Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)

Computing Contingent Plans Using Online Replanning

Radimir Komarnitsky and Guy Shani
Information Systems Engineering
Ben Gurion University, Israel
{radimir,shanigu}@bgu.ac.il

Abstract

In contingent planning under partial observability with sensing actions, agents actively use sensing to discover meaningful facts about the world. For this class of problems the solution can be represented as a plan tree, branching on various possible observations. Recent successful approaches translate the partially observable contingent problem into a non-deterministic fully observable problem, and then use a planner for non-deterministic planning. While this approach has been successful in many domains, the translation may become very large, encumbering the task of the non-deterministic planner.

In this paper we suggest a different approach — using an online contingent solver repeatedly to construct a plan tree. We execute the plan returned by the online solver until the next observation action, and then branch on the possible observed values, and replan for every branch independently. In many cases a plan tree can be exponential in the number of state variables, but still, the tree has a structure that allows us to compactly represent it using a directed graph. We suggest a mechanism for tailoring such a graph that reduces both the computational effort and the storage space. Furthermore, unlike recent state of the art offline planners, our approach is not bounded to a specific class of contingent problems, such as limited problem width, or simple contingent problems. We present a set of experiments, showing our approach to scale better than state of the art offline planners.

Introduction

Agents operating in a partially observable environment gain important information using sensing actions. For example, a robot navigating in a hallway may use its proximity sensors to alert it about the nearby walls. When the agent must achieve some goal, it often takes different actions given different observations it senses. In this case, the agent's decision problem can be modeled as contingent planning under partial observability with sensing actions, and the solution to the problem can be represented as a plan tree, where nodes are labeled by actions, and outgoing edges are labeled by possible observations resulting from the action.

Copyright © 2016, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

The size of the plan tree can be exponential in the number of unknown variables, where each different sequence of observations leads to a different sequence of actions. Thus, offline planners that compute the complete tree often met scaling up difficulties (Albore, Palacios, and Geffner 2009; Bryce, Kambhampati, and Smith 2006). Recently, Muise, Belle, and McIlraith (2014) translated a partially observable problem into a fully observable non-deterministic planning (Bonet and Geffner 2011), and then ran the non-deterministic planner PRP. A disadvantage of this approach is that the translation either becomes large as the problem size increases, or is restricted to certain classes of problems, such as simple contingent problems, or bounded width.

In this paper we take a different approach. Instead of translating the entire problem into a non-deterministic one, we compute a solution directly over the contingent problem, leveraging an online contingent planner. Online planners avoid computing a complete plan tree offline, by taking into account a concrete state of the system, making decisions throughout the plan execution (Shani and Brafman 2011; Brafman and Shani 2012; Albore, Palacios, and Geffner 2009; Bonet and Geffner 2011). These planners often plan for the next action (or until the next sensing action) observe its output, and then plan again (replan) given the newly obtained information(Maliah et al. 2014). As such, these planners traverse a single branch of a complete plan tree.

Our method uses repeated calls to such an online replanner, capable of receiving the current partially observable state, i.e., the system description and a sequence of actions and observations that were already obtained, and computing a sequence of actions that will lead to the goal, if possible, or to a sensing action, producing more information, otherwise. We begin at the initial state, asking the planner to produce a plan to a sensing action, then call the planner again for each of the possible observations that the sensing action has produced. This process continues until all branches end at goal leaves. Given a sound and complete online replanner, and limiting ourselves to deterministic domains with no deadends, this approach provides a sound and complete contingent planner for computing contingent plan trees.

Plan trees may be exponential, requiring a different sequence of actions for every possible initial state. In many cases, however, the environment allows for a structured solution, allowing us to represent the plan tree more compactly

as a directed acyclic graph, avoiding the repeated computation of identical sub-plans. Indeed, PO-PRP uses the PRP planner which is able to identify such structures, and create plan graphs, which may be more compact than plan trees.

