

# An Analysis on Transformational Analogy: General Framework and Complexity

Vithal Kuchibatla & Héctor Muñoz-Avila

Department of Computer Science & Engineering;  
Lehigh University; Bethlehem, PA 18015

**Abstract.** In this paper we present TransUCP, a general framework for transformational analogy. Using our framework we demonstrate that transformational analogy does not meet a crucial condition for a well-known worst-case complexity scenario, and therefore the results about plan adaptation being computationally harder than planning from the scratch does not apply to transformational analogy. We prove this by constructing a counter-example that does not meet this condition. Furthermore, we perform experiments that demonstrate that this counter-example is not an exception. Rather, our experiments show that it is unlikely that this condition will be met when performing plan adaptation with transformational analogy.

## 1 Introduction

Transformational analogy is a problem-solving 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 from actions. Interest on transformational analogy started from early case-based reasoning systems (Cox *et al.*, 2006). In particular, the CHEF system constructs cooking recipes, which are plans because recipes are sequences of cooking steps such as boiling a certain amount of water (Hammond, 1990). These recipes are modified depending on factors such as the ingredients currently available.

Over the years, derivational analogy, an alternative problem-solving technique that advocates reusing the sequence of derivations that led to a solution plan rather than the plan itself, gained prominence among the case-based planning community. Part of the reason for this prominence is the interest in problem solving by combining first-principles planners and case-based reasoning. If the first-principles planner is used to generate plans, then it is straightforward to annotate the derivations that these planners followed to obtain the plans (Veloso, 1994). Thus, derivational analogy is a good fit for this line of research. There has been recent work on developing DerUCP, a framework using derivational analogy (Au *et al.*, 2002). It enhances the universal classical planning (UCP) framework to build a generic, domain-independent plan adaptation algorithm. An analysis of DerUCP demonstrates that it does not fall under the worst-case complexity scenario by Nebel and Koehler (1995), and therefore, their results about plan adaptation being computationally harder than plan adaptation does not apply to it.

Despite some well-documented applications of derivational analogy, a major difficulty of using this technique is the requirement about the availability of the derivational trace that led to a solution. Even when a domain theory is available, we might not know how a particular plan was created. For example, the rules for playing chess are known but we might not know the reasoning behind a player making a sequence of moves. This knowledge engineering requirement of derivational analogy is well known (Cunningham *et al.*, 1996). Perhaps for this reason, application-oriented papers in case-based reasoning conferences that use some form of adaptation frequently use transformational analogy. Yet, despite this interest no general framework for analyzing transformational analogy exists to date.

In this paper, we present TransUCP, a general framework for transformational analogy built on top of UCP. Using our framework we demonstrate that transformational analogy does not meet the worst-case complexity results of Nebel and Koehler (1995), and therefore, their results about plan adaptation being computationally harder than plan adaptation does not apply to it. We prove this by constructing a counter-example in which a crucial condition is not met. Furthermore, we perform experiments that demonstrate that this counter-example is not an exception. Rather, our experiments show that it is very unlikely that transformational analogy falls under the scenario described in Nebel and Koehler (1995).

The paper continues as follows. The next section describes the Universal Classical Framework, on which TransUCP is based. Section 3 presents TransUCP. The next two sections describe an example of problem-solving with TransUCP and analyze the search space followed by TransUCP. Section 6 proves that TransUCP does not fall under the scenario of Nebel and Koehler (1995). The next section describes the experimental results. We conclude this paper with some final remarks.

## 2 Background

The SPA system (Hanks and Weld, 1995) is a general purpose algorithm for transformational analogy. SPA takes advantage of the partial-order plan representation of partial-order planners to modify an existing plan. Our general framework enhances SPA to other forms of planning by taking advantage of the UCP framework, which we describe below.

### 2.1 Partial Plan

The algorithm proposed in this paper uses to a large extent similar representation format and data structures as of that used in the UCP algorithm as proposed by Kambhampati and Srivastava, (1995). A partial plan is represented by the 4-tuple  $\langle T, O, B, L \rangle$  where:

- $T$  is the set of all the steps in the partial plan,
- $O$  is the set of ordering constraints between the steps of  $T$ ,
- $B$  is the set of binding (co-designation constraints, which require variables to take the same value) and prohibitive bindings (non co-designation constraints, which requires variables not to take the same value) in the preconditions and post-conditions of the operators, and,

- L is the set of auxiliary constraints, which are of 3 types:
  - Ordering constraints are of the form  $(t_i \rightarrow t_j)$  indicating that step  $t_i$  precedes step  $t_j$ , though not necessarily immediately.
  - Interval Preservation Constraints which are the form  $(t_i \rightarrow^Q t_j)$  which means that the condition Q has to be true between the steps  $t_i$  and  $t_j$  of T. This is a “causal link” used in partial-order planners such as SNLP. If  $(t_i \rightarrow^Q t_j)$  holds, it implies that  $(t_i \rightarrow t_j)$  also holds.
  - Contiguity constraints, which are the form  $(t_i * t_j)$  which means that the step  $t_i$  has to be immediately followed by step  $t_j$ .

We illustrate these concepts with an example in the logistics transportation domain. In this domain, there are different packages located at various locations and some or all of these packages have to be re-located to specific locations. There are also means of transportation such as trucks located at various cities and these are used to move the packages. The instance of the problem used here is the same as that used by Au *et al.* (2002). The problem shown in Figure 1 requires package P1 in location A, and package P2 in location D, to be relocated in location C. The figure also shows two trucks V1 and V2 at locations A and D respectively.

