Plan Critiquing for Assessing Adversarial Effects on Plans

Ian Kariniemi, Ugur Kuter

SIFT,LLC
{ikariniemi, ukuter}@sift.net

Abstract

The execution of plans in practical applications are always subject to adversarial conditions. Sometimes these conditions are intentional, e.g., non-collaborative agents are attempting to fail the plan being executed. Other times they are not, e.g., weather conditions or even the actions of our collaborators. Building on an existing counter-planning and plan-critiquing framework, called Murphy, this paper introduces our capabilities to help assess adversarial effects on plans. We describe the original Murphy framework, our extensions to Murphy to support adversarial reasoning, and our example use cases demonstrating our proposed approach. We conclude with a discussion on the challenges of this problem and our planned future work.

1 Introduction

Most automated planning systems presume that the system will have primary responsibility for constructing a plan, that when executed achieves the goals. In practical applications such as collaborative teamwork planning, logistics planning, air vehicle planning and scheduling, intrusion detection problems, cybersecurity, and others, this classical premise does not hold: the system that is executing plans encounters uncontrolled conditions that break the generated plans. Furthermore, even if the automated planners can generate and execute plans, often the responsibility of planning still remains with human planners. This can be because the automated system lacks the ability to handle the complexity of a real world situation, or simply because the human planners do not fully trust the system with the full responsibility of planning. Therefore, the system must be able to explain the adversarial conditions it encountered to the users both during planning and execution.

Traditional conditional or contingency planning approaches (Barish and Knoblock 2002; Pryor and Collin 1996; Anderson, Smith, and Weld 1998), albeit being theoretically relevant, do not address these challenges in practical domains. Using an adversarial model in the context of plan improvement has also been discussed before in the literature, including in domains such as game-playing (Willmott et al. 2001), and goal-driven approaches that use goal recognition for adversaries to predict the goals of plans (Jiang, Dong, and Zhao 2006; Pozanco et al. 2018). Unlike these approaches, Murphy identifies breakdown conditions that are not only in the goals but also throughout the execution of plan.

There are also adversarial models that consider cost, such as the work in "Planning in the Presence of Cost functions controlled by an adversary" (McMahan, Gordon, and Blum 2003). This work considers a scenario in which the adversary controls the cost functions of a problem after a policy is fixed in a Markov Decision Process. Murphy does not use a Markov Decision Process and considers cost functions in a problem to be fixed.

In this paper, we briefly describe Murphy, an existing counter-planning and plan-critiquing framework (Goldman, Kuter, and Schneider 2012) that was designed for generating counter examples to show how plans might be broken via the effects of uncontrolled actions and plans. We introduce Hokage (Kuter and Goldman 2017) as the counter-example generation engine in Murphy. Previously, Murphy leveraged classical heuristic planners such as FF (Hoffmann 2001). Hokage is a heuristic planner that uses the same theorem-proving capabilities, such as axiomatic reasoning, numeric reasoning, and expressional function calls as SHOP3, even if it is a non-hierarchical planner.

We then describe our approach to extend the adversarial reasoning capabilities in Murphy. This includes a skolemization technique that allows counter-example generation to perform delayed committed during the process. Unlike most planners that eagerly bind variables, including lifted planners such as SHOP3 (Goldman and Kuter 2019), delayed commitments allows Murphy to reason about uncertainty and ambiguities in adversarial conditions and behaviors. We also describe a method of probabilistic (cost-based) reasoning over adversarial conditions, providing meaningful comparisons of the value of a particular counterexample.

We conclude with a short walkthrough of the approach in a suite of example use cases in an abstract collaborative multi-agent domain. We discuss our future work directions in this approach.

2 Background

We use the same definitions and notation for constant and variable symbols, literals, ground atoms, state, planning operators and actions, plans, planning problems and solutions to those problems as in classical planning (Ghallab, Nau, and Traverso 2004).

Murphy is a counter-planning and plan-critiquing framework (Goldman, Kuter, and Schneider 2012), which aims to provide feedback for plans on how they might fail, and can be applied to plans made by humans or plans that are generated by planning systems. Murphy originally only supported problems formulated with the Planning Domain Description Language (PDDL) (Ghallab et al. 1998), but we have now added support for problems formulated in the SHOP3 formalism (Nau et al. 2003), excluding the portions of the formalism that define HTN methods and HTN tasks.