Following their footsteps, we augment our approach using a mechanism to identify branches of the tree where an already computed plan can be applied. Given useful problem structure, this mechanism helps us to create a directed acyclic graph representation rather than a plan tree, avoiding an explicit exploration of an exponential number of branches. This mechanism is applicable only for the limited class of simple contingent problems, yet our general approach is applicable to all contingent problems.

We experiment with well known contingent benchmarks. For simple benchmarks that allow for structured solutions, our planner scales well beyond the reach of state-of-the-art planners. In other cases our planner may be significantly slower, but does not suffer as much from memory consumption problems. We further show that on non-simple problems, although we compute a plan tree rather than a graph, we scale much better than previous approaches.

The main contribution of this paper is in constructing contingent plan trees using repeated calls to an online contingent replanner after every sensing action. A second contribution is the identification of plan structure, allowing a compact plan graph representation instead of a plan tree.

Background

Partially observable contingent planning problems are characterized by uncertainty about the initial state of the world, partial observability, and the existence of sensing actions. Actions may be non-deterministic, but much of the literature focuses on deterministic actions, and in this paper we will assume deterministic actions, too.

Problem Definition

A contingent planning problem is a quadruple: $\pi = \langle P, A, \varphi_I, G\rangle$. P is a set of propositions, A is a set of actions, φ_I is a formula over P that describes the set of possible initial states, and $G \subset P$ is the goal propositions. We often abuse notation, treating a set of literals as a conjunction of the literals in the set, as well as an assignment of the propositions in it. For example, $\{p, \neg q\}$ will also be treated as $p \wedge \neg q$ and as an assignment of *true* to p and *false* to q.

A state of the world, s, assigns a truth value to all elements of P. A *belief-state* is a set of possible states, and the initial belief state, $b_I = \{s : s \models \varphi_I\}$ defines the set of states that are possible initially. An action $a \in A$ is a three-tuple, $\{$*pre*(a),*effects*(a),*obs*$(a)\}$. *pre*(a) is a set of literals denoting the action's preconditions. *effects*(a) is a set of pairs (c, e) denoting conditional effects, where c is a set (conjunction) of literals and e is a single literal. Finally, *obs*(a) is a set of propositions, denoting those propositions whose value is observed when a is executed. We assume that a is well defined, that is, if $(c, e) \in$ *effects*(a) then $c \wedge$ *pre*(a) is consistent, and that if both $(c, e), (c', e') \in$ *effects*(a) and $s \models c \wedge c'$ for some state s then $e \wedge e'$ is consistent. In current benchmark problems, either the set *effects* or the set *obs* are empty. That is, actions either alter the state of the world but provide no information, or they are pure sensing actions that do not alter the state of the world, but this is not a mandatory limitation.

We use $a(s)$ to denote the state that is obtained when a is executed in state s. If s does not satisfy all literals in *pre*(a), then $a(s)$ is undefined. Otherwise, $a(s)$ assigns to each proposition p the same value as s, unless there exists a pair $(c, e) \in$ *effects*(a) such that $s \models c$ and e assigns p a different value than s. We assume throughout that all observations are deterministic and accurate, and reflect the state of the world *prior* to the execution of the action. Thus, if $p \in$*obs*(a) then following the execution of a, the agent will observe p if p holds now, and otherwise it will observe $\neg p$. Thus, if s is the true state of the world, and b is the current belief state of the agent, then $b_{a,s}$, the belief state following the execution of a in state s — $b_{a,s} = \{a(s')|s' \in b, s'$ and s agree on *obs*$(a)\}$ — corresponds to the progression through a of all states in the old belief state b that assign the propositions in *obs*(a) the same values as s does.

A contingent problem is simple if the hidden propositions are static, i.e., do not change throughout the execution, and no hidden variable appears in the condition c of a conditional effect (c, e) of any action. Simple contingent problems are easier to solve (Bonet and Geffner 2011).

Contingent Plans

A plan for a contingent planning problem is an annotated tree $\tau = (N, E)$. The nodes, N, are labeled with actions, and the edges, E, are labeled with observations. A node labeled by an action with no observations has a single child, and the edge leading to it is labeled by the null observation *true*. Otherwise, each node has one child for each possible observation value. The edge leading to this child is labeled by the corresponding observation.

