An Analysis of Transformational Analogy: General Framework and Complexity

Vithal Kuchibatla & Héctor Muñoz-Avila

Department of Computer Science and Engineering,
19 Memorial Drive West
Lehigh University
Bethlehem, PA, 18015, USA

Abstract

We present TransUCP, a formalism for Transformational Analogy in the context of classical domain-independent planning. TransUCP defines precisely possible plan modification operations for Transformational Analogy and covers a wide range of existing implementations. We use TransUCP to analyze the implications for Transformational Analogy of well-known results about the complexity of general plan adaptation.

Motivation

In this paper we present a formalism for Transformational Analogy in the context of automated planning. Using this formalism we analyze general results about the complexity of plan adaptation and how they relate to Transformational Analogy. Transformational Analogy is a CBR adaptation technique in which a pre-selected plan, defined as a sequence of actions, is modified to solve a new problem (Carbonell, 1983). Possible modifications to the plan include removing actions, adding new actions, and changing the parameters of actions.

Interest in Transformational Analogy has been recurrent since its inception in Carbonell (1983). It was the adaptation method used in the first case-based planning system, CHEF (Hammond, 1990) and it is an alternative to Derivational Analogy, the other main plan adaptation technique. A major difficulty of using Derivational Analogy is the requirement for the availability of the derivations that led to a solution (Cunningham *et al*., 1996). Perhaps for this reason, application-oriented papers in case-based reasoning that use some form of adaptation most frequently use Transformational Analogy. Yet, despite this interest no general framework for precisely defining and analyzing Transformational Analogy exists to date. A general framework of Derivational Analogy can be found in Au *et al*. (2002).

Our formalism for Transformational Analogy is called TransUCP (Kuchibatla & Munoz-Avila, 2006). TransUCP covers any variant of Transformational Analogy for classical domain-independent planning. To achieve this, TransUCP is built on top of the Universal Classical Planning framework (UCP). TransUCP provides a formalization of Transformational Analogy across a variety of planning paradigms.

We use TransUCP to analyze the implications for Transformational Analogy of the studies about the complexity of general plan adaptation published in Nebel and Koehler (1995). This study concluded that in the worst case, plan adaptation takes exponentially more time than planning from scratch (Nebel & Koehler, 1995). We use our model to show that Transformational Analogy does not satisfy a key assumption, called minimal plan modification, needed for the complexity analysis in Nebel & Koehler (1995). Roughly, a plan adaptation algorithm satisfies the minimal plan modification condition if it is guaranteed to reuse the largest possible number of steps of a given plan when solving a new problem. We empirically illustrate the difficulties that Transformational Analogy has to meet the minimal plan modification in a well-known domain. Since TransUCP covers several Transformational Analogy systems, including those based on partial-order planning (e.g., SPA (Hanks and Weld, 1995) and POPR (Van der Krogt and De Weerdt, 2005)) and those based on total-order planning (e.g., CASER (Tonidandel and Rillo, 2005)), all of which have empirically demonstrated gains over plan generation from the scratch, our work clarifies the difference between the empirical results obtained by these systems and the worst case complexity analysis.

Background

TransUCP expands on the SPA system. SPA is a general purpose algorithm for Transformational Analogy that takes advantage of the partial-order plan representation of partial-order planners to modify an existing plan (Hanks and Weld, 1995). Our general framework enhances SPA to other forms of planning by using UCP.

UCP is an abstract algorithm in which planning is done by making repeated refinements to partial plans, stopping when a complete plan is obtained (Kambhampati & Srivastava, 1996). All known classical planning algorithms are special cases of UCP. In UCP, a *partial plan* consists of a set of steps and a set of constraints that must be satisfied. Due to space limitations we cannot

define partial plans formally. Rather we illustrate the elements of a partial plan with an example in the logistics transportation domain. In this domain, there are different packages located at various places and these packages need to be re-located to specific locations. There are trucks and airplanes that are used to move the packages.