We have adapted the subsequent definitions from (Goldman, Kuter, and Schneider 2012) for the sake of completeness in this paper. A counterplanning problem for a solved planning problem, $<P, \pi>$ (We now refer to a counter planning problem with respect to the plan, π) is defined as $C = <P, \pi, L, c_0, A>$. L is a set containing positive literals that is a superset of the literals of P as above, and $c_0 \supseteq s_0$ is an augmented initial state, a superset of the initial state. We augment them to capture the state of agents outside the control of the original planner. $A = U \cup O$ is the set of operators of the counter planning problem, where U is a set of uncontrolled operators modeled in the same way as the operators O. The set of operators between U and O differ such that $U \cap O = \emptyset$.

A counterexample for π wrt C is a sequence of uncontrolled actions, Π, that is an interleaving of π with some uncontrolled actions π–$\Pi = \hat{\pi} \mid \hat{\pi}$– containing some controllable α such that $\Pi = \Pi'\alpha\Pi''$ where Π' is executable, but $\alpha, result(\Pi')$ is not executable $(result(c_0, \Pi') \not\models pre(\alpha))$. A solution for C is a counterexample if one exists, or the empty set if one does not exist.

3 Approach

Murphy takes as input (1) a planning domain D, (2) a planning problem P, (3) a plan that is a solution for P, and (4) a counter planning domain definition with uncontrolled objects and operators. The system translates these inputs into a counter-planning problem specification and a constraint counter planning domain. Informally, as described in (Goldman, Kuter, and Schneider 2012), Murphy translates a plan into a "counter planning" problem, combining a representation of the plan with the definitions of a set of uncontrolled actions. These uncontrolled actions could include the actions of other agents in the environment, the weather, or even the actions of controlled agents that conflict with each other. The resulting translation creates a planning problem for a classical planner to solve, for which the solution provided is a *counterexample* to the initial plan. For more details on Murphy see (Goldman, Kuter, and Schneider 2012). In the subsequent subsections, we describe our recent extensions to Murphy.

Cost of Disruptions

Originally, Murphy lacked depth in the comparison of counterexamples. The intuitive comparison is to note that counterexamples that break a plan beyond repair are "better" (for the purposes of plan critiquing) than ones that require only a slight change to the plan to accomplish the goal of the original plan.

As a tool for making plans robust to adversarial effects, it is in the interest for Murphy to work towards searching for the most devastating break (for the original agent):

1. Take a counterexample and apply all of its actions but the final action to the counterdomain, essentially creating the state that occurs just before the action in the original plan, with the assumptions behind the application of the next action broken (The final action in a counterexample is a pseudo-action that represents a break, but doesn't alter the world state except to add the goal predicate that the classical planner is looking for)
2. Take all the predicates of that state and use it as the initial state of a new planning problem.
3. Send the new planning problem to a classical planner (currently only Hokage at the moment) with the original goal being assigned as the goal of the new problem.
4. If the planner returns a new plan, then we find the difference in cost between the old plan and the new plan, showing us how much cost the counterexample adds to completion of the plan.
5. If no new plan is generated, then that means the counterexample breaks the plan completely.

We could consider the cost in absolute terms, and rank counterexample severity on the basis of cost alone, assuming the most powerful adversary possible. For uncontrolled actions that concern weather events, this sort of modeling might be enough. However, it might also be interesting to model an adversary as rational, and consider cost of the adversary in executing the counterexample as well as the cost that the disruption creates for the original planner. This behavior is well supported by SHOP3 cost functions, which can call Lisp functions to calculate cost and use bound values of variables in these calls.

Hokage

Murphy in the past worked only with PDDL planners such as FF (Hoffmann 2001) or FastDownward (Helmert 2006). We have extended it for the Hokage planner (Kuter and Goldman 2017). Hokage is a planning algorithm that performs lifted search for generating a solution plan. It uses a lifted relaxed planning graph as a heuristic computation in a best-first search algorithm.

Although Hokage is not a hierarchical planner, it is designed and implemented on the theorem-proving facilities of the SHOP3 planner (Goldman and Kuter 2019; Nau et al. 2003). As such, Hokage is capable of scoped axiomatic reasoning, numeric reasoning, and external function calls during its heuristic search. This is important for generating expressive and practical counter examples for adversarial reasoning.