A belief state $n.b$ can be associated with each node n in the tree, i.e., the set of possible states when n is reached during run-time. b_I is the belief state associated with the root. If $n.b$ is the belief state associated with node n labeled by a, and n' is a child of n connected with an edge labeled by φ, then the belief state $n'.b$ associated with n' will be $\{a(s) : s \in b, s \models \varphi\}$.

We illustrate this using a 4×4 Wumpus domain (Albore, Palacios, and Geffner 2009). Figure 1(a) illustrates this domain, where an agent is located on a 4×4 grid. The agent can move in all four directions, and if moving into a wall, it remains in place. The agent is initially in the low-left corner and must reach the top-right corner. There are two monsters called Wumpuses hidden along the grid diagonal, the agent knows that each Wumpus is hiding in one of two possible locations, but must observe its stench, which carries to all adjacent locations, in order to deduce its whereabouts. The possible states can be characterized by the whereabouts of the Wumpuses — in both lower locations (denoted *dd* for down-down), in both upper locations (denoted *uu* for up-up), lower location first and then upper location (*du*), and upper location first and then lower location (*ud*). Figure 1(b) shows a possible plan tree for the Wumpus domain.

A contingent plan may be exponential in the number of hidden state variables, requiring a different branch for every

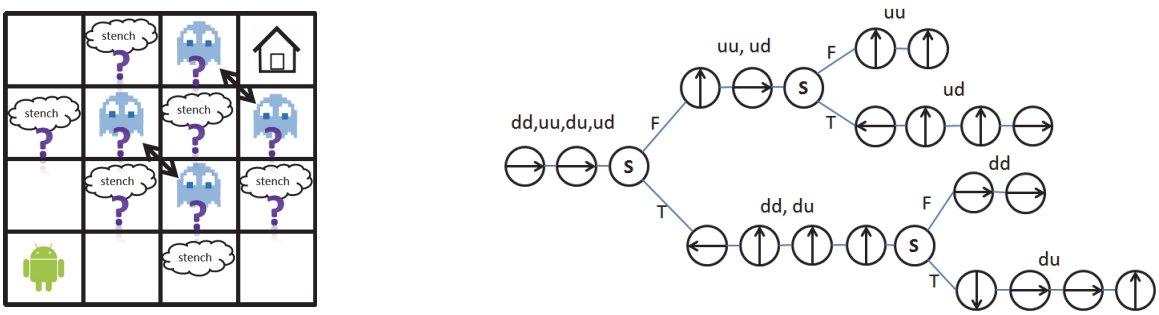

Figure 1: The 4×4 Wumpus domain. A plan tree, with arrows denoting movement actions, *S* for sensing a stench, and outgoing edges marked by T or F (stench was observed or not). Branches are associated with a belief — the set of possible states.

possible start state. Indeed, classical planning problems may also have an exponential solution, but in the contingent case this is not a theoretical worst case, but rather a feature of many well-known benchmark domains, such as the Wumpus plan tree that we present above. Moreover, it may be that although there is a polynomial path to the goal given any start state, the complete plan tree is exponential in the number of hidden variables. This is the main advantage of online solvers that traverse only a polynomial single branch of the tree, avoiding the exponential plan representation.

Online Contingent Replanning

Often, the agent must respond rapidly to new observations, and one must pre-compute a complete plan tree before starting to act in the world. In some real world problems, however, it is acceptable for the agent to pause acting at an arbitrary time to compute its future actions. In such cases one can take an approach where the agent does not compute a complete plan before acting ("offline") but instead repeatedly pauses to recompute plan fragments ("online").

A branch in a contingent plan can be viewed as a sequence of sensing actions, separated by sequences of non-sensing actions. In the Wumpus example, each branch is a sequence of smell actions, separated by sequences of move actions. A common strategy to online planning is to compute a sequence of non-sensing actions that will lead to the next sensing action (Maliah et al. 2014). Following the sensing action the agent receives an observation that modifies its knowledge of the world, and needs to replan — compute a new plan given the new observation, that would either reach the goal, or allow it to execute a new sensing action, obtaining additional useful information. This strategy is known as online replanning, and is key to most state-of-the-art contingent planners (Albore, Palacios, and Geffner 2009; Bonet and Geffner 2011; Shani and Brafman 2011; Brafman and Shani 2012; Maliah et al. 2014).