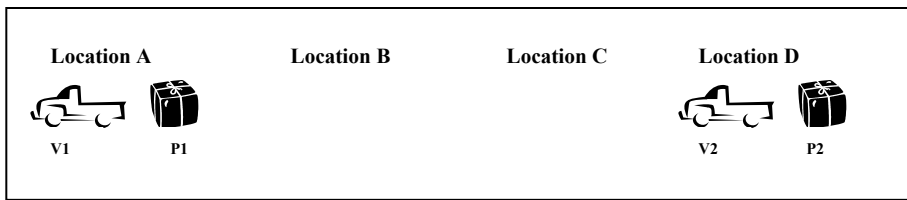


Figure 1: Planning problem in the logistics transportation domain

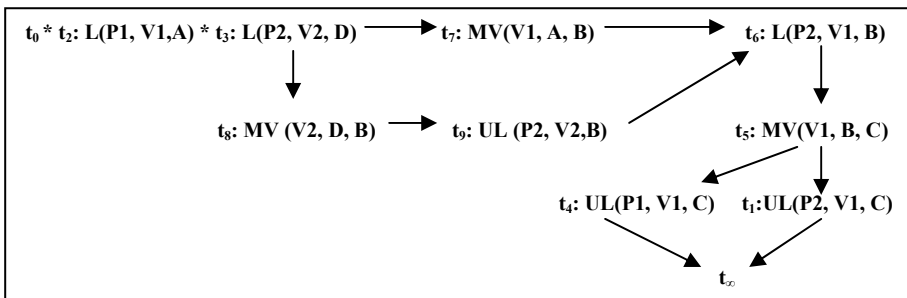


Figure 2: Partial plan solving the problem in Figure 1

Figure 2 shows a partial plan solving the problem depicted in Figure 1. Arrows denote either ordering or interval preservation constraints. The plan consists of 11 steps, denoted by  $t_k$ . Steps  $t_0$  and  $t_{10}$  are special steps that we will explain momentarily. The other steps are labeled according to the following conventions:

- The action 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 onto location Lc.
- MV (V, L1, L2) indicates that V is *moved* from L1 to L2.

Under these conventions P1 is loaded in V1 and P2 into V2. P2 is relocated in B using V2, where it is picked by V1, which has moved from A to B. V1 continues to C, where both packages are unloaded. Besides these steps, other primary attributes of a partial plan are its open conditions and threats. A *condition* has the form  $(\rightarrow^Q t_k)$

indicating that the condition  $Q$  has to be satisfied for step  $t_j$ . Each step  $t_j$  in the plan can produce effects ( $t_j \rightarrow^Q$ ) which can be used to satisfy conditions. A condition  $\rightarrow^Q t_k$  is satisfied by adding an interval preservation constraints  $t_m \rightarrow^Q t_k$ , 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  $(t_k, t_i \rightarrow^Q t_j)$  where  $t_k$  can be inserted between  $t_i$  and  $t_j$  and the post condition of  $t_k$  can negate or add  $Q$ . 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  $t_j$ , but another step  $t_k$  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.

*Operators* have a set of preconditions which must be satisfied before the step can be applied and a set of post-conditions which are true after the step has been applied. When an operator is applied it is added to a plan as a step. This is how steps  $t_1$  to  $t_9$  where added to the plan in Figure 2. When planning starts a so-called initial *null plan* is created. This plan consists of two steps:  $t_0$  and  $t_\infty$ . The step  $t_0$ , called the initial state, has no preconditions and has as effects the conditions that are true in the opening state. The step  $t_\infty$ , called the *final state*, has no effects and has as the conditions the goals to be achieved. The null plan has also an ordering constraint,  $t_0 \rightarrow t_\infty$ , and no bindings. The objective of the planning problem is to refine the initial plan into a *solution plan*, a partial plan with no open conditions and no threats. Open conditions and threats in a partial plan are referred to as *flaws* of the plan.

## 2.2 Universal Classical Planning

The Universal Classical Planner (UCP) takes a partial plan and performs refinements to it in an iterative manner until a solution plan is generated. During each pass of the iteration, the refinement done to the plan can be addition of steps or constraints to the existing partial plan. The possible types of refinements that a UCP planner can choose to perform on the partial plan on each iterative pass are:

- i. **Forward state space plan refinement:** A head step of a partial plan is defined as a step  $t_j$  of the plan where  $t_0 * t_1 * \dots * t_j$  and there is no step  $t'$  such that  $t_j * t'$ . The sequence of steps  $t_0 * t_1 * \dots * t_j$  is called the header of the plan. The set of all states  $t_i$  that can immediately follow the head step  $t_j$  is called the head fringe. Forward state space plan refinement involves selecting a new step or a step from the head fringe of a plan and appending it to its header.
- ii. **Backward state space plan refinement:** A tail step of a partial plan is defined as the step of  $t_j$  of the plan where  $t_j * t_{j-1} * \dots * t_\infty$  and there is no step  $t'$  such that  $t' * t_j$ . The sequence of steps  $t_j * t_{j-1} * \dots * t_\infty$  is called the trailer of the plan. The set of all states  $t_i$  that can immediately precede the tail step  $t_j$  is called the tail fringe. Backward state space plan refinement involves selecting a new step or a step from the tail fringe of a plan and putting it immediately before its trailer.
- iii. **Partial plan space refinement:** During plan space refinement, a flaw is selected at random from the current plan. This flaw could be either an open condition or a threat. If it is an open condition, it is resolved by adding or changing ordering constraints to the plan or by adding a new step such that it satisfies the required open condition. There can be more than one way to reorder the steps and, similarly, there can be more than one step that can be added to satisfy the open

