probability of satisfaction, $Pr[\Phi]$, is 0.3.

We will need the following additional notation to define $Pr[\Phi]$ formally. A partial assignment α of the variables V is a sequence of $k \leq n$ literals $l_1; l_2; \ldots; l_k$ such that no two literals in α have the same underlying variable. Given l_i and l_j in an assignment α, $i < j$ implies that the assignment to $|l_i|$ was made before the assignment to $|l_j|$. A positive (negative) literal v ($\overline{v}$) in an assignment α indicates that the variable v has the value `true` (`false`). The notation $\Phi(\alpha)$ denotes the SSAT problem Φ' remaining when the partial assignment α has been applied to Φ (i.e. clauses with `true` literals have been removed from the matrix, `false` literals have been removed from the remaining clauses in the matrix, and all variables and associated quantifiers not in the remaining clauses have been removed from the prefix) and $\phi(\alpha)$ denotes ϕ', the matrix remaining when α has been applied. Similarly, given a set of literals L, such that no two literals in L have the same underlying variable, the notation $\Phi(L)$ denotes the SSAT problem Φ' remaining when the assignments indicated by the literals in L have been applied to Φ, and (i.e. clauses with `true` literals have been removed from the matrix, `false` literals have been removed from the remaining clauses in the matrix, and all variables and associated quantifiers not in the remaining clauses have been removed from the prefix). $\phi(L)$ denotes ϕ', the matrix remaining when the assignments indicated by the literals in L have been applied. A literal $l \notin \alpha$ is *active* if some clause in $\phi(\alpha)$ contains l; otherwise it is *inactive*.

Given an SSAT problem Φ, the maximum probability of satisfaction of Φ, denoted $Pr[\Phi]$, is defined according to the following recursive rules:

1. If ϕ contains an empty clause, $Pr[\Phi] = 0.0$.
2. If ϕ is the empty set of clauses, $Pr[\Phi] = 1.0$.
3. If the leftmost quantifier in the prefix of Φ is existential and the variable thus quantified is v, then $Pr[\Phi] = \max(Pr[\Phi(v)], Pr[\Phi(\overline{v})])$.
4. If the leftmost quantifier in ϕ is randomized and the variable thus quantified is v, then $Pr[\Phi] = (Pr[\Phi(v)] \times Pr[v]) + (Pr[\Phi(\overline{v})] \times Pr[\overline{v}])$.

These rules express the intuition that a solver can select the value for an existentially quantified variable that yields the subproblem with the higher probability of satisfaction, whereas a randomly quantified variable forces the solver to take the weighted average of the two possible results. These rules imply that a solver should build a tree of all possible assignments to the variables, assigning values to variables in the order specified by the prefix. At each leaf, the probability of satisfaction of the complete assignment represented by the path to that leaf is either 1.0, if the assignment satisfies all the clauses, leaving the empty set of clauses, or 0.0, if the assignment produces an empty clause. The probability of satisfaction of an internal node is calculated from the probabilities of its children. If the node represents an assignment to an existential variable x_i, the solver can choose the value for x_i that leads to the higher probability of satisfaction. If the node represents an assignment to a randomized variable y_i, however, the solver cannot choose the value for y_i. Instead, the probability of satisfaction at that node is the average of the two probabilities of satisfaction that the different values of y_i lead to, weighted by the probabilities of the values of y_i. (As an aside, we note that any QBF instance can be solved by transforming it into an SSAT instance—replace the universal quantifiers with randomized quantifiers—and checking whether $Pr[\Phi] = 1.0$.)

There are simplifications that allow an algorithm implementing this recursive definition to avoid the generally infeasible task of enumerating all possible assignments. Of course, if the empty set of clauses, or an empty clause, is reached before a complete assignment is made, the solver can immediately return 1.0, or 0.0, respectively. Further efficiencies are gained by interrupting the normal left-to-right evaluation of quantifiers to take advantage of *unit* and *pure* literals.

A literal l is *unit* if it is the only literal in some clause (and we will refer to such a clause as a *unit clause*); in this case, $|l|$ must be assigned the value that makes l `true`. A literal l is *pure* if l is active and $\bar{l}$ is inactive; if l is an existential pure literal, $|l|$ can be set to make l `true` without changing $Pr[\Phi]$. These simplifications modify the rules given above for determining $Pr[\Phi]$, but we omit a restatement of the modified rules, instead describing an algorithm to solve SSAT instances based on the modified rules (Figure 1). Note that both ZANDER and APPSSAT construct and return the optimal solution tree (plan), but we omit the details of solution tree construction in the algorithm description.

```
SolveSSAT ($\Phi$)
  if $\phi$ contains an empty clause: return 0.0;
  if $\phi$ is the empty set of clauses: return 1.0;
  if some l is an existential unit literal:
    return SolveSSAT ($\Phi$(l));
  if some l is a randomized unit literal:
    return SolveSSAT ($\Phi$(l)) * Pr[l];
  if some l is an existential pure literal:
    return SolveSSAT ($\Phi$(l));
  if the leftmost quantifier in $\Phi$ is $\exists$ and its variable is v:
      if l is the literal corresponding to the first value
        of v explored and SolveSSAT ($\Phi$(l)) = 1.0:
        return 1.0;
      else:
        return max(SolveSSAT ($\Phi$(v)), SolveSSAT ($\Phi$($\bar{v}$)));
  if the leftmost quantifier in $\Phi$ is $\forall$ and its variable is v:
    return SolveSSAT ($\Phi$(v)) * Pr[v] + SolveSSAT ($\Phi$($\bar{v}$)) * Pr[$\bar{v}$];
```

Fig. 1. The basic algorithm for solving SSAT instances

3 ZANDER

ZANDER works on partially observable probabilistic propositional planning domains consisting of a finite set of distinct *propositions*, any of which may be `true` or `false` at any discrete time t. A *state* is an assignment of truth values to these propositions. A possibly probabilistic *initial state* is specified by a set of decision trees, one for each proposition. *Goal states* are specified by a partial assignment to the set of propositions; any state that extends this partial assignment is a goal state. Each of a finite set of *actions* probabilistically transforms a state at time step t into a state at time step $t+1$ and so induces a probability distribution over the set of all states at time step $t+1$. A subset of the set of propositions is the set of *observable propositions*. The task is to find an action for each time step t as a function of the value of observable propositions at time steps before step t and that maximizes the probability of reaching a goal state.