In this paper we use a contingent replanner as a subroutine. Hence, we require a contingent replanner that can take as input the current knowledge of the agent concerning the state of the world, and produce a sequence of actions that would either reach the goal, or achieve the preconditions of a sensing action that provides new information.

We choose here SDR (Shani and Brafman 2011) for the replanning subroutine. SDR operates by choosing a possible current state s, assuming it to be the true state of the system. Then, SDR plans to reach the goal as if s was the true state. If SDR learns, as a result of a sensing action, that s is not the true state, it chooses a new state and replans.

An online planner is complete if it produces a plan leading to the goal (through multiple replanning episodes) for every valid sequence of observations. Shani and Brafman show that for deterministic domains without deadends, SDR is sound and complete, as in every replanning iteration it eliminates at least one possible state. Eventually only one possible state remains, and one could use a sound and complete classical planner. In the presence of deadends, an online algorithm may make an early decision from which it cannot recover. There are currently no known sound and complete online solvers for contingent planning with deadends.

Belief Maintenance

(Brafman and Shani 2014) suggest to avoid the computation of new formulas representing the updated belief, by maintaining only the initial belief formula, and the history — the sequence of executed actions and sensed observations. When the agent needs to query whether the preconditions of an action or the goal hold at the current node, the formula is regressed through the action-observation sequence back towards the initial belief. Then, one can apply SAT queries to check whether the query formula holds.

In addition, they maintain a cached list of facts $F(n)$ that are known to hold at node n, given the action effects or observations. All propositions p such that $p \notin F(n) \wedge \neg p \notin F(n)$ are said to be unknown at n. The cached list is useful for simplifying future regressed formulas.

Contingent Planning using Online Replanning

We now describe our CPOR (Contingent Planning using Online Replanning) algorithm, which uses repeated calls to a contingent replanner to construct a plan tree. We then describe a mechanism for identifying nodes from which an already computed solution can apply.

Algorithm 1 describes CPOR. The algorithm computes a plan tree, represented as nodes labeled by actions, and edges labeled by observations resulting from executing the node

actions. Each node is also associated with a belief, using the regression mechanism (Brafman and Shani 2014).

We begin by initializing the root node, with the associated initial belief state of the world. The initial node is inserted into a stack. At each iteration the algorithm pops a node n out of the stack. If n has not yet been handled (the mechanism for identifying already handled nodes is described later), then we call the online replanning subroutine. This results in a sequence of actions, starting at the belief state associated with n, that either reaches the goal, or ends with a sensing action. This plan is then simulated over the environment. If the plan ends with a sensing actions, the children of the sensing action are pushed onto the stack. This process explores the contingent plan tree in a DFS manner, until the stack is empty.

The algorithm uses a data structure for representing the plan tree nodes. Each node maintains: $n.h$: the action-observation history sequence leading to n, $n.a$: the action to execute at n, $n.children$: single child for non-observation actions, and multiple children for observation actions, $n.parents$: its parent nodes (for plan graphs), and $n.closed$: denoting whether all paths from n to the goal were already computed, that is, whether n satisfies the goal, or all child nodes of n are closed.

We write $a(n)$ to denote the execution of a in the belief state (maintained using b_I and $n.h$) associated with a node n. The result is either a single node n' in the case of non-sensing actions where $n'.h = n.h + a$, or multiple nodes N' where each node $n' \in N'$ corresponds to a single observation o and $n'.h = n.h + ao$.

During plan computation goal conditions are checked using the regression belief mechanism. In addition, as certain online replanners (Shani and Brafman 2011; Brafman and Shani 2012) may include an inapplicable action in their plans, due to sampling, we also use regression to ensure than all action preconditions are met prior to executing them. If this validation process fails, the replanner is called again on the current node. For ease of exposition this validation phase is not presented in Algorithm 1.