For example, axioms in SHOP3 are analogous to Horn clauses in Prolog and derived predicates in some PDDL planners (e.g., (Gerevini, Saetti, and Serina 2003)) and during search in Hokage's relaxed states, we can calculate dependencies for axioms, linking the predicates that are required to be true for an axiom to be true to the action for which the axiom is a precondition.

```
(:- (within-storm-radius ?truck ?storm-id)
  (and
    (at ?truck ?truck-loc-lat ?truck-loc-
        long)
    (storm-center ?storm-id ?center-loc-lat
        ?center-loc-long)
    (assign ?distance-km
      (/ (sqrt (+ (expt (- '?truck-loc-lat
          '?center-loc-lat) 2)
                  (expt (- '?truck-loc-long
                      '?center-loc-long) 2))
                  )
      111.0))
    (eval (<= distance-km 5.0))))
```

The above axiom checks checks whether a truck is within the radius of a particular storm, which we approximate as a circle centered on the radius. It uses the first two portions of the conjunction to bind values for the locations of the truck and storm center, then uses `assign` to calculate the euclidean distance between the two points, assigning the output value to the variable `?distance-km` (converting to kilometers in an approximation of the earth as a sphere). It uses that value in a call to `eval` to see that the distance is within 5 kilometers.

For dependency tracking, we would track dependence of the axiom in the relaxed planning graph by finding where the subpredicates that are required to be true for it to be satisfied are sourced. Notably, this axiom's dependence can be established without considering the `eval` subpredicate, as its only variable `distance-km` is an output of computations with the variables for the location of the truck and the storm, and therefore adds no new dependencies to the axiom. This means that the `eval` predicate in this case requires no additional linking of nodes.

Delayed Binding

As another extension, we have added a skolemization approach to Hokage. Skolemization here refers to the process of removing existential quantifiers by elimination, taking variables that come from existential qualifiers and replacing their usage with skolem functions (Russell and Norvig 2009). Though we do not use existential qualifiers explicitly in our state models, we do some analysis on the states to find means of establishing existential qualifiers in it. We observed that in many cases, plans would have multiple entities for which it was not important which entity was being used, as long as the capability matched the requirements of the plan, providing us an existential qualifier of the capabilities we need.

This skolemization approach has applications for adversarial reasoning, providing at least two benefits to Murphy: (1) it defines *equivalence classes* of planning states for Hokage to search over and solution plans in those equivalence classes that apply to fully-grounded states, and in doing so, (2) it improves the speed of the heuristic search, by enabling counter-example generation over such equivalence classes.

This suggests that Hokage can delay binding a variable symbol in the relaxed planning heuristics; instead, the planner replaces the variables with a skolem function that specifies the properties, as constraints, of the object (i.e., constant symbol) that should be bound to that variable for a sound plan. After Hokage generates a plan with skolem functions in it as a solution, it post-processes the plan and generates variable bindings according to the generated constraints during heuristic search.

There are classes of planning problems in which such post-processing will fail to generate a grounded plan due to the constraints generated during the search, since the planner does not change its decisions on the fly during the search. However, the search constraints can be given as input to the planner to start another search episode for an incremental counter-example generation. We discuss this more in applying this delayed binding to an example problem.

4 Example Use Cases

For our use cases, we have designed a small domain to demonstrate the extensions to expressivity in our plans and the generated counterexamples.

We call this domain *Game of Dragons*. It encompasses a fantasy setup where-in we have dragons, elves and dwarves as agents we control in our domain. The goal of our original agents is often to destroy some number of targets, which could be different castles. Castles might be destroyed by dragons breathing fire on them, dragons dropping rocks on them, or elves casting spells to destroy them. The agents we control can navigate on a grid of cells, which are indexed as integer pairs in our map. Dragons can fly, and therefore are not restricted on which cells to navigate, but elves and dwarves cannot fly. Elves can ride dragons, and provide extra capability to the dragon.