ZANDER translates the planning problem into an SSAT problem. Figure 2 shows an example of such an SSAT plan encoding. In this problem, a part must be painted, but the paint action succeeds only with probability 0.7 and it is an error to try to paint the part if it is already painted. The agent has two time steps, so the best plan is to paint the part at time step 1 and observe whether the action was successful, painting again at time step 2 if it was not, and doing nothing (noop) at time step 2 otherwise. There are five types of clauses: initial conditions, goal conditions, action exclusivity, action effects, and frame axioms.

$$\exists pa_1 \exists no_1 \rotatebox[origin=c]{180}{R}\, optd_1 \exists pa_2 \exists no_2 \rotatebox[origin=c]{180}{R}^{0.7} cvpa_1 \rotatebox[origin=c]{180}{R}^{0.7} cvpa_2 \exists ptd_0 \exists err_0 \exists ptd_1 \exists err_1 \exists ptd_2 \exists err_2$$

$$\begin{array}{lll}
\{\ \{\overline{pa_0}\} & \wedge\ \{pa_2\} & \wedge \\
\quad \{\overline{err_0}\} & \wedge\ \{\overline{err_2}\} & \wedge \\
\quad \{pa_1 \vee no_1\} & \wedge\ \{pa_2 \vee no_2\} & \wedge \\
\quad \{\overline{pa_1} \vee \overline{no_1}\} & \wedge\ \{\overline{pa_2} \vee \overline{no_2}\} & \wedge \\
\quad \{\overline{pa_1} \vee \overline{ptd_0} \vee ptd_1\} & \wedge\ \{\overline{pa_2} \vee \overline{ptd_1} \vee ptd_2\} & \wedge \\
\quad \{\overline{pa_1} \vee ptd_0 \vee \overline{cvpa_1^{0.7}} \vee ptd_1\} & \wedge\ \{\overline{pa_2} \vee ptd_1 \vee \overline{cvpa_2^{0.7}} \vee ptd_2\} & \wedge \\
\quad \{\overline{pa_1} \vee ptd_0 \vee cvpa_1^{0.7} \vee \overline{ptd_1}\} & \wedge\ \{\overline{pa_2} \vee ptd_1 \vee cvpa_2^{0.7} \vee \overline{ptd_2}\} & \wedge \\
\quad \{\overline{pa_1} \vee \overline{ptd_1} \vee optd_1\} & \wedge & \\
\quad \{\overline{pa_1} \vee ptd_1 \vee \overline{optd_1}\} & \wedge & \\
\quad \{\overline{pa_1} \vee \overline{err_0} \vee err_1\} & \wedge\ \{\overline{pa_2} \vee \overline{err_1} \vee err_2\} & \wedge \\
\quad \{\overline{pa_1} \vee err_0 \vee \overline{ptd_0} \vee err_1\} & \wedge\ \{\overline{pa_2} \vee err_1 \vee \overline{ptd_1} \vee err_2\} & \wedge \\
\quad \{\overline{pa_1} \vee err_0 \vee ptd_0 \vee \overline{err_1}\} & \wedge\ \{\overline{pa_2} \vee err_1 \vee ptd_1 \vee \overline{err_2}\} & \wedge \\
\quad \{\overline{ptd_0} \vee ptd_1\} & \wedge\ \{\overline{ptd_1} \vee ptd_2\} & \wedge \\
\quad \{ptd_0 \vee \overline{ptd_1} \vee pa_1\} & \wedge\ \{ptd_1 \vee \overline{ptd_2} \vee pa_2\} & \wedge \\
\quad \{\overline{err_0} \vee err_1\} & \wedge\ \{\overline{err_1} \vee err_2\} & \wedge \\
\quad \{err_0 \vee \overline{err_1} \vee pa_1\} & \wedge\ \{err_1 \vee \overline{err_2} \vee pa_2\} & \wedge \\
\quad \{\overline{no_1} \vee \overline{optd_1}\} & \wedge & \\
\quad \{pa_1 \vee \overline{optd_1}\}\ \} & &
\end{array}$$

Fig. 2. An example of an SSAT plan encoding, where pa_t = paint at time step t, no_t = noop at time step t, $optd_1$ = observe that the part is painted after the action at time step 1, $cvpa_t^{0.7}$ = chance variable encoding the success probability of action pa_t, ptd_t = painted at time step t, and err_t = error at time step t

Initial and goal condition clauses are unit clauses that force the initial and goal conditions to be satisfied. In this problem, the initial condition clauses are $\{\overline{pa_0}\}$ and $\{\overline{err_0}\}$, i.e. the part is not painted and there is no error at time step 0. The goal condition clauses are $\{pa_2\}$ and $\{\overline{err_2}\}$, i.e. the part must be painted and there must still be no error at time step 2.

Action exclusivity clauses ensure that exactly one action is taken at each time step. In this problem, $\{pa_1 \vee no_1\}$ forces one of the actions (paint or noop) to be selected at time step 1 and $\{\overline{pa_1} \vee \overline{no_1}\}$ forces one of the actions to *not* be selected. There are similar clauses for time step 2. Note that in the general case, for A actions, at each time step ($t > 0$) there will be a single clause containing A positive literals for all the actions (one action must be taken) and there will be $\binom{A}{2}$ clauses containing all possible pairs of negated action literals (for each possible pair of actions, at least one of them must not have been taken).