We call a node closed when a plan for this node was already computed. For non-simple problems, GetClosedNode simply checks whether n is a goal state, and UpdateClosedNodes does nothing. We later suggest a different implementation for GetClosedNode and UpdateClosedNodes which allows us to create plan graphs for simple problems.

CPOR is trivially sound and complete — its soundness flows naturally from the soundness of the belief maintenance mechanism, ensuring that actions are executed only when their preconditions are met, and that goal conditions are properly verified. Its completeness stems from the completeness of the online replanner. Every branch in the plan tree that we compute corresponds to a single execution of an online planner with multiple replanning episodes. A complete online planner, which reaches a goal given any possible sequence of observations, will produce branches that end at goal nodes. Currently complete online replanners exist only for deterministic problems with no deadends, and as such, CPOR is currently complete only for such domains.

Algorithm 1 CPOR

```
S ← the empty stack
n_0 ← the plan tree node corresponding to b_I
S.Push(n_0)
while S not empty do
  n = S.Pop
  n' ←GetClosedNode(n)
  if n' ≠ ⊥ then
    Replace n with n' in n.parent
    UpdateClosedNodes(n')
  else
    plan ← OnlinePlan(n)
    for Action a ∈ plan do
      n.a ← a
      if a is a sensing action then
        Push all nodes of a(n) into S
      else
        n ← a(n)
      end if
    end for
  end if
end while
return n_0
```

Identifying Closed Nodes in Simple Domains

While CPOR is sound and complete, it can be inefficient. This is because plan trees are often exponential in the number of hidden state variables. Still, in many domains the plans can be compactly represented by directed graphs, reusing many already computed portions of a plan. To construct such a graph we must be able to determine whether there is an already closed node n for which a plan graph has been computed, which is also applicable for the currently explored node n'. If such a node n exists, we can simply link the parent of n' directly to n and stop the tree expansion. In general, identifying whether a plan for node n' is applicable for node n is difficult. Of course, one can simply try to execute the plan in n', validating action preconditions and goal conditions at the leaves, but this process can be time consuming. We now describe a more efficient mechanism for simple domains.

Algorithm 2 identifies whether a plan for a given node n already exists. First, for goal nodes there is always a plan because the null plan applies in all goal nodes. Next, we check for a closed node n' whose plan is applicable for n. Intuitively, the plan of n' is applicable at n, if all the preconditions of the actions in the plan will be applicable when executed from n, and the plan achieves the goal. We call these the relevant facts for the execution of the plan.

After a plan graph is computed for a node n', we regress the goal and plan preconditions backwards from the goal leaves towards n'. Some of these relevant facts are already known at n' (denote $n'.known \subseteq F(n)$), and some of them are revealed only while executing the plan rooted at n' (denoted $n'.hidden$). In simple domains, known variables cannot become unknown. Thus, the known variables at n will remain known throughout the execution of the plan. When regressing back from the goal, however, a known precondition p may become unknown. This happens when one ob-

Algorithm 2 GetClosedNode(n)

if n is a goal state **then**
  return n
end if
for $n' \in C$ **do**
  $match \leftarrow false$
  if $n'.known \subseteq F(n) \wedge \forall p \in n'.hidden, p \notin F(n) \wedge \neg p \notin F(n)$ **then**
    $match \leftarrow true$
    for $p \in n'.hidden$ **do**
      for $(o_{i,1}, ..., o_{i,k_i}) \in R_{p,n'}$ **do**
        if $(o_{i,1}, ..., o_{i,k_i}) \rightarrow p$ does not hold in n **then**
          $match \leftarrow false$
        end if
      end for
    end for
    if $match = true$ **then**
      return n'
    end if
  end if
end for
return $\perp$