condition. Therefore, resolving the open condition can result in multiple partial plans. If the selected flaw is a threat, it is handled by a “Resolve Threat” function. Given a threat of the form  $(t_k, t_i \rightarrow^Q t_j)$ , this function resolves it by either

- Ordering  $t_k$  before  $t_i$  consistently, or,
- Ordering  $t_k$  after  $t_j$  consistently, or,
- Adding the appropriate binding constraints to the plan so as to negate the threat

Figure 3 illustrates the working of a universal classical planner. During each iteration, it can choose one of the above three refinements and modify the partial plan according to the selected refinement strategy.

### 2.3 Definitions

In this paper, we introduce the TransUCP framework and prove that it does not fall under the category of a conservative planner, as per definitions below, taken directly from (Nebel & Koehler, 1995).

1. **PLANSAT** is the following decision problem: given an instance of the planning problem  $\Pi$ , does there exist a plan  $\Delta$  that solves  $\Pi$ ?
2. A **conservative approach** to plan modification is one that solves the following *plan modification problem*: given a planning-problem instance  $\Pi_i$  and a plan  $\Delta$  that solves another instance  $\Pi$ , produce a plan  $\Delta_i$  that solves  $\Pi_i$  by minimally modifying  $\Delta$
3. **MODSAT** is the following decision problem: Given a planning-problem instance  $\Pi_i$ , a plan  $\Delta$  that solves another instance  $\Pi$ , and an integer  $k$ , does there exist a plan  $\Delta_i$  that solves  $\Pi_i$  and contains a sub plan of  $\Delta$  of at least length  $k$ ?

## 3 The TransUCP Planning Algorithm

The main idea behind the TransUCP algorithm is to solve the planning problem by using transformational analogy over UCP. The inputs to the algorithm are: the initial state, the goal state and the case library. TransUCP returns the solution plan or a failure message if it could not generate one.

*Progressive refinements* are defined as those modifications made to the plan which increase its possible number of ground linearisations or increase the total number of steps in the plan. In terms of searching in a plan space graph, all those refinements that take a node to its parent nodes can be looked upon as non-progressive refinements and the ones that take the current node to one of its children would be progressive refinements. The three kinds of refinements used in the universal classical planning algorithm (Section 2.2) constitute the progressive refinements. All refinements made to a plan that are not progressive refinements can be termed as *non-progressive*.

**Purpose Tags.** The TransUCP algorithm generates and modifies a partial plan in an iterative manner doing one refinement (progressive or non-progressive) in each pass. During each pass, a step and/or a set of constraints are added/deleted to/from the plan. We associate each set modifications done to the plan in each pass is with a data

structure called the *purpose tag* which indicates the purpose of these modifications. These tags are primarily used when we retract the plan backwards i.e. when we delete steps or constraints from a plan. The different types of tags used in TransUCP are:

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

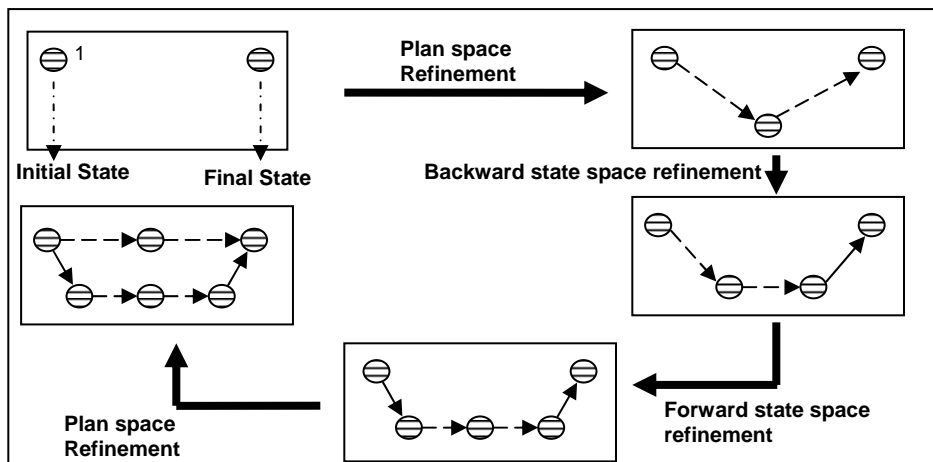


Figure 3: Universal Classical Planning

### 3.1 The Algorithm

TransUCP first retrieves a case from the case library that is the best match for the current problem in terms of most similar initial and goal states. Though this suggests a probable heuristic for case retrieval from the case library, the actual logic to be used for this is not discussed in this paper as it is not the primary focus.