Action effects clauses model the (sometimes probabilistic) effects of actions. The effect of the paint action on whether or not the part is observed to be painted is a deterministic action effect. For example, if paint action is taken at time step 1 and the part is successfully painted, then it will definitely be observed that the part is painted, i.e. $pa_1 \wedge ptd_1 \rightarrow optd_1$. (Note that the effects of actions, including ensuing observations, have the same time index as the action that produced them.) Negating the antecedent and converting the implication to a disjunction yields the equivalent $\{\overline{pa_1} \vee \overline{ptd_1} \vee optd_1\}$, which becomes a clause in the plan encoding. Note that, if the first two literals are `false`, i.e. the paint action is taken at time step 1 and the part is painted at time step 1, then the third literal, $optd_1$, must be `true` to satisfy the clause, i.e. the part is observed to be painted at time step 1.

The effect of the paint action on the part, if the part is unpainted, is a probabilistic action effect. For example, if the paint action is taken at time step 1 and the part is not painted at time step 0, the part will be painted at time step 1 with probability 0.7. This can be expressed by two implications involving a chance variable:

$$pa_1 \wedge \overline{ptd_0} \wedge cvpa_1^{0.7} \rightarrow ptd_1$$

and:

$$pa_1 \wedge \overline{ptd_0} \wedge \overline{cvpa_1^{0.7}} \rightarrow \overline{ptd_1}$$

The first implication says that if the paint action is taken at time step 1 and the part is not painted at time step 0 *and* the chance variable $cvpa_1^{0.7}$ is `true` (which will be the case with probability 0.7), then the part will become painted. The second implication says that if the paint action is taken at time step 1 and the part is not painted at time step 0 *and* the chance variable $cvpa_1^{0.7}$ is `false` (which will be the case with probability 0.3), then the part will not become painted. Again, negating the antecedents and converting the implications to disjunctions yields the equivalent:

$$\{\overline{pa_1} \vee ptd_0 \vee \overline{cvpa_1^{0.7}} \vee ptd_1\}$$

and:

$$\{\overline{pa_1} \vee ptd_0 \vee cvpa_1^{0.7} \vee \overline{ptd_1}\}$$

which become clauses in the plan encoding. Note that if the first two literals are `false`, i.e. the paint action is taken at time step 1 and the part was not painted at time step 0, we have:

$$\{\overline{cvpa_1^{0.7}} \vee ptd_1\}$$

and:

$$\{cvpa_1^{0.7} \vee \overline{ptd_1}\}.$$

Then, in the first case, if the part is *not* painted at time step 1 (ptd_1 is `false`), the chance variable $cvpa_1^{0.7}$ must be `false` to satisfy the clause, and this will be the case with probability 0.3. In the second case, if the part *is* painted at time step 1 (ptd_1 is `true`), the chance variable $cvpa_1^{0.7}$ must be `true` to satisfy the clause, and this will be the case with probability 0.7.

Frame axiom clauses model the fact that if the value of a state proposition changes, one of some subset of actions (usually a small subset) must have been taken that is capable of effecting that change. For example, if the part becomes newly painted it must be the case that the paint action was taken, i.e. $\overline{ptd_0} \wedge ptd_1 \rightarrow pa_1$. Note that if more than one action can produce the change, the consequent will be the disjunction of all the actions that can produce that change. In this case, negating the antecedent and converting the implication to a disjunction yields the equivalent $\{ptd_0 \vee \overline{ptd_1} \vee pa_1\}$, which becomes a clause in the plan encoding. If the first two literals are `false`, i.e. the part was not painted at time step 0 and then became painted at time step 1, the third literal must be `true` to satisfy the clause, i.e. the paint action must have been taken at time step 1. If no actions can produce a particular change, then the consequent is `false` and the resulting clause enforces the fact that no action can cause that change to occur. For example, the clauses $\{\overline{ptd_0} \vee ptd_1\}$ and $\{\overline{ptd_1} \vee ptd_2\}$ model the fact that there is no action that can cause the part to become "unpainted," i.e. if the painted state proposition is `true` at time step 0 (1), it must be `true` at time step 1 (2) to satisfy these clauses.

The unit clauses in Figure 2 force certain variables to have certain values. The initial conditions $\{\overline{pa_0}\}$ and $\{\overline{err_0}\}$ force pa_0 and err_0 to be `false` and the goal conditions $\{pa_2\}$ and $\{\overline{err_2}\}$ force pa_2 and err_2 to be `true` and `false`, respectively. These variables are assigned the values necessary to satisfy the unit clauses they appear in and the results are propagated throughout the formula (unit clauses indirectly created by the elimination of existing unit clauses are also eliminated and their effects propagated) before APPSSAT operates on the formula (just as these clauses would force ZANDER to set the values of the variables in these clauses first regardless of their position in the quantifier ordering). Figure 3 shows the SSAT plan encoding after all the unit clauses have been eliminated and their effects propagated (although it attempts to maintain the relative position the clauses occupy in Figure 2). Note that although the goal conditions have disappeared completely from the formula, satisfying this reduced formula implies that the goal conditions are satisfied as well.