Arithmetic Evaluation

Having arbitrary arithmetic evaluation allows us to describe the grid world succinctly, without having to enumerate all the positions of the grid symbolically. Instead, we have a single predicate `(bounds ?min-x ?min-y ?max-x ?max-y)` that describes the bounding box of the grid (inclusive), and have operators for each direction of movement. So for our domain, we have an operator named `walk-north`, which takes as arguments the goblin and the start location (making a head of `!walk-north ?goblin ?start-loc`).

The precondition of that action is the following.

```
(and
  (bounds ?min-x ?min-y ?max-x ?max-y)
  (goblin ?goblin)
  (at ?goblin ?start-loc)
  (assign ?end-loc (generate-coord-no-bounds
      :north '?start-loc))
  (eval (coord-in-bounds '?end-loc '?min-x
      '?min-y '?max-x '?max-y))
  (not (blocked ?start-loc ?end-loc)))
```

And as a postcondition it adds an atom unifying with `(at ?goblin ?end-loc)` to the state, and deletes an atom unifying with `(at ?goblin ?start-loc)`.

In the above operator, we establish in a conjunction that the goblin is at a particular location, binding that location `?start-loc` as a variable, and use that location to generate the location above it that corresponds

to movement of 1 cell in the northern direction. The call to generate the next coordinate is to a Lisp function `generate-coord-no-bounds`, and we use `assign` to assign the result of that call to the variable `end-loc`. In this way, we have a means of generating the next location without having to enumerate the adjacency relationships between each location. When Hokage is planning with this operator, it understands the locations numerically, and the power of Lisp evaluation gives us tools to generate the next location with that understanding.

Delayed Binding example

Suppose the initial state for a problem is described by the following state atoms.

Listing 1: Initial State

```
((bounds 1 0 2 2)
 (target target1 (2 1) :castle)
 (target target2 (2 2) :castle)

 (dragon dragon1)
 (energy dragon1 6)
 (available dragon1)
 (at dragon1 (2 0))

 (dragon dragon2)
 (energy dragon2 6)
 (available dragon2)
 (at dragon2 (2 0))
 (green-fire dragon2)

 (dragon dragon3)
 (energy dragon3 6)
 (available dragon3)
 (at dragon3 (2 0))
 (green-fire dragon3))
```

And, at `(1 1)`, a goblin lies in wait, with a bow and arrow also at `(1 1)` for them to pick up and fire at the dragon.

During the counterplanning problem setup process, our approach decides to abstract away one of the dragons, as they are both at the same location and have the same properties, producing a new state that delays the choice of which dragon to pick in Listing 2.

Listing 2: Delayed binding version of the State

```
((bounds 1 0 2 2)
 (target target2 (2 2) :castle)
 (target target1 (2 1) :castle)
 (dragon #:skol-dragon1151)
 (at #:skol-dragon1151 (2 0))
 (available #:skol-dragon1151)
 (energy #:skol-dragon1151 6))
```

This state is passed first to Hokage to generate an input plan in Listing 3, then the plan and initial state are passed to Murphy for counterplanning, and the resulting counterplan we get is on that abstracted version, as we see in Listing 3.

Listing 3: Plan on skolemized state

```
((!fly-north #:skol-dragon1151 (2 0))
 (!burn-target #:skol-dragon1151 target1)
 (!fly-north #:skol-dragon1151 (2 1))
 (!burn-target #:skol-dragon1151 target2))
```

Listing 4: Counterexample

```
((((!pick-up-arrow goblin1 (1 1))
  (!pick-up-bow goblin1 (1 1))
  (!fly-north #:skol-dragon1151 (2 0))
  (!burn-target #:skol-dragon1151 target1)
  (!walk-east goblin1 (1 1))
  (!fly-north #:skol-dragon1151 (2 1))
  (!walk-north goblin1 (2 1))
  (!fire-arrow-at-dragon goblin1 #:skol-
     dragon1151 (2 2))
  (!burn-target #:skol-dragon1151 target2))
  ...)
```

However, if we imagine that the first castle is shielded, such that it requires a dragon with green-fire (`dragon2` or `dragon3` fit the bill), we can expand the set of constraints applied to the skolem constant, adding `(green-fire #:skol-dragon1151)` to our starting state so that the search will return a result for the initial plan.

5 Discussion

There are a few directions we are looking at for future work and improvements on counter-planning with Murphy.