The plan returned from the case library, *LibraryPlan*, is adjusted so as to make its initial and goal states the same as Initial and Goal. This process includes adding sub-goals of Initial and Goal that are not present in the initial and goal states of the *LibraryPlan* and deleting those sub-goals that are not present in Initial and Goal and are present in their counterparts of *LibraryPlan*. During this process of addition and deletion, some steps of the partial plan along with their ordering, binding and auxiliary constraints might have to be deleted. As a result, there might be open conditions and threats in the adjusted plan. The TransUCP algorithm tries to remove these open conditions and threats to make the plan a solution plan.

The plan returned by the *AdjustExactly* function, *AdjustedPlan*, is then checked to see if it is a solution to the current problem. If it is not, we add the plans,  $\langle \text{AdjustedPlan}, \text{UP} \rangle$  and  $\langle \text{AdjustedPlan}, \text{DOWN} \rangle$  to the *PlanPool*. The purpose of the direction indicators, UP and DOWN is discussed in detail in later sections. This

collection of plans PlanPool is passed to the function TransformPlan, which returns the solution plan.

**TransUCP** (Initial, Goal, Library) Returns: Final Plan P or Failure

```
LibraryPlan = select the plan from the Library with the most similar initial and
                goal states
AdjustedPlan = AdjustExactly (LibraryPlan, Initial, Goal)
IF AdjustedPlan is a solution
    THEN return AdjustedPlan
PlanPool = {<AdjustedPlan, UP>, <AdjustedPlan, DOWN>}
FinalPlan = TransformPlan (Initial, Goal, PlanPool)
IF FinalPlan == failure
    THEN return failure
Return FinalPlan
```

The TransformPlan function is called recursively until a solution plan is arrived at or a failure is returned, which happens when the PlanPool is extinguished. Given the PlanPool, which is passed as an argument to the function, it chooses a plan from it non-deterministically and checks if it is a solution, and if so, returns it. If not, it checks for the direction pointer of the selected plan P.

If the direction pointer of the chosen plan is DOWN, it performs progressive refinements and if not, it performs non-progressive refinements. If the direction pointer is DOWN, the algorithm chooses non-deterministically which of the three possible refinements is to be applied. All the plans returned by the progressive refinement chosen are added to the PlanPool and the TransUCP is recursively called until a solution plan or a failure is encountered.

If the direction pointer is UP, one of the decisions made in the current plan is retracted back through non-progressive refinements, by the call to RetractRefinement, and the resulting plans are added to the PlanPool followed by the recursive call to TransUCP. Basically, the RetractRefinement function selects a step from the current partial plan and removes it from the plan. The RetractRefinement function takes as argument the plan P and selects a purpose tag from it. Having chosen the tag to retract, it undoes the progressive refinement associated with this tag. The exact criteria to be followed in choosing the purpose tag to be retracted are not dealt in this paper. A good heuristic to be followed for this selection process can be found in Hanks and Weld (1996). The progressive refinement to undo can be any of the three kinds of refinements used in the universal classical planning algorithm (Section 2.2). In all of the three cases, the function removes the steps, constraints or bindings which were associated with this tag and added to the plan. If the tag selected was associated with forward state space refinement, then the step added is removed, through a call to RemoveStep, and all other ways of performing forward state space refinement to the plan are returned to be added to the plan pool. Before adding the plan P<sub>1</sub> to PlanPool, it is made sure that it does not map onto the original retracted plan P. This is to ensure that we do not add the same plan back to the pool again. By saying that one plans *maps* onto another plan, we mean that there is a 1:1 mapping between their steps, links, binding constraints and purpose tags.

Similar processing is done for backward and partial plan space refinement tags. When the refinement to be retracted is a partial plan space refinement, the step

associated with the purpose tag selected is retracted and the flaw that originally caused this refinement to be made to the plan is also returned to RefinePlanSpace. This is done to ensure that only those partial plans formed by resolving this particular flaw through RefinePlanSpace are added to the plan pool. In essence, we are constricting RefinePlanSpace from selecting a flaw to be resolved at random. This completes the explanation of the TransUCP algorithm. In the subsequent sections, the working of the algorithm and its properties are elucidated with examples.

**TransformPlan** (Initial, Goal, PlanPool) Returns: Final Plan P or Failure

```

If PlanPool is empty THEN return failure.
<P, D> = select an element from PlanPool.
Delete <P, D> from PlanPool
If P is a solution THEN return P
If D == DOWN THEN //Progressive Refinements
    Non-deterministically, select any one of
    1. P' = RefinePlanForwardStateSpace (P)
        Add <P', DOWN> and <P', UP> to PlanPool
    2. P' = RefinePlanBackwardStateSpace (P)
        Add <P', DOWN> and <P', UP> to PlanPool
    3. RefinePlanSpace (P)
        For each plan Pi returned by RefinePlan (P, null) do
            Add <Pi, DOWN> to PlanPool
ELSE IF (D == UP) // Non-progressive Refinements
    Add all elements of RetractRefinement (P) to PlanPool
Recursive Invocation:
    TransUCP (Initial, Goal, PlanPool)

```

## 4 Example

In this section, we take a planning problem and the solution plan returned when the UCP algorithm (Kambhampati & Srivastava; 1996) solves it. For this example, we use the problem and plan described in Figures 1 and 2 respectively as the input case. In the new problem to be solved, we have the same goals, to relocate package p1 and p2 into location C. The difference is that this time there is no truck in location D as illustrated in Figure 4. The AdjustExactly function takes this plan and modifies it so as to match the initial and goal states of the new problem and of the case. In our example, the goals happen to be the same but the initial states are different. 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 get the partial incomplete plan as shown in Figure 5. It can be seen that steps  $t_3$ ,  $t_7$ ,  $t_8$  and  $t_9$  have been deleted. The open threats and conditions that result from this modification and that need to be resolved are also shown in the figure.