The variables in an SSAT plan encoding fall into three segments [1]: the action-observation segment (variables pa_1, no_1, $optd_1$, pa_2, no_2 in Figure 3), the domain uncertainty segment (variables $cvpa_1^{0.7}$, $cvpa_2^{0.7}$ in Figure 3), and a segment representing the result of the actions taken given the domain uncertainty (variable ptd_1 in Figure 3). The action-observation segment is an alternating sequence of existentially quantified variable blocks (one block for each action choice) and randomly quantified variable blocks (one block for each set of possible observations at a time step). In APPSSAT, the randomly quantified observation variables will become *branch variables* with no associated proba-

$$\exists pa_1 \exists no_1 \rotatebox[origin=c]{180}{R}\, optd_1 \exists pa_2 \exists no_2 \rotatebox[origin=c]{180}{R}^{0.7} cvpa_1 \rotatebox[origin=c]{180}{R}^{0.7} cvpa_2 \exists ptd_1$$

$$\begin{array}{lll}
\{\ \{pa_1 \vee no_1\} & \wedge\ \{pa_2 \vee no_2\} & \wedge \\
\ \ \{\overline{pa_1} \vee \overline{no_1}\} & \wedge\ \{\overline{pa_2} \vee \overline{no_2}\} & \wedge \\
\ \ \{\overline{pa_1} \vee \overline{cvpa_1^{0.7}} \vee ptd_1\} \wedge & & \\
\ \ \{\overline{pa_1} \vee cvpa_1^{0.7} \vee \overline{ptd_1}\} & \wedge\ \{\overline{pa_2} \vee ptd_1 \vee cvpa_2^{0.7}\} & \wedge \\
\ \ \{\overline{pa_1} \vee \overline{ptd_1} \vee optd_1\} & \wedge & \\
\ \ \{\overline{pa_1} \vee ptd_1 \vee \overline{optd_1}\} & \wedge & \\
 & \ \ \ \{\overline{pa_2} \vee \overline{ptd_1}\} & \wedge \\
\ \ \{\overline{ptd_1} \vee pa_1\} & \wedge\ \{ptd_1 \vee pa_2\} & \wedge \\
\ \ \{\overline{no_1} \vee \overline{optd_1}\} & \wedge & \\
\ \ \{pa_1 \vee \overline{optd_1}\}\ \} & &
\end{array}$$

Fig. 3. The SSAT plan encoding shown in Figure 2 after the unit clauses have been eliminated, where pa_t = paint at time step t, no_t = noop at time step t, $optd_1$ = observe that the part is painted after the action at time step 1, $cvpa_t^{0.7}$ = chance variable encoding the success probability of action pa_t, and ptd_t = painted at time step t

bility (see Section 4 for a detailed explanation of branch variables). We will refer to an instantiation of the action and observation variables as an *action-observation path.* The domain uncertainty segment is a single block containing all the randomly quantified variables that modulate the impact of the actions on the observation and state variables. The result segment is a single block containing all the existentially quantified state variables. Essentially, ZANDER uses the solver described in Section 2 to find an assignment *tree* that specifies the assignments to existentially quantified action variables for all possible settings of the observation variables, such that the probability of satisfaction (which is also the probability that the plan will reach the goal) is maximized [1]. In what follows, we will refer to such a tree as an *action-observation tree.* We will also sometimes refer to existentially and randomly quantified variables as *choice* and *chance* variables, respectively.

4 APPSSAT

Before we describe APPSSAT it is worth looking at a previous approach to approximation in this framework. This approach illuminates some of the problems associated with formulating an approximation algorithm in this framework and explains some of the choices we made in developing APPSSAT. An algorithm called randevalssat that uses stochastic local search in a reduced plan space is described in [2]. The randevalssat algorithm uses random sampling to select a subset of possible chance variable instantiations (thus limiting the size of the contingent plans considered) and stochastic local search to find the best size-bounded plan. There are two problems with this approach. First, since chance variables are used to describe observations, a random sample of the chance variables describes an observation sequence as well as an instantiation of the uncertainty in the domain, and the observation sequence thus produced may not be *observationally consistent*, and these inconsistencies can make it impossible to find a plan, even if one exists. Second, this algorithm returns a partial policy, that specifies actions only for those situations represented by paths in the random sampling of chance variables. APPSSAT addresses these two problems by:

1. designating each observation variable as a new type of variable, termed a *branch* variable that does *not* have a probability associated with it, and
2. evaluating the approximate plan's performance under all circumstances, not just those used to generate the plan.

The introduction of branch variables violates the pure SSAT form of the plan encoding, but is justified, we think, for the sake of conceptual clarity. We could achieve the same end in the pure SSAT form by making observation variables chance variables with a probability of 0.5 (as in [1]), and not including them when the possible chance-variable assignments are enumerated. But, rather than taking this circuitous route, we have chosen to acknowledge the special role played by observation variables; these variables indicate a potential branch in a contingent plan (hence, the name). Like choice variables, a branch variable does not have an associated probability, but, unlike choice variables, the value of a branch (observation) variable node in the assignment tree described above is a combination of, rather than a choice between, the values of its children. More specifically, the value of such a node is the *sum* of the values of its children, since the probability of satisfaction (reaching the goal) for each course of action contingent on the observation branch contributes additively to the overall probability of satisfaction (reaching the goal). This introduces a minor modification into the ZANDER approach and has the benefit of clarifying the role of the observation variables. In the example shown in Figures 2 and 3, the variable $optd_1$ (observe that the part is painted at time step 1) is an observation, or branch, variable. The solution tree will split at this point to reflect the fact that the optimal plan may require different actions depending on the status of this observation. In what follows, we will use the terms *observation variable* and *branch variable* interchangeably.

APPSSAT incrementally constructs the optimal action-observation tree by updating the probabilities of the possible action-observation paths in that tree as it processes the instantiations of the chance variables. We provide a brief summary of APPSSAT's operation, after which we describe the details of its operation. In a single iteration, APPSSAT:

1. Selects the chance variable instantiation with the next highest probability π. In the painting example (Figure 3 in which the unit clauses have been propagated), the instantiation with the highest probability would be $cvpa_1^{0.7} =$ `true` and $cvpa_2^{0.7} =$ `true`, with a probability $\pi_1 = 0.7 \times 0.7 = 0.49$.