Suppose a problem is given where packages P1 and P2 are in locations A and D, and two trucks V1 and V2 are located at A and D respectively. The two goals are to relocate P1 and P2 to a location C. Figure 1 shows a partial plan solving this problem. Arrows denote either ordering or interval preservation constraints (defined below). The "*" mark between the steps indicates ***contiguity constraints***, whereby no other steps are allowed in between. Steps in the plan are denoted by t_k. Steps t_0 and t_∞ are special steps that we will explain later. The other steps are labeled according to the actions they perform: *L(P,V,Lc)* indicates that the vehicle *V* is loading the package *P* from location *Lc*. *UL(P,V,Lc)* indicates that *P* is unloaded from *V* into location *Lc*, and *MV(V, L1, L2)* indicates that V is moved from *L1* to *L2*. Under these conventions P1 is loaded into V1 and P2 into V2. P2 is relocated to B using V2. V1 moves from A to B to pick P2. V1 continues to C, where both packages are unloaded.

Besides steps, other primary elements of a partial plan are its open conditions and threats (not shown in Figure 1). A condition has the form ($\rightarrow^Q t_k$) indicating that the condition Q has to be satisfied for step t_k. Each step t_j in the plan can produce an effect Q, written $t_j \rightarrow^Q$, which can be used to satisfy conditions. A condition $\rightarrow^Q t_k$ is satisfied by adding an ***interval preservation constraint*** $t_m \rightarrow^Q t_k$, such that $t_m \rightarrow^Q$ holds. If a condition has not been satisfied, it is said to be an open condition. A threat is a 3-tuple (t_i, $t_m \rightarrow^Q t_k$) where t_i can be inserted between t_m and t_k and $t_i \rightarrow^{\neg Q}$

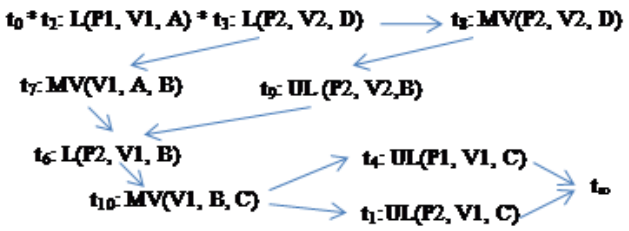

Figure 1: a complete plan

holds. Threats occur as a result of the partial ordering between steps. So for example, the condition Q might use a truck to satisfy a condition in t_k, but another step t_i might use the same truck. Threats are solved by adding constraints to the plan such as ordering relations between steps. For instance, one might reorder the steps to make sure that the truck is used only once at any point of time.

UCP starts out with an initial plan, which consists of two steps t_0 (it has no conditions and has as effects each literal in the initial state) and t_∞ (it has as conditions the goals of the problem and no effects) and a single constraint ($t_0 \rightarrow t_\infty$) indicating that t_0 must occur before t_∞. Each partial plan π can be seen as representing a set of candidate solution plans, namely the ones that are consistent with the constraints in π. Thus, the initial plan represents all plans that are solutions to the planning problem. UCP has several ***refinement strategies*** that it can use to produce refinements of a partial plan. These refinement strategies can be seen as ways to constrain the conditions that any solution plan must meet; thus with each refinement that UCP makes to π, the set of solution plans is reduced. Each refinement strategy adds new steps or constraints to *P*. We do not have space to explain these strategies in detail. But basically UCP allows three kinds of refinements: forward state-space, which results in a totally ordered chain of actions, called ***the head fringe***, starting from the initial state (in Figure 1 this chain is formed by the steps t_0, t_2 and t_3). The second kind is backward state-space, which results in a totally ordered chain of actions, called ***the tail fringe***, starting from the goals (in Figure 1 t_∞ is the only step in this chain). The third kind is plan-space refinements, which allow a partial order between the steps (in Figure 1 steps t_1 and t_4-t_9 were obtained this way). Planning ends when a plan with no ***flaws*** (i.e., a partial plan with no open conditions and no threats) is obtained. In such case the plan is said to be ***complete***.