Let us denote this plan by  $P_1$ . Since this is not a solution plan, we add the direction pointer pairs  $\langle P_1, UP \rangle$  and  $\langle P_1, DOWN \rangle$  to the PlanPool and this pool is passed to TransformPlan. Let us assume that TransformPlan chooses the pair  $\langle P_1, DOWN \rangle$  from the plan pool to refine. Since the direction pointer is DOWN, it performs

progressive refinement on the plan  $P_1$ . Assuming without the loss of generality that the refinement strategy chosen is forward state space refinement, the current head step is  $t_2$  and we can append a new step  $t_8$   $MV(V1, A, D)$  to the head step with the constraint  $t_2 * t_8$  as the preconditions of  $t_8$  are satisfied at  $t_2$ . The resulting plan, labeled as  $P_2$  is shown in Figure 6 and the 2-tuple  $\langle P_2, DOWN \rangle$  is added to the PlanPool.

**RetractRefinement** (Plan P): Returns: List of (Plan, Direction) pairs

Define L: a local list of plans  
R = select a purpose  
(F, P<sub>0</sub>) = RemoveStep(R, P)  
Add the tuple (P<sub>0</sub>, UP) to local list L  
IF purpose in R was forward state space refinement THEN  
    For each plan P<sub>1</sub> returned by RefinePlanForwardStateSpace (P<sub>0</sub>)  
    Do If P<sub>1</sub> does not map onto P, add  $\langle P_1, DOWN \rangle$  to list L.  
Else IF purpose in R was backward state space refinement THEN  
    For each plan P<sub>1</sub> returned by RefinePlanBackwardStateSpace (P<sub>0</sub>)  
    Do If P<sub>1</sub> does not map onto P, add  $\langle P_1, DOWN \rangle$  to list L.  
Else IF purpose in R was partial plan space refinement THEN  
    For each plan P<sub>1</sub> returned by RefinePlanSpace (P<sub>0</sub>, F)  
    Do If P<sub>1</sub> does not map onto P, add  $\langle P_1, DOWN \rangle$  to list L.  
Return all plans, direction pairs collected in list L.

During the second pass of TransUCP, the PlanPool contains the pairs  $\langle P_1, UP \rangle$ ,  $\langle P_2, DOWN \rangle$ ,  $\langle P_2, UP \rangle$ . If the pair chosen by TransUCP was  $\langle P_2, DOWN \rangle$  and the progressive refinement chosen was partial plan space refinement, let us assume that the open condition selected to resolve is  $(\rightarrow^Q t_6)$ , where  $Q = at(B, truck)$ . This flaw is resolved by adding a new step  $t_9$ :  $MV(V1, A, B)$  and ordering it before  $t_6$  with the ordering constraint  $t_8 < t_9 < t_6$ . This resolves the flaw and results in the plan  $P_3$  shown in Figure 7. By adding the step  $t_9$ , the open condition  $(\rightarrow^Q t_9)$ , where  $Q = at(A, truck)$ , is introduced and is added to the set of flaws of the plan. Continuing in this manner TransUCP continuously keeps refining the partial plan and searches for the first solution node. Figure 8 shows one of the possible solution plans returned by TransUCP.

## 5 How TransUCP Traverses the Search Space

Plan adaptation as done by TransUCP to find a solution plan is carried out in a similar fashion as searching through a partial plan space. The entire process is comparable to searching for a solution plan node in a graph, in which, each node represents a partial plan. Edges between the nodes represent refinements between the plans represented by the nodes – progressive or non-progressive.

The nodes resulting from performing non-progressive refinements on a node are referred to, in this paper, as the parents of the node and similarly, the nodes resulting from performing progressive refinements are referred as the children of the node. The graph being searched is not necessarily a tree because, given a node, non-progressive refinements on it can be performed in more than one way, thus producing multiple

parents for a given node. Given a plan from the case library and a planning problem (initial and goal states), TransUCP first modifies the case plan so as to match its initial and goal states to those of the given problem. Once this has been done, it starts the process of searching for the solution plan in the plan space. The modified input plan would be the starting point of the search (see Figure 9). This node would be an inner node in the graph. In the TransUCP function, when the direction pointer chosen is UP, the planner browses upwards into the parents of the current node by performing non-progressive refinements, i.e. by deleting some steps or constraints from the current plan.

**RemoveStep** (Purpose R, Plan P): Returns: List of (Flaw, Plan) pair

IF purpose in R was forward or backward state space refinement THEN  
 Remove the step tagged with R and all links and constraints associated with it to get the resulting plan  $P_0$ .  
 Return (*null*,  $P_0$ ).