2. Uses a SAT solver to find all the satisfying assignments that extend this chance variable assignment. Continuing the example, there are two satisfying assignments that are consistent with this setting of the chance variables:

($pa_1 =$ `false`, $noop_1 =$ `true`, $optd_1 =$ `false`, $pa_2 =$ `true`, $noop_2 =$ `false`) and ($pa_1 =$ `true`, $noop_1 =$ `false`, $optd_1 =$ `true`, $pa_2 =$ `false`, $noop_2 =$ `true`). The first assignment indicates that the agent did nothing (noop) at time step 1, observed that the part was not painted, and then executed the paint action at time step 2. The second assignment indicates that the agent executed the paint action at time step 1, observed that the part was painted, and then did nothing at time step 2. Note that painting at both time steps is not consistent with this chance variable instantiation because $cvpa_1^{0.7} =$ `true` indicates that the first paint action would be successful, and painting again would cause an error.

3. Installs the action-observation portion of each satisfying assignment found above in (2) into the action-observation tree and updates, as necessary, the probabilities of success of the actions and the optimal action at each point in the tree. Thus, the action-observation paths (with the `false` actions deleted) would be $noop_1 =$ `true` $\rightarrow optd_1 =$ `false` $\rightarrow pa_2 =$ `true` and $pa_1 =$ `true`– $optd_1 =$ `true` $\rightarrow noop_2 =$ `true`. These paths would be created in the solution tree and the value of each action would be set at $\pi_1 = 0.49$.

4. Computes the probability of success of the optimal plan so far and compares it to the target threshold probability. The two action-observation paths installed so far disagree on the first action to be taken, so only one can be chosen as the optimal plan so far: paint at time step 1 and then do nothing at time step 2, or do nothing at time step 1 and then paint at time step 2. Since they have the same probability of success, one of them would be chosen arbitrarily and the probability of success would be computed as 0.49. Assuming that APPSSAT was looking for the optimal plan (i.e. the threshold is set at 1.0), APPSSAT would continue processing chance variable instantiations in an attempt to find a better plan.

The chance variable instantiations are efficiently generated in descending order using a priority queue. The satisfying assignments that extend a given chance variable instantiation are found using zChaff [22], currently one of the fastest SAT solvers available. zChaff, however, is not well-suited to finding *all* the satisfying assignments for a particular formula. The method we adopt to address this problem is to add a clause to the current formula for each satisfying assignment found that prevents that assignment from being discovered again. This, however, introduces very long clauses into the formula in large problems, which makes finding a new satisfying assignment increasingly difficult. In addition, we encountered technical problems with zChaff after adding many large clauses. Nonetheless, this method proved feasible for a suitably wide range of problems to demonstrate the efficacy of the APPSSAT approach.

APPSSAT maintains a tree of action-observation paths that indicates the current optimal plan. In other words, each action node indicates the values (probabilities of satisfaction) of each action, given the subtree rooted there, and the best action. Each time a satisfying assignment is found it is installed in that tree in such a way that the optimal plan so far can be easily extracted from the tree. Given a new action-observation path, the algorithm follows existing nodes in the tree to the extent possible (i.e. if they match the actions and observations in the path being installed). If a point is reached when this is no longer possible, additional nodes are constructed to make it possible to install that action-observation path. The probability of that path (i.e. the probability of the chance variable instantiation that generated the action-observation path) is then propagated upward from the leaf of that path, changing the values of actions and the best action at each action node along that path as necessary. To continue the example begun in the summary of APPSSAT's operation above, the next chance variable instantiation to be processed would be either $cvpa_1^{0.7} = \texttt{true}$ and $cvpa_2^{0.7} = \texttt{false}$, or $cvpa_1^{0.7} = \texttt{false}$ and $cvpa_2^{0.7} = \texttt{true}$, each with a probability $\pi_2 = 0.21$. Suppose the first one is chosen. There is only one satisfying assignment that is consistent with this setting of the chance variables: $pa_1 = \texttt{true}$, $noop_1 = \texttt{false}$, $optd_1 = \texttt{true}$, $pa_2 = \texttt{false}$, $noop_2 = \texttt{true}$. The corresponding action-observation path, $pa_1 = \texttt{true} \rightarrow optd_1 = \texttt{true} \rightarrow noop_2 = \texttt{true}$, is already in the tree, so the value of all the actions on this path (paint at time step 1 and do nothing at time step 2) would be increased to $0.49 + 0.21 = 0.7$. Since the value of doing nothing at time step 1 is still 0.49 and the value of painting at time step 1 is now 0.7, the optimal plan extracted will now be to paint at time step 1 and do nothing at time step 2. This plan has a value, or probability of success, of 0.7, which makes sense since the probability of success of the paint action is 0.7.

