2006

Transformational analogy : a general framework, semantics and complexity

Sarat Chandra Vital Kuchibatla Venltata
Lehigh University

Vithal Kuchibatla
Venkata, Sarat
Chandra

Transformational Analogy: A General Framework...

May 2006

Transformational Analogy:
A General Framework, Semantics and Complexity

By

Sarat Chandra Vithal Kuchibatla Venkata

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

May 2006

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

April 24 06
Date

Thesis Advisor

Chairperson of Department

Acknowledgements

I would like to thank Dr. Hector Muñoz-Avila for his support and guidance and the numerous stimulating discussions without which, this thesis would have been impossible.

Finally, I would like to thank my family and friends for their constant support and encouragement when it was most required.

Table of Contents

List of Tables

List of Figures

Abstract

This thesis presents TransUCP, a general framework for transformational analogy. The particulars and working of this framework are further explained with relevant working examples. It is proved, using TransUCP, that transformational analogy does not meet the worst-case complexity scenario of Nebel and Koehler (1995). By generating counter-examples, the framework is also used to prove that their results about plan adaptation being harder than planning from first principles do not apply to it. The implementation details of TransUCP and the results derived from the experimentation are also explained. A synopsis of the TransUCP framework has been accepted for publication at the upcoming European Conference for Case-Based Reasoning (ECCBR-06).

1 Introduction

Planning can be defined as the process generating a solution for a well-defined problem. More precisely, it can be defined as generating a set of actions to be executed to achieve a specified set of goals, given the initial conditions (Bergmann *et al.*, 1996). Case-based planning is a classical planning technique where in solutions of previously solved problems are reused to solve new ones. Two main approaches of case-based planning are derivational analogy and transformational analogy.

Over the years, derivational analogy, a 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.

In this thesis, TransUCP, a general framework for transformational analogy built on top of the universal classical planning model is presented. 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. 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.

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.

Using the TransUCP framework, it is demonstrated that transformational analogy does not meet the worst-case complexity results of Nebel and Koehler (1995), and therefore, their results about plan adaptation been computationally harder than plan adaptation does not apply to it. This is proved by constructing a counter-example in which a crucial condition is not met. Furthermore, experiments are performed that demonstrate that this counter-example is not en exception. Rather, these experiments show that it is very unlikely that transformational analogy falls under the scenario described in Nebel and Koehler (1995).

2. Preliminaries and Related Work

In this section, we touch upon those topics and definitions which form the basis of case-based planning and related topics. TransUCP, the transformational planner and the primary focus of this document, being introduced in later sections as, is built upon universal classical framework, whose underlying concepts and the data structures used are explained in this chapter. Our general framework enhances the SPA system (Hanks and Weld, 1995), to other forms of planning by taking advantage of the universal classical planning (UCP) framework.

2.1 Logistics Domain

In this section, we describe the logistics domain which is used to explain the terms and concepts defined and used in the following sections. The same domain will be used to through out the length of this document to explain various concepts and to illustrate examples.

In the domain being currently used, there are different packages located initially at various geographic locations or cities and some or all of these packages have to be transported to specific locations. There are also means of transportation such as trucks located at various cities and these are used to move the packages. Table 1 shows the set of available operators in this domain, their descriptions, their pre-conditions and effects. A description of the steps and the conventions followed in their use are given below:

- The action L (P, V, L_c) indicates that the vehicle V is ***loading*** the package P from location L_c.
- UL (P, V, L_c) indicates that P is ***unloaded*** from V onto location Lc.
- MV (V, L_1, L_2) indicates that V is ***moved*** from L1 to L2.

Operator	Pre-conditions	Effects	Description
MV(truck, loc1, loc2)	The truck is initially at loc1	The truck is at loc2	Moves the truck with its contents from loc1 to loc2
L(package, truck, loc1)	The truck is at loc1 AND The package is at loc1	The package is in the truck at loc1.	Loads the package in to the truck
UL(package, truck, loc1)	The truck is at loc1 AND The package is in the truck	The package is at loc1.	Unloads the package from the truck

Table 1: Operators in the transportation domain

2.2 Partial Order Plan

The algorithm proposed in this thesis 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 $<T, O, B, L>$ 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.
    - 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 followed by step t_j.

Using the above mentioned definition of a partial plan, we define the following terms related with a partial plan problem:

Initial Step: The step t_0, called the ***initial step*** has no preconditions and has as effects the conditions that are true in the opening state.

Final Step: The step t_f, which has no effects and has as pre-conditions the goals to be achieved, is called the ***final step.***

Null-Plan: A null-plan can be defined as a plan where

$T = \{t_0, t_\infty\}$,

$O = \{(t_0 < t_\infty)\}$, and,

$B = L = \text{null}$

Condition: 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. For, example, in the logistics domain, MV and L are operators and MV (V_1, B, C) is a step. 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 ($t_m \rightarrow^Q$) 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. For example, for the operator *L (package, truck, loc1)*, which loads the package at location loc1 into the truck, the preconditions (that both the truck and the package must be at location loc1) have to be satisfied.

Open condition: If a condition that needs to be true for the execution of a step has not been satisfied, it is said to be an ***open condition***. An open condition is a constraint of the form ($\rightarrow^Q t_j$) where the condition Q has to be true for step t_j and there is no step before t_j whose post condition satisfies Q.

Threat: 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.

Flaws: Open conditions and threats in a partial plan are together referred to as ***flaws*** of the plan.

Planning Problem: A planning problem is a 2-tuple *(Initial, Goal),* where *Initial* is the set of initial conditions and *Goal* is the set of desired goal conditions.

Solution Plan: If every safe ground linearization of a plan takes the initial state to the goal state, and the plan has no open conditions or threats, it is said to be a ***solution plan*** for the given problem.

We now illustrate a planning problem and its partial plan with an example in the logistics transportation domain. In the problem shown in Figure 1, there are four locations, namely: Location A, Location B, Location C and Location D. There is a package P1 located at Location A and another package P2 at Location D. The planning problem requires package P1 in location A, and package P2 in location B, to be relocated in location C. The figure also shows two trucks V1 and V2 at locations A and D respectively.

Figure 2 shows a solution plan for the problem depicted in Figure 1. Arrows denote either ordering or interval preservation constraints. The plan consists of 11 steps, denoted by t_k. It can be seen that after t_3 is executed, there are two possible orderings of execution of steps as depicted below.

... $t_3 \rightarrow t_7 \rightarrow t_8 \rightarrow t_9 \rightarrow t_6$ OR ... $t_3 \rightarrow t_8 \rightarrow t_9 \rightarrow t_7 \rightarrow t_6$

Both these sequences satisfy the partial ordering constraints of the plan. 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.