ELSE //partial plan space refinement  
 IF R is of the form (protect  $((t_k, t_i \rightarrow t_j))$ ) THEN  
 $F = (t_i \rightarrow^Q t_j, t_k)$   
 $P_0 =$  a copy of P  
 Delete from  $P_0$  all constraints tagged with R  
 Return (F,  $P_0$ )

ELSE IF R is of the form (establish link,  $t_i \rightarrow t_j$ ) THEN  
 $F = (\rightarrow^Q t_j)$   
 $P_0 =$  a copy of P  
 Delete link  $(t_i \rightarrow t_j)$  from  $P_0$   
 Delete from  $P_0$  all constraints tagged with R  
 IF  $P_0$  contains no link of the form  $t_i \rightarrow t_j$  for any step  $t_k$  and condition Q THEN  
 Delete  $t_i$  from  $P_0$  along with all constraints tagged with (Step Added,  $t_i$ , plan space)  
 Return (F,  $P_0$ )

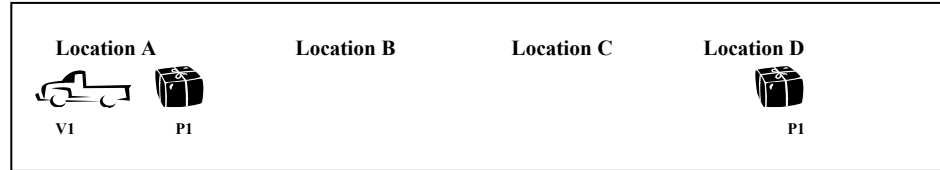


Figure 4: Planning problem in the transportation domain to be solved

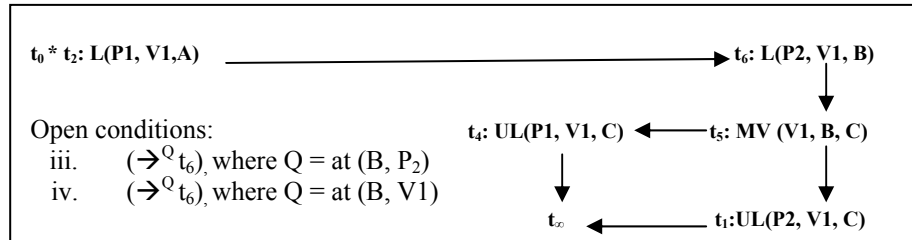


Figure 5: Partial plan  $P_1$  after initial adjusting

When the direction pointer chosen is DOWN, it scales the graph “downward”, into the children nodes of the node by performing progressive refinements, i.e. by adding steps or constraints. It performs this process of traversing the graph until it hits the first node that satisfies the conditions of being a solution plan for the given problem. It is to be noted that the planner takes care never to visit a node more than once during its execution.