Pursuing this example further, we find that the the next chance variable instantiation to be processed, $cvpa_1^{0.7} = \texttt{false}$ and $cvpa_2^{0.7} = \texttt{true}$, is consistent with two action-observation paths: $pa_1 = \texttt{true} \rightarrow optd_1 = \texttt{false} \rightarrow pa_2 = \texttt{true}$ and $noop_1 = \texttt{true} \rightarrow optd_1 = \texttt{false} \rightarrow pa_2 = \texttt{true}$. The first action-observation path agrees on the first action ($pa_1 = \texttt{true}$) with the already-installed action-observation path $pa_1 = \texttt{true} \rightarrow optd_1 = \texttt{true} \rightarrow noop_2 = \texttt{true}$, but differs in the value of the observation and the action at time step 2: this action-observation path provides a course of action when it is observed that the part has not been painted ($optd_1 = \texttt{false}$), namely to paint at time step 2 ($pa_2 = \texttt{true}$). The value of the paint action at time step 2 on this path will have the value 0.21 (the value of the chance variable instantiation that produced it), and this value will be propagated upward to the paint-at-time-step-1 action and will increase the value of this action to $0.7 + 0.21 = 0.91$. Intuitively, this is because we now have a potential plan tree that prescribes an action for both possible values of the observation variable after the paint action is taken at time step 1: when it is observed that the paint action did work, the old action-observation path ($pa_1 = \texttt{true} \rightarrow optd_1 = \texttt{true} \rightarrow noop_2 = \texttt{true}$) prescribes the noop action at time step 2, and when it is observed that the paint action did not work, the new action-observation path ($pa_1 = \texttt{true} \rightarrow optd_1 = \texttt{false} \rightarrow pa_2 = \texttt{true}$) prescribes the paint action at time step 2. The value of the first path, when the observation variable is $\texttt{true}$, is 0.7, and the value of the new path, when the observation variable is $\texttt{false}$, is 0.21. Since the value of an observation (or branch) node is the sum of the values of its children, the value of this observation node, and thus the value of the paint-at-time-step-1 action that leads to it, is 0.91.

The second consistent action-observation path ($noop_1$ = `true` $\rightarrow$ $optd_1$ = `false` – pa_2 = `true`) increases the value of that existing path, but only from 0.49 to 0.49 + 0.21 = 0.7. So the optimal plan extracted at this point would prescribe the paint action at time step 1 and then the noop action at time step 2 if the paint action is successful and another paint action at time step 2 otherwise. The probability of success of this plan would be calculated as 0.91. And this plan and probability of success, which are optimal, will remain the same when the last chance variable instantiation is processed ($cvpa_1^{0.7}$ = `false` and $cvpa_2^{0.7}$ = `false`) since there are no satisfying assignments that extend this instantiation.

Note that APPSSAT knows after the installation of every action-observation path whether the probability of success of the current optimal plan has increased. And because the values of all the actions are maintained in each action node, it is easy to extract the current optimal plan, thus making this an *anytime* algorithm. It is also possible at this point, although computationally relatively expensive to evaluate the real probability of success of that plan. Assuming we have processed only some fraction of the chance variable instantiations and their associated satisfying assignments, the probability of success of the current optimal plan is only a lower bound on the real probability of success of that plan, since the current plan may be successful for some chance variable instantiations that have not yet been processed. APPSSAT can find the real probability of success by evaluating the full assignment tree using that plan, essentially by running the ZANDER algorithm with the values of the action variables dictated by the plan being evaluated. This can be done every time the probability of the current tree increases or at intervals.

If the lower bound on the probability of success of the current optimal plan (or the actual probability of success) is sufficient (either 1.0 or exceeding a user-specified threshold), APPSSAT halts and return the plan and probability; otherwise, APPSSAT continues processing chance variable assignments. Note that the probability of success of the just-extracted plan can be used as a new lower threshold in subsequent plan evaluations, often allowing additional pruning to be done. The quality of the plan produced increases (if the optimal success probability has not already been attained) with the available computation time. Figure 4 presents pseudocode for the algorithm.

Because the chance variable instantiations are investigated in descending order of probability, a plan with a relatively high percentage of the optimal success probability can potentially be found quickly. An exception is a domain in which the high probability situations are hopeless and the best that can be done is to construct a plan that addresses some number of lower probability situations. Even here, the basic SSAT heuristics used will allow APPSSAT to quickly discover that no plan is possible for the high-probability situations, and lead it to focus on the low-probability situations for which a plan is feasible. Of course, if *all* chance-variable assignments are considered, the plan extracted is the optimal plan, but, as we shall see, the optimal plan may sometimes be produced

```
APPSSAT ($\Phi$) {
  nc = number of chance variables;
  k = pow(2, nc) = number of chance variable instantiations;
  cp = current plan, initially empty;
  $\pi_{cp}$ = probability of success of the current plan, initially 0.0;
  $\pi_{thresh}$ = minimum acceptable probability of success;
  w = function that maps action-observation paths to probabilities,
      initially all 0.0;
  j = 1;

  while (j <= k $\wedge$ $\pi_{pc}$ < $\pi_{thresh}$) {
    ci$_j$ = jth chance variable instantiation in descending order
          of probability;
    Pr[ci$_j$] = probability of chance variable instantiation ci$_j$;

    for each action-observation path (aop) found by
    the SAT solver that is consistent with ci$_j$ {
      w(aop) = w(aop) + Pr[ci$_j$];
    }

    cp = current best plan;
    $\pi_{cp}$ = Pr[cp reaches the goal];
  }

  return cp and $\pi_{cp}$

}
```

Fig. 4. The APPSSAT algorithm for approximating the solution of probabilistic planning problems encoded as SSAT instances

even after only a relatively small fraction of the chance-variable assignments have been considered.

Unlike ZANDER, which, in effect, looks at chance variable instantiations at a particular time step based on the instantiation of variables (particularly action variables) at previous times steps, APPSSAT, by enumerating complete instantiations of the chance variables in descending order of probability, examines the most likely outcomes of all actions at all time steps. This approach may seem counter-intuitive. Instead of instantiating *all* the chance variables, perhaps it would make more sense to instantiate chance variables based on the instantiation of variables (particularly action variables) at previous times steps. For example, given the action choice made at the first time step, what is the most likely instantiation of chance variables facing the agent now? This type of approach is more similar to the operation of ZANDER—searching the SSAT tree of possible assignments—except that the search would be sped up by considering only the more likely chance variable instantiations. This would seem to require a significant amount of overhead to keep track of which chance variable instantiations have been checked for which action choices and, in many cases, would entail the repeated solving of a number of Ssat subproblems with one or more chance variable settings changed.