The TransUCP Adaptation Framework

The TransUCP algorithm defines Transformational Analogy plan adaptation over UCP. TransUCP receives as input the new problem (an initial state, a goal state) and a complete plan solving an old problem. It returns a complete plan solving the new problem or a failure

```
TransUCP (S, G, π_old)
//input: State S, Goals G, input plan π_old
//output: complete plan π_new or Failure

π_adj = AdjustPlan(π_old, S, G)
if π_adj is complete then return π_adj
PlanPool = {< π_adj, ↑>, < π_adj,↓> }
while (PlanPool ≠ Ø) do
   <π, D> ← select an element from PlanPool.
   Delete <π, D> from PlanPool
   if π is complete then return π
   if D = ↓ then
     Non deterministically, select any one of:
     1. RefinePlanForwardStateSpace (π)
     2. RefinePlanBackwardStateSpace (π)
     3. RefinePlanSpace (π)
     for each resulting plan π' do
        add < π', ↓> to PlanPool
   else // (D == ↑)
      select retracted plan π' of π with purpose tag p
      add < π', ↑> to PlanPool
      for each π'' ≠ π with π'' refining π' wrt p do
         add < π'', ↓> to PlanPool

return failure
```

Figure 2: the TransUCP algorithm

message if none can be generated.

Figure 2 shows the pseudo-code of TransUCP. It first calls AdjustPlan. AdjustPlan first maps common elements between the new problem and elements occurring in π_{old}. It adds to t_∞ any goal in G that is not mapped to a condition in t_∞. It also adds literals from the state S that do not map to effects in t_0. Afterwards, adjustPlan works by repeatedly (1) removing a step s that mentions objects in π_{old} that are not mapped into objects (S,G) and (2) removing any ordering or interval preservation constraint connecting to/from s. If the adjusted plan obtained, π_{adj}, is not complete then two copies are added to PlanPool; one labeled up (↑) and one labeled down (↓). The ↑ (↓) arrow indicates that a retraction (refinement) needs to be performed. PlanPool contains all candidate plans to be transformed in the next iteration. Hence, if PlanPool is empty, no candidate plan exists and the algorithm returns *Failure*. At each iteration of the while loop, a pair <π, D> is non deterministically chosen. If π is complete, it is returned and the algorithm terminates. Otherwise, if D is down (↓) then it basically calls one iteration of UCP, where one of three possible kinds of refinements is non deterministically chosen and all possible refinements of the chosen kind are performed. The resulting plans are added to PlanPool with down mark.

A ***retraction*** (↑) removes steps, constraints or bindings. To do so we needed to identify a step and/or a set of constraints that were added to a partial plan. We associate each set of modifications that are done to a plan with *purpose tags*, which indicate the flaw that was solved and the modifications made to solve it. So when D is ↑, we select the plan π' obtained by removing all modifications as indicated by the purpose tag. This plan is added with an up mark to the pool. We also add to the pool all possible alternative refinements π'' of π' solving this flaw, excluding π itself. The different types of ***purpose tags*** are:

i. Purpose (Step Added, t_j, forward state): This tag is for steps which are added to the plan during forward state-space refinement.
ii. Purpose (Step Added, t_j, backward state): This tag is for steps which are added to the plan during backward state-space refinement.
iii. Purpose (protect ((t_k, $t_i \rightarrow^Q t_j$))): This tag is for ordering/binding constraint which has been added to the plan to resolve the threat ((t_k, $t_i \rightarrow^Q t_j$)).
iv. Purpose (establish link, $t_i \rightarrow^Q t_j$): This tag is for ordering constraints which has been added to the plan to satisfy the open condition ($\rightarrow^Q t_j$).

If the tag selected was a forward state space refinement, then the step s added is removed; any step occurring in the head fringe after s is also removed. If the tag selected was a backward state space refinement, then the step s added is removed; any step occurring in the tail fringe before s is also removed. Similar processing is done for plan space refinement tags. Before adding the plan to PlanPool, TransUCP needs to be made sure that it is not a previously retracted/added plan (not shown in the pseudo-code).