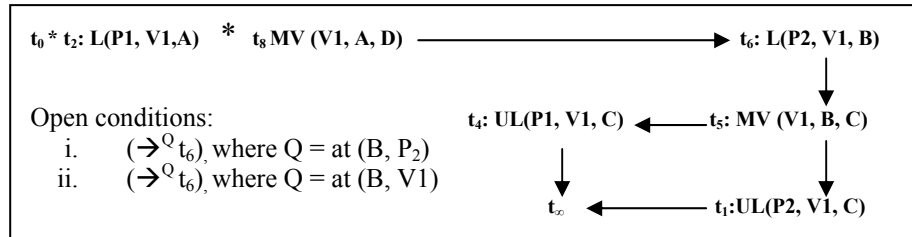


Figure 6: Partial plan  $P_2$

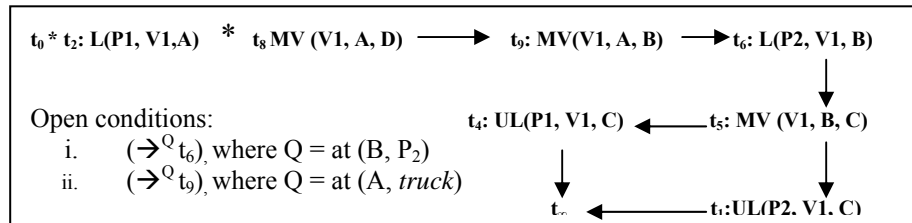


Figure 7: Partial plan  $P_3$

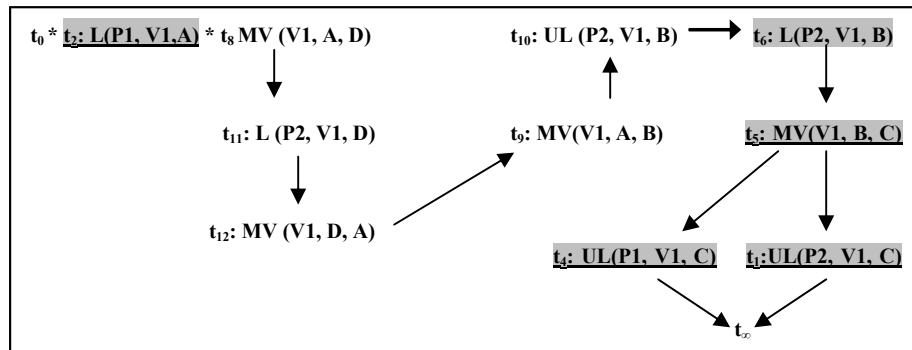


Figure 8: Solution plan generated by TransUCP

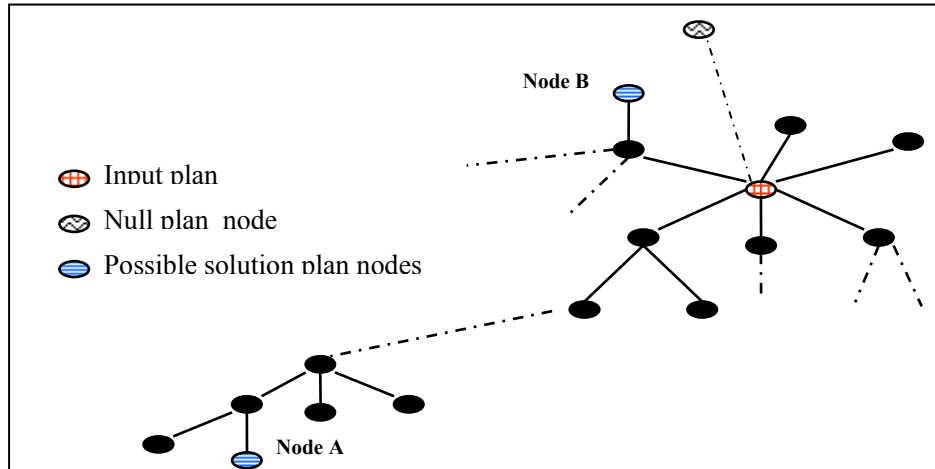
## 6 Properties

In this section, we show that TransUCP does not use a conservative plan modification approach, as per the definitions in Section 2.3, to find the solution plan for a given planning problem. We also discuss completeness and the issue of non-determinism in TransUCP. The three possible ways in which TransUCP traverses the search space and finds the solution plan node are:

- i. The planner finds the solution plan node without having to retract beyond the starting node (input plan node in Figure 9). That is, it never visits the parents of

the input plan node. This is the case when Node A in Figure 9 is returned as the solution node by the planner.

- ii. The planner, in search of the solution plan node, retracts all the way back to the null plan node and starts planning from first principles thereon.
- iii. The planner retracts, but not all the way until the null plan node. This is the case when Node B in Figure 9 is returned as the solution node by the planner.



**Figure 9: Graph Traversal by TransUCP**

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

**Proof:** The proof is by contradiction. Let us consider the first case above where the planner does not retract beyond the starting node (input plan node in Figure 9). Let us assume that TransUCP always produces solution plans that are minimal modifications of the given case plans. We shall provide a counter example to show that this is not true.

Consider the planning problem instance (Figure 4) and its solution generated by TransUCP (Figure 8). The solution plan generated by TransUCP certainly comes under the first case because the planner does not retract beyond the input plan node at any point during the execution of the algorithm. If TransUCP were to always produce solution plans that are minimally modified, then no other plan which solves the same instance of the planning problem should contain a sub plan of the original case plan which is greater in size (number of steps) than the sub plan of solution produced by TransUCP.

But the plan shown in Figure 10 solves the planning problem in Figure 4 and is minimally modified from the case plan shown in Figure 2. The highlighted steps show those that have been reused from the original plan. It is to be noted that only those steps that have been directly taken from the original plan are taken into account as reused steps and those that have been derived from first principles are not. If this plan is compared to the solution plan generated by TransUCP (Figure 8), also in which the reused steps have been highlighted, it can be seen that it is not minimally modified from the original case plan. This is a contradiction to our initial assumption. We can similarly produce counter examples for the remaining two cases and show that TransUCP does not necessarily always generate solution plans that are minimally modified from the original case plans. Hence we can conclude and prove that

TransUCP is not a conservative planner in the sense of as per the definitions of Nebel and Koehler (1995).□

Therefore, TransUCP does not fall under the category of MODSAT as defined in Section 2.3. It has been proved by Nebel and Koehler that answering the MODSAT decision problem can be computationally harder than PLANSAT. Since TransUCP does not satisfy the requirement for being a MODSAT problem, as it does not guarantee to generate a minimally modified plan, its complexity would not fall in this worst case scenario, i.e. problem solving with TransUCP will be computationally harder than problem solving from scratch.

**Non-determinism of TransUCP.** There are “decision points” at various stages of the implementation where choices are made non-deterministically, without any heuristic being used. The selection of a plan from the plan pool and the choice of progressive or non-progressive refinements to be made to the plan constitute some of these decision points. Forward state space and backward state space planning further contains points where random choices are made. The TransUCP framework, as proposed in this paper, is meant as a generic domain-independent framework for a planner. Hence there is non-determinism at various decision points. It is expected that, when the planner is used in a particular domain, appropriate domain dependent heuristics would be added and used at these decision points to improve the performance and efficiency of the planner. One of the most likely places where heuristics could help the most are when selecting a plan from the PlanPool, where by assigning weights based on heuristics to each plan, the choice of the next plan to be chosen for refined can be altered. This would ensure that the search of the plan space is carried out in a more guided and efficient fashion. Therefore, it is quite logical that non-determinism is replaced by heuristics in the actual implementation of this planner.