Local Minima in Heuristic

During evaluation of Hokage, we noticed that the relaxed planning graph heuristic could get "stuck" in local minima in the search. In our preliminary experiments this tended to appear when there were multiple serializable subgoals that had to be accomplished to make the main goal true. Actions that took the state closer to accomplishing a particular subgoal would have a higher heuristic value (where lower heuristic values are preferred) than actions that took the state to a spot that was the closest to all the subgoals. In FF, these situations would be handled by its fallback to best-first breadth first search (Hoffmann 2001), but this approach is practically-infeasible for our possibly infinitely sized domains.

Critiquing of HTN generated plans

The approach Murphy adopts is currently structured for classical planning, but we believe there might be applications in HTN planning as well. While conventional HTNs do not have preconditions or effects on nonprimitive tasks (Goldman and Kuter 2019), the plan trees that SHOP3 generates could be critiqued in the same way Murphy critiques preconditions in classical planning plans.

Axioms for expanding skolemization function

Because Hokage supports axioms, we are considering using axioms as a means of expanding skolemization functionality. The skolemization process attempts to interpret a collection of entities as a singular one and encoding that singular entity with the shared properties of the collection. Axioms could be a means of encoding different relevant properties, and we are considering using Axioms (which are essentially Horn Clauses) to reason about skolem constants in our models.

Acknowledgments.

This material is based upon work supported by the U.S. Army Combat Capabilities Development Command Avia-

tion & Missile Center under Contract No. W911W6-19-C-0064. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the U.S. Army Combat Capabilities Development Command Aviation & Missile Center. Distribution Statement A: Approved for Public Release PR20220122.

References

Anderson, C.; Smith, D.; and Weld, D. 1998. Conditional effects in Graphplan. 44–53.

Barish, G.; and Knoblock, C. A. 2002. Planning, Executing, Sensing, and Replanning for Information Gathering. In *Proceedings of Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2002)*, 184–193. Menlo Park, CA: AAAI Press.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning through stochastic local search and temporal action graphs in LPG. *Journal of Artificial Intelligence Research*, 20: 239–290.

Ghallab, M.; Knoblock, C.; Wilkins, D.; Barrett, A.; Christianson, D.; Friedman, M.; Kwok, C.; Golden, K.; Penberthy, S.; Smith, D.; Sun, Y.; and Weld, D. 1998. PDDL - The Planning Domain Definition Language.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: theory and practice*. Elsevier.

Goldman, R. P.; and Kuter, U. 2019. Hierarchical Task Network Planning in Common Lisp: the case of SHOP3. In *ELS*, 73–80.

Goldman, R. P.; Kuter, U.; and Schneider, T. 2012. Using classical planners for plan verification and counterexample generation. In *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26: 191–246.

Hoffmann, J. 2001. FF: The fast-forward planning system. *AI magazine*, 22(3): 57–57.

Jiang, H.; Dong, Y.-Q.; and Zhao, Q.-s. 2006. A counterplanning approach based on goal metric in the adversarial Planning. In *2006 International Conference on Machine Learning and Cybernetics*, 78–84. IEEE.

Kuter, U.; and Goldman, R. P. 2017. HOKAGE: A Heuristic Lifted Planning Algorithm for Generating Diverse Generalized Plans. In *Proceedings of the ICAPS-17 Workshop on Generalized Planning*.

McMahan, H. B.; Gordon, G. J.; and Blum, A. 2003. Planning in the presence of cost functions controlled by an adversary. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, 536–543.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *J. Artif. Intell. Res. (JAIR)*, 20: 379–404.

Pozanco, A.; Blanco, A.; E-Martín, Y.; Fernández, S.; and Borrajo, D. 2018. Counterplanning in Real-Time Strategy Games through Goal Recognition. In *Proceedings of Workshop on Goal Reasoning at IJCAI'18*.

Pryor, L.; and Collin, G. 1996. Planning for Contingency: a Decision Based Approach. *J. of Artificial Intelligence Research*, 4: 81–120.

Russell, S.; and Norvig, P. 2009. *Artificial Intelligence: A Modern Approach*. USA: Prentice Hall Press, 3rd edition. ISBN 0136042597.

Willmott, S.; Richardson, J.; Bundy, A.; and Levine, J. 2001. Applying adversarial planning techniques to Go. *Theoretical Computer Science*, 252(1-2): 45–82.