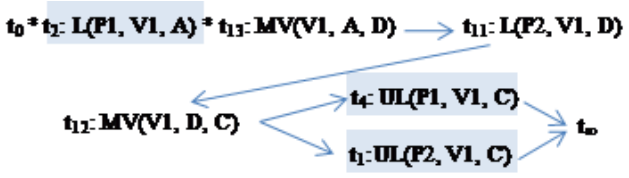

Figure 3: resulting complete plan

As an example, we use the plan shown in Figure 1. In the new problem to be solved, we have the same two goals as before, to relocate package P1 and P2 into location C. The difference is that this time there is no truck (i.e., V2) in location D. The AdjustPlan function takes this plan and modifies it so as to match the initial and goal states of the new problem and of the plan. Since the truck V2 is not available in the new problem, we remove V2 from the initial state and all those steps and constraints that involve V2. In doing so, we remove steps t_3, t_8, and t_9. Since the resulting plan is not a complete plan (e.g., step t_6 requires P2 to be in location B but step t_9 achieving this has been removed), we add the direction pointer pairs <π_{adj}, ↑> and <π_{adj}, ↓> to PlanPool. The final resulting plan is shown in Figure 3. Highlighted steps indicate steps retained from the input plan. Basically it uses V1 to first pick P1, move to D, load P2, move to B, and then unload both packages.

Minimal Plan Modification

In this section, we show that TransUCP does not use a conservative plan modification approach. Definitions 1–4 are the same ones used in (Nebel & Koehler, 1995).

Definition 1. An instance of propositional STRIPS planning is denoted by a 4-tuple *(Pr, O, I, G)*, where *Pr* is a finite set of ground atoms. *O* is a finite set of operators of the form $Pre \rightarrow Post$, where $Pre \subseteq L$ *are* the preconditions and $Post \subseteq L$ is the postconditions or effects. $I \subseteq Pr$ is the initial state. $G \subseteq L$ is the goal.

Definition 2. PLANSAT is the following decision problem: given an instance of the planning problem $\prod = (Pr, O, I, G)$, does there exist a plan Δ that solves $\prod$?

Definition 3. A **conservative strategy** to plan modification is one that solves the following **plan modification problem**: given a planning-problem instance $\prod_1 = (Pr, O, I_1, G_1)$ and a plan Δ that solves another instance $\prod = (Pr, O, I, G)$, produce a plan Δ_1 that solves $\prod_1$ by minimally modifying Δ.

Definition 4. MODSAT is the following decision problem: Given a planning-problem instance $\prod_1 = (Pr, O, I_1, G_1)$, a plan Δ that solves another instance $\prod = (Pr, O, I, G)$, and an integer k, does there exist a plan Δ_1 that solves $\prod_1$ and contains a subplan of Δ of at least length k?.

As pointed out by Nebel and Koehler, MODSAT is the definition produced by turning the plan modification problem into a search problem. Given a plan-modification strategy *M*, we can write a decision-theoretic version of *M* that uses *M* to produce a plan, measures the plan's length *l*,

and returns "yes" iff $l \leq k$. If M is a conservative strategy then the decision-theoretic version of M solves MODSAT. The main result of Nebel and Koehler is the following.

Proposition 1. If PLANSAT is PSPACE-complete or NP-complete, then MODSAT is a PSPACE-complete or NP-complete problem, respectively.

Furthermore, they show that the converse is not true; in some situations plan generation is a polynomial time problem while plan modification is NP-complete. This result provides strong evidence that plan adaptation can be computationally worse than plan generation and, therefore, calls into question problem-solving by plan adaptation.

We now turn our attention to the way TransUCP generates complete plans. The three possible ways in which TransUCP traverses the search space and finds the solution plan node are:

i. It finds the complete plan without having to retract beyond the adjusted plan.
ii. It retracts beyond the adjusted plan all the way back to the initial plan and starts planning from first principles thereon.
iii. The planner retracts beyond the adjusted plan, but not all the way until the initial plan.