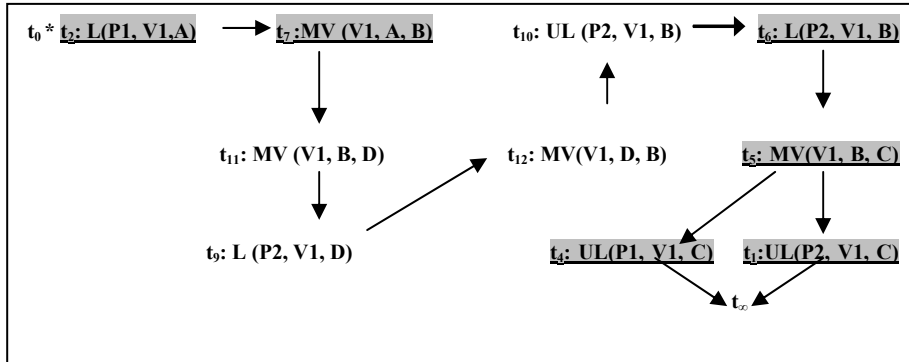
**Completeness of TransUCP.** Completeness of a planner, as defined by Hanks and Weld (1995), says that a planner will eventually find the solution plan for a particular planning problem, if there exists one. It has been proven (Kuchibatla, 2006) that TransUCP is complete, under the assumption that the given planning problem has a solution and that the partial plan search space has finite boundaries.

## 7 Empirical Results

In this section, we describe the details of the implementation of TransUCP and elucidate its results. The purpose of these experiments was to show that the counter-example shown in the previous section was not an exception and that that TransUCP very rarely behaves like a conservative planner.

The TransUCP algorithm was implemented to generate solutions in the logistics transportation domain. In the experiments performed, the problems were randomly generated, meaning each of them contains an arbitrary number of the elements in the domain such as trucks and locations. 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 2. After running TransUCP giving as input 10 randomly generated problems and the case plan, non-minimal solution plans were generated in every run. Figure 11 shows the average number of

nodes in the plan space graph that were traversed before the solution node was found versus the number of elements (trucks, cities and packages) in the problem.

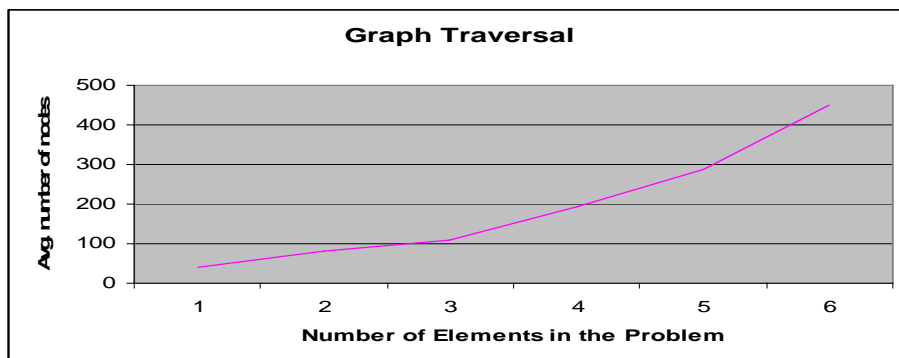


**Figure 10: Minimally modified solution plan**

For example, 6 elements mean that there were 4 locations and 2 trucks in the problem. From the figure we can see that even for a small problem with just 6 elements, the size of the search space is very large, which is why it is very unlikely that a minimal plan modification is generated.

## 8 Final Remarks

In this paper we introduced TransUCP, a general framework for transformational analogy. Using our framework we demonstrate that transformational analogy does not always perform a conservative plan adaptation by carefully constructing an example where conservative plan adaptation does not occur. Therefore, transformational analogy does not fall under the worse case scenario of Nebel & Koehler (1995). Furthermore, we perform experiments that demonstrate that it is unlikely that any plan adaptation with transformational analogy will be conservative.



**Figure 11: Number of nodes traversed in the graph**

## Acknowledgements

This research is sponsored by DARPA and managed by NRL under grant N00173-05-1-G034. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, NRL, or the US Government.

## References

- Au, T.C., Muñoz-Avila, H., & Nau, D.S. (2002) On the Complexity of Plan Adaptation by Derivational Analogy in a Universal Classical Planning Framework. In proceedings of the Sixth European Conference on Case-Based Reasoning (ECCBR-02). Springer.
- Carbonell, J.G. (1983) Learning by analogy: formulating and generalizing plans from past experience. *Machine Learning: An Artificial Intelligence Approach*. R.S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.). Tioga, Palo Alto, California.
- Carbonell, J.G. (1986) Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. *Machine Learning*.
- Cunningham, P., Finn, D., & Slattery, S. (1996) Knowledge Engineering Requirements in Derivational Analogy. In *Proceedings of the European Workshop on Case-Based reasoning (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 the Third European Workshop on Planning (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.
- Ihrig, L. & Kambhampati, S. Design and implementation of a replay framework based on a partial order planner. In Weld, D., editor, In: *Proceedings of AAAI-96*. IOS Press, 1996.
- Nebel, N. and Koehler, J (1995). Plan reuse versus plan generation: a theoretical and empirical analysis, *Artificial Intelligence*, 76, p427–454.
- Veloso, M. *Planning and learning by analogical reasoning*. Berlin: Springer-Verlag, 1994.
- Cox, M. T., Munoz-Avila, H., & Bergmann, R. (2006). Case-based planning. To appear in *Knowledge Engineering Review*.
- Weld D. (1994) An Introduction to Least Commitment Planning. *AI Magazine*, 15(4), pages 27-61. AAAI Press.
- Kuchibatla, V. (2006) TransUCP: An Analysis on Transformational Analogy. MS Thesis. Computer Science and Engineering. Department. Lehigh University.