Algorithm 3 UpdateClosedNodes(n)

if $n.closed = true$ **then**
  return
end if
if n is goal state **then**
  $n.known \leftarrow G$, $n.hidden \leftarrow \emptyset$
  $n.closed \leftarrow true$
else
  if All child nodes of n are closed **then**
    $n.hidden \leftarrow \bigcup_{n_c \in n.children} n_c.hidden$
    $n.known \leftarrow \bigcup_{n_c \in n.children} n_c.known$
    $\forall p \in n.hidden, R_{p,n} = \bigcup_{n_c \in n.children} R_{p,n_c}$
    if $n.a$ is a sensing action with observation o **then**
      $\forall p \in n.hidden$ s.t. o is relevant to p, add o to all observation sequences in $R_{p,n}$
    end if
    for p s.t. $p \in n.known \wedge \neg p \in n.known$ **do**
      Remove $p, \neg p$ from $n.known$
      Add p to $n.hidden$
      Add the empty sequence to $R_{n,p}$
    end for
    $n.closed \leftarrow true$
  else
    return
  end if
end if
if $n.parent \neq \perp$ **then**
  $\forall n' \in n.parents$ UpdateClosedNodes(n')
end if

servation of a sensing action results in concluding p, while another observation results in concluding $\neg p$, that is, p holds in one branch of the plan from n, while in another branch $\neg p$ holds. When regressing, by using the set $F(n')$ of known variables at node n', we can easily identify the known and unknown regressed preconditions at n.

For the plan of n' to hold for n, all facts in $n'.known$ must also be known at n, and all the facts in $n'.hidden$ must also be unknown at n. However, only checking $n'.known$ and $n'.unknown$ is insufficient. Some facts in $n'.unknown$ would become known during the plan execution not following a direct observation of their value, but rather by reasoning about their value from other observations. For example, in Wumpus, one never directly observes the location of the Wumpus, but rather reasons about its whereabouts given stench observations in nearby cells.

One must also ensure that the reasoning made throughout the execution of the plan starting at n' will also hold if the plan is executed at n. Hence, for each relevant hidden fact p, we maintain the set of observations leading to the reasoning that p holds in the context of the plan. As the plan may require p to hold in a number of branches, there can be different sequences of observations leading to reasoning that p holds. We hence maintain a set $R_{n,p} = \{< o_{1,1}, ..., o_{1,k_1} > , ..., < o_{m,1}, ..., o_{m,k_m} >\}$, where each $< o_{i,1}, ..., o_{i,k_i} >$ is a set of observations leading to reasoning that p holds.

These sets can be restricted to contain only observations relevant to p. In simple contingent problems, an observation over a variable p' is relevant for a hidden variable p if: (1) $p' = \neg p$, (2) p' and p both appear in a clause in the initial state formula, or (3) p' appears in a clause in the initial belief together with another variable which is relevant for p (transitive closure). The set of relevant variables for p is often much smaller than the entire set of observations, although in domains such as Wumpus, all observations are relevant.

For a hidden relevant fact $p \in n'.hidden$, we check whether given the observations of every observation sequence in $R_{n,p}$, we can conclude that p holds. We do so by regressing $(o_{i,1} \wedge ... \wedge o_{i,k_i}) \rightarrow p$ through $n.h$, checking its consistency with the initial belief. This step can only be applied in simple domains, where the hidden variables are static and do not change. While there can be an exponential number of such regression queries, in many domains, given the restriction to relevant variables only, this validation phase is very fast.

The identification of closed nodes is sound and complete for simple contingent problems. It is sound because we verify that all known regressed preconditions hold, and all hidden preconditions can be reasoned about given the observation sequences. It is complete because a node that does not meet our matching criterion, fails because either some known precondition does not hold, and thus some action in the plan will not be applicable, or because some sequence of observations does not entail a required precondition p, which will also cause some action along one branch in the tree, corresponding to those observations, to be inapplicable.

Once a closed node n has been identified, we compute its data structures — $n.known$, $n.hidden$, $R_{n,p}$. Furthermore, when a node in the tree has only closed children it is also closed. Algorithm 3 illustrates the process of updating the closed node and its parents. When identifying that the parent of the closed node is also closed, the process updates also the parent, and so forth, until the root node has been reached, or a parent with at least one unclosed child.