The APPSSAT approach seeks to accomplish a similar goal in a different way. By enumerating complete instantiations of the chance variables in descending order of probability, APPSSAT is choosing the most likely outcomes of all actions at all time steps. Because it is not taking variable independencies into account, it does so somewhat inefficiently. At the same time, however, by instantiating all the chance variables at the same time, APPSSAT reduces the Ssat problem to a much simpler SAT problem. Although this approach will also entail the repeated solving of a number of subproblems with one or more chance variable settings changed, the conjecture is that solving a large number of SAT problems will take less time than solving a large number of Ssat problems, particularly if one makes use of a state-of-the-art SAT solver like zChaff [22]. Obviously, this will depend on the relative number of problems involved, but we have chosen to explore the approach embodied in APPSSAT first.

Most of the operations in APPSSAT can be performed as or more efficiently than the operations necessary in the ZANDER framework. The chance variable instantiations can be generated in descending order in time linear in the number of instantiations using a priority queue. APPSSAT finds all consistent action-observation paths using zChaff, as described above. Although, in some cases, this was clearly faster than the alternative of doing a depth-first search of the assignment tree checking for satisfiability using pruning heuristics (the central operation of ZANDER), it is not clear that this is always the best choice. More investigation is needed here. The current best plan is continuously maintained in the tree of action-observation paths resulting from the processing of the chance variable instantiations. Finally, plan evaluation requires a depth-first search of the entire assignment tree, but heuristics speed up the search, and the resulting probability of success can be used as a lower threshold if the search continues, thus potentially speeding up subsequent computation.

5 Results

We tested APPSSAT on three domains similar to those ZANDER was tested on in [1]: TIGER, COFFEE-ROBOT, and SPEARFISHING. All three problems have uncertain initial conditions and noisy observations. The characteristics of these problems are described in Table 1. All experiments were conducted on a 2.5 GHz Power Mac G5 with 1 Gbyte of RAM, running OS X 10.4.3.

Before discussing the results, we note that ZANDER relies heavily on *memoization*, i.e. saving the results of solved subproblems for possible future reuse. Clearly, this can use up a great deal of memory. At the same time, memoization allows the solver to avoid needless duplication of identical subtrees, so avoiding memoization completely is not always the most effective way to conserve memory. We have provided results for three types of memoization in ZANDER: no

Table 1. Characteristics of test problems

Planning Problem (Number of States)	Size Statistic	Number of Time Steps				
		5	10	15	20	25
TIGER (8)	Num Actions/Step	3	3	3	3	3
	State Propositions/Step	3	3	3	3	3
	Num Observations/Step	1	1	1	1	1
	Total Number Variables	45	90	135	180	225
	Total Number Clauses	154	309	464	619	774
COFFEE (64)	Num Actions/Step	3	3	3	3	3
	State Propositions/Step	6	6	6	6	6
	Num Observations/Step	1	1	1	1	1
	Total Number Variables	83	163	243	323	403
	Total Number Clauses	192	382	572	762	952
SPEAR (256)	Num Actions/Step	8	8	8	8	8
	State Propositions/Step	8	8	8	8	8
	Num Observations/Step	2	2	2	2	2
	Total Number Variables	212	422	632	842	1052
	Total Number Clauses	525	1040	1555	2070	2585

memoization, memoization of all subproblems, and memoization of only those subproblems that are solved when both values of a variable are explored, i.e. do not memoize a subproblem that arises when a unit clause forces the value of a variable. In general, as indicated in Table 2, this last option is usually the best one. (Please refer to Table 2 in the following discussion of results.)

In the TIGER problem, ZANDER easily finds the optimal plans for both a 5-step plan (0.93925 probability of success) and a 10-step plan (0.994371 probability of success). Using memoization allows it to find an optimal 15-step plan (0.998732 probability of success) and the selective memoization technique allows ZANDER to find the optimal 20-step plan (0.999856 probability of success), but at the 25-step level all variants of ZANDER exhaust memory before they are able to find a plan. APPSSAT finds the optimal 5-step plan, although it takes longer than ZANDER. However, although APPSSAT is never able to find the optimal plan for this problem as quickly as ZANDER, APPSSAT is able to find a plan with *some* chance of success in all the cases in which ZANDER exhausts memory and is thus unable to find *any* plan. In the 25-step case, for example, all variants of ZANDER exhaust memory, but APPSSAT is able to produce a plan with a success probability of 0.130602 in 5.87 CPU seconds. For other plan lengths (10, 15, and 20 steps) Table 2 shows success probabilities that APPSSAT is able to reach in reasonable amounts of time. We see the same results for both the COFFEE-ROBOT and SPEARFISHING problems, the disparity between the two planners being most pronounced in the SPEARFISHING domain, in which all variants of ZANDER are unable

Table 2. APPSSAT is often able to produce a plan with some probability of success in cases where ZANDER would run out of memory

Problem	Solver	Type of Memoization	CPU Seconds and Probability (optimal in boldface) by Number of Steps in Plan				
			5	10	15	20	25
TIGER	APP	NA	5.62 **0.93925**	3.13 0.755651	5.45 0.401477	3.81 0.20102	5.87 0.130602
	ZAN	NONE	0.015 **0.93925**	0.45 **0.994371**	M	M	M
		SPLT	0.015 **0.93925**	0.113 **0.994371**	3.84 **0.998732**	185.0 **0.999856**	M
		ALL	0.016 **0.93925**	0.092 **0.994371**	2.82 **0.998732**	M	M
COFFEE	APP	NA	0.47 **0.8906**	0.55 0.7665	0.40 0.815	0.18 0.5	0.23 0.5
	ZAN	NONE	0.11 **0.8906**	M	M	M	M
		SPLT	0.08 **0.8906**	190.98 **0.93761**	M	M	M
		ALL	0.12 **0.8906**	M	M	M	M
SPEAR	APP	NA	0.12 0.23365	0.16 0.23365	0.28 0.23365	1.71 0.23365	2.93 0.23365
	ZAN	NONE	7.80 **0.242737**	M	M	M	M
		SPLT	1.65 **0.242737**	M	M	M	M
		ALL	2.81 **0.242737**	M	M	M	M

APP=APPSSAT, ZAN=ZANDER
NA=Memoizing Not Applicable, NONE=No Memoizing
SPLT=Memoize Only Subproblems Based on Splits
ALL=Memoize All Subproblems, M=Memory exhausted

to produce a plan with a length of 10 or more steps while APPSSAT is able to produce suboptimal plans (probability of success of 0.23365) for all horizons up to 25 steps.