Theorem: In each of the three cases mentioned above, TransUCP does not necessarily produce minimal modifications of the given case plan Δ.

Proof sketch: The proof is by contradiction. By taking the example in the transportation domain that we have presented in this paper and showing that in each case a plan is generated that is not necessarily minimal. The plan shown in Figure 3 retains 3 steps. A minimal plan modification can retain up to 6 steps. □

Therefore, TransUCP does not fall under the category of MODSAT. The definition of a conservative plan adaptation strategy used in the theoretical studies (Nebel & Koehler, 1995) does not hold for Transformational Analogy, so their complexity results do not apply to this form of adaptation. The reason is that Transformational Analogy, rather than looking to reuse the largest possible number of steps, just uses the first complete plan it happens to find—and if this turns out be minimal, this is purely coincidental. Thus, for each case in which Transformational Analogy produces a minimal plan modification, there are many others where it does not. To illustrate this point we did a small experiment. We implemented a program simulating the search space that the TransUCP algorithm would traverse when generating solutions in the logistics transportation domain. Refinements (↓) and retractions (↑) were selected randomly. In the experiments performed, the problems were also randomly generated. Since the search space can be very large we added pruning techniques reducing the chances that plans that clearly contain redundant steps (e.g., move from A to B, followed by move from B to A) are further refined. The case plan to be reused is the one from Figure 1. After running TransUCP giving as input 10 randomly generated problems and the case plan, non-minimal solution plans were generated in every single run.

Final Remarks

We presented TransUCP a general framework for Transformational Analogy covering classical planning paradigms. Like the UCP framework on which it is based, TransUCP is not intended to be an efficient problem solving algorithm in itself. Rather, it has a twofold purpose. First, it is intended as a tool for defining unambiguously the various transformational modifications that can be made to a plan. Second, it is intended to serve as an analysis tool to understand possibilities and limitations of transformational adaptation. Although we have only partly advanced on this second purpose in this paper, by clarifying the implications for Transformational Analogy of the complexity results of Nebel & Koelher (1995), we believe that our work points to a direction whereupon other CBR methods can be carefully defined and its properties analyzed. In future work we will analyze trade-offs in the space traversed by TransUCP versus UCP.

Acknowledgements

This research was in part supported by the National Science Foundation (NSF 0642882).

References

Au, T.C., Muñoz-Avila, H., and Nau, D.S. (2002) On the Complexity of Plan Adaptation by Derivational Analogy in a Universal Classical Planning Framework, *In Proceedings of ECCBR-02*, Springer

Carbonell, J.G. (1983) Learning by Analogy: formulating and generalizing plans from past experience. *Machine Learning: An Artificial Intelligence Approach*.

Cunningham, P., Finn, D., & Slattery, S. (1996) Knowledge Engineering Requirements in Derivational Analogy. In *Proceedings of ECCBR-96*. Springer.

Kambhampati, S. and Srivastava B. (1995) Universal Classical Planner: An algorithm for unifying state-space and plan-space planning. In: *Proceedings of EWSP-95*.

Hammond, K. (1990). Explaining and repairing plans that fail. Artificial Intelligence, 45: 173-228.

Hanks, S. and Weld, D. (1995). A domain-independent algorithm for plan adaptation. *Journal of Artificial Intelligence Research*, 2.

Kuchibatla, V., and Muñoz-Avila, H. (2007) An Analysis on Transformational Analogy: General Framework and Complexity, *In Proceedings of ECCBR-07*, Springer

Nebel, B. and Koehler, J (1995). Plan reuse versus plan generation: a theoretical and empirical analysis, *Artificial Intelligence*.

Tonidandel, F. and Rillo, M. (2005) Case Adaptation by Segment Replanning for Case-Based Planning Systems. *Proc. ICCBR-05*.

van der Krogt, R.P.J. and de Weerdt, M.M.. (2005) Plan Repair as an Extension of Planning. In *Proceedings of ICAPS-05*.