Table 1: Comparing CPOR to CLG and PO-PRP (best reported variant). Doors, CTP, and Wumpus are simple.

	Time			Size		
Problem	CPOR	CLG	PO-PRP	CPOR	CLG	PO-PRP
Doors-7	1.73	3.5	0.04	105	2492	770
Doors-9	3.25	187.6	1.07	181	50961	28442
Doors-11	5.68	FAIL	FAIL	277	FAIL	FAIL
Doors-13	9.37	FAIL	FAIL	393	FAIL	FAIL
Doors-15	15.8	FAIL	FAIL	529	FAIL	FAIL
ctp-10	0.95	2.2	0.02	32	4093	31
ctp-15	1.43	133.24	0.07	47	131069	46
ctp-20	2.01	FAIL	0.22	62	FAIL	61
ctp-25	2.52	FAIL	0.55	77	FAIL	76
ctp-35	3.94	FAIL	2.2	107	FAIL	106
ctp-50	6.74	FAIL	12.3	152	FAIL	151
ctp-100	33.45	FAIL	449.1	302	FAIL	301
Wumpus 5	2.19	0.44	0.14	102	854	233
Wumpus 7	16.9	9.28	1.14	900	7423	770
Wumpus 10	157.1	1379.6	7.56	3971	362615	2669
Wumpus 15	FAIL	FAIL	51.06		FAIL	15628
Wumpus 20	FAIL	FAIL	FAIL		FAIL	FAIL
localize5	3.8	0.72	X	121	137	X
localize7	10.8	3.80	X	266	314	X
localize9	23.9	17.96	X	542	602	X
localize11	51.5	FAIL	X	712	FAIL	X
localize13	123.5	FAIL	X	1194	FAIL	X
RockSample 4,3	1.6	X	X	54	X	X
RockSample 8,3	2.46	X	X	81	X	X
RockSample 8,5	13.13	X	X	281	X	X
RockSample 8,7	71.19	X	X	1563	X	X

Experiments

We now provide a set of experiments comparing CPOR to state of the art contingent planners, CLG (Albore, Palacios, and Geffner 2009) and PO-PRP (Muise, Belle, and McIlraith 2014) on several benchmarks, both simple and non-simple. The statistics for CLG and PO-PRP are taken from the corresponding papers. CPOR is implemented in C#, and the experiments were executed on a Windows 7 machine with 16GB of RAM, and an i7 Intel CPU. CPOR uses the node matching algorithm for the simple domains only and SDR as the online replanner.

Table 1 compares CPOR, CLG, and PO-PRP. As can be seen, CPOR is typically faster than CLG for larger problems, but sometimes slower than PO-PRP. On the other hand, CPOR has very minor memory requirements, allowing it to scale to much larger domains. In our experiments, CPOR never exceeded 100MB of memory. In domains where the plan tree has a very strong structure in that many identical subtrees appear in different parts of the plan tree, such as doors and ctp, CPOR easily scales to very large problem sizes. In domains where the tree is less structured, that is, the plan graph has an exponential size, such as Wumpus, CPOR is less effective. CPOR created much smaller policies on the doors benchmark, almost identical plan tree size for ctp problems, and somewhat larger policies for the Wumpus domains, which require the most complex solutions. In the larger Wumpus domains, although CPOR had no memory issues, it did not terminate within 2 hours. Looking more closely at the time spent on the various components of CPOR, we see that the classical planner requires almost 99% of the runtime (tested both on Wumpus and Doors instances). Thus, the identification of closed nodes is not a bottleneck in our implementation.

The localize domains are the non-simple. PO-PRP is restricted to simple problems, and cannot be run on localize. On these domains, CPOR scales much better than CLG. The RockSample domains are non-simple, and also have problem width higher than 1, making CLG inapplicable. For these domains only CPOR is applicable. This domain also exhibits the exponential growth in tree size and runtime.