6 Further Work

We need to improve the efficiency of APPSSAT if it is to be a viable approximation technique, and there are a number of techniques we are in the process of implementing that should help us to achieve this goal. First, when APPSSAT is processing the chance variable instantiations in descending order, in many cases the difference between two adjacent instantiations is small. We can probably take advantage of this to find the action-observation paths that satisfy the new chance variable instantiation more quickly.

Second, since we are repeatedly running a SAT solver to find action-observation paths that lead to satisfying assignments for the chance-variable assignments, and since two chance variable assignments will frequently generate the same satisfying action-observation path, it seems likely that we could speed up this process considerably by incorporating learning into APPSSAT.

Finally, we are investigating whether plan simulation (instead of exact calculation of the plan success probability) would be a more efficient way of evaluating the current plan.

One possible use for this (or any) approximation technique is to use the approximation technique in a framework that interleaves planning and execution, in order to scale up to even larger domains than approximation alone could attack. The idea here would be to use the approximation technique to calculate a "pretty good" first action (or action sequence), execute that action or action sequence, and then continue this planning/execution cycle from the new initial state. This approach could improve efficiency greatly (at the expense of optimality) by focusing the planner's efforts only on those contingencies that actually materialize.

Acknowledgment: We would like to thank the anonymous reviewers for suggestions that greatly improved the clarity of several sections of this paper.

References

1. Majercik, S.M., Littman, M.L.: Contingent planning under uncertainty via stochastic satisfiability. Artificial Intelligence **147** (2003) 119–162
2. Littman, M.L., Majercik, S.M., Pitassi, T.: Stochastic Boolean satisfiability. Journal of Automated Reasoning **27** (2001) 251–296
3. Drummond, M., Bresina, J.: Anytime synthetic projection: Maximizing the probability of goal satisfaction. In: Proceedings of the Eighth National Conference on Artificial Intelligence, Morgan Kaufmann (1990) 138–144
4. Onder, N., Pollack, M.E.: Contingency selection in plan generation. In: Proceedings of the Fourth European Conference on Planning. (1997) 364–376
5. Koller, D., Parr, R.: Computing factored value functions for policies in structured MDPs. In: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, The AAAI Press/The MIT Press (1999) 1332–1339
6. Koller, D., Parr, R.: Policy iteration for factored MDPs. In: Proceedings of the Sixteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI 2000). (2000) 326–334
7. Guestrin, C., Koller, D., Parr, R.: Max-norm projections for factored MDPs. In: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence. (2001) 673–682
8. Poupart, P., Boutilier, C., Schuurmans, D., Patrascu, R.: Piecewise linear value function approximation for factored MDPs. In: Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-2002). (2002) 292–299

9. Boutilier, C., Dearden, R.: Approximating value trees in structured dynamic programming. In: Proceedings of the Thirteenth International Conference on Machine Learning. (1996) 56–62
10. St-Aubin, R., Hoey, J., Boutilier, C.: APRICODD: Approximate policy construction using decision diagrams. In: Advances in Neural Information Processing Systems 13 (NIPS-2000). (2000) 1089–1095
11. Feng, Z., Hansen, E.: Approximate planning for factored POMDPs. In: Sixth European Conference on Planning (ECP-01). (2001)
12. Poupart, P., Boutilier, C.: VDCBPI: an approximate scalable algorithm for large scale POMDPs. In: Advances in Neural Information Processing Systems 17 (NIPS-2004). (2004) 1081–1088
13. Sanner, S., Boutilier, C.: Approximate linear programming for first-order MDPs. In: Proceedings of the Twenty-first Conference on Uncertainty in Artificial Intelligence (UAI-05). (2005) 509–517
14. Kearns, M.J., Mansour, Y., Ng, A.Y.: A sparse sampling algorithm for near-optimal planning in large markov decision processes. Machine Learning **49** (2002) 193–208
15. Kearns, M., Mansour, Y., Ng, A.: Approximate planning in large POMDPs via reusable trajectories. In: Advances in Neural Information Processing Systems 12 (Proceedings of the 1999 Conference. MIT Press (2000)
16. McAllester, D.A., Singh, S.: Approximate planning for factored POMDPs using belief state simplification. In: Proceedings of the Fifteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-99). (1999) 409–416
17. Zhang, N.L., Lin, W.: A model approximation scheme for planning in partially observable stochastic domains. Journal of Artificial Intelligence Research **7** (1997) 199–230
18. Theocharous, G., Kaelbling, L.P.: Approximate planning in POMDPs with macro-actions. In: Advances in Neural Information Processing Systems 16 (NIPS-2003). (2003)
19. Fern, A., Yoon, S., Givan, R.: Approximate policy iteration with a policy language bias. In: Advances in Neural Information Processing Systems 16 (NIPS-2003). (2003)
20. Blatt, D., Murphy, S., Zhu, J.: A-learning for approximate planning. Technical Report 04-63, The Methodology Center, Pennsylvania State University (2004)
21. Papadimitriou, C.H.: Games against nature. Journal of Computer Systems Science **31** (1985) 288–301
22. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: 39th Design Automation Conference (DAC 2001). (2001)