Comparing CPOR to PO-PRP is interesting. PO-PRP is built on the PRP planner (Muise, McIlraith, and Beck 2012), which uses a similar approach to CPOR, running an underlying classical planner on every possible child of a non-deterministic action. Non-deterministic planning is in general much harder than deterministic contingent planning, which contains no cycles in its plan graph. Thus, the task of identifying already explored parts of the graph, and tailoring the plan graph is probably more difficult in non-deterministic planning. It might thus be, that using a general purpose non-deterministic planner is perhaps an overkill for contingent planning, and a specially created search algorithm can work better, even on the translation. On the other hand, for non-deterministic contingent problems, CPOR cannot be easily adjusted to produce finite plan trees or graphs. PO-PRP, however, can probably handle non-determinism because its underlying planner PRP is designed for non-determinism. By slightly modifying its translation to allow for non-deterministic effects, PO-PRP may provide very good plan graphs and scale to similar problem sizes.

PO-PRP and CPOR construct different policies. First, the construction techniques are different – PRP is designed for fully observable planning, and maintains no data structures for knowns, unknowns, and reasoning, as we do. Even over the translation, PRP uses node matching rules that are more restrictive than us, due to the need to capture cycles. As such, our method is more general, identifying identical nodes that PRP would not identify. This is obvious from the policy size results on doors. Of course, our approach is not applicable without significant modifications to the case of non-determinism and cycles.

The above results shed some light on the limitations of computing complete plan trees offline. While a plan computed offline allows the agent to reduce the computational resources needed online, and to respond faster to observations, in some cases even the minimal offline plan has an exponential size. In such cases any effort to compute complete plans is futile, and we have no choice but to resort to online planning. We suspect that this is the case for the Wumpus domain. Even if we can't prove that the minimal plan graph for Wumpus is exponential in the grid size, the observed growth in runtime and size certainly hints to that. When exploring offline contingent planning, it is perhaps best to avoid such domains, and focus our attention on domains that allow for non-exponential conditional plan graphs. In order to further develop offline contingent planning, it is needed to identify more such interesting benchmark domain.

Conclusion

In this paper we suggested to repeatedly use an online replanner for creating complete contingent plan graphs. In addition, we suggest a mechanism for simple contingent problems, allowing us to identify already computed plans that are suitable for the current node, thus creating a plan graph, rather than a complete tree. We experiment with several

benchmark domains, both simple and non-simple, showing CPOR to produce smaller policies, and scale to larger problems, when the solution plan tree exhibits structure.

In the future we can explore more efficient mechanisms for identifying applicable plans, and methods for forcing the plan graphs to provide more structure that can later be exploited, encouraging the online planner to reach already explored nodes.

Acknowledgments

This work was supported by ISF Grant 933/13. We thank Chrisitan Muise for his generous help with the PO-PRP planner.

References

Albore, A.; Palacios, H.; and Geffner, H. 2009. A translation-based approach to contingent planning. In *IJCAI*, 1623–1628.

Bonet, B., and Geffner, H. 2011. Planning under partial observability by classical replanning: Theory and experiments. In *IJCAI*, 1936–1941.

Brafman, R. I., and Shani, G. 2012. A multi-path compilation approach to contingent planning. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Brafman, R. I., and Shani, G. 2014. On the properties of belief tracking for online contingent planning using regression. In *ECAI 2014 - 21st European Conference on Artificial Intelligence*, 147–152.

Bryce, D.; Kambhampati, S.; and Smith, D. E. 2006. Planning graph heuristics for belief space search. *JOURNAL OF AI RESEARCH* 26:35–99.

Maliah, S.; Brafman, R. I.; Karpas, E.; and Shani, G. 2014. Partially observable online contingent planning using landmark heuristics. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS*.

Muise, C. J.; Belle, V.; and McIlraith, S. A. 2014. Computing contingent plans via fully observable non-deterministic planning. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*.

Muise, C. J.; McIlraith, S. A.; and Beck, J. C. 2012. Improved non-deterministic planning by exploiting state relevance. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS*.

Shani, G., and Brafman, R. I. 2011. Replanning in domains with partial information and sensing actions. In *IJCAI*, 2021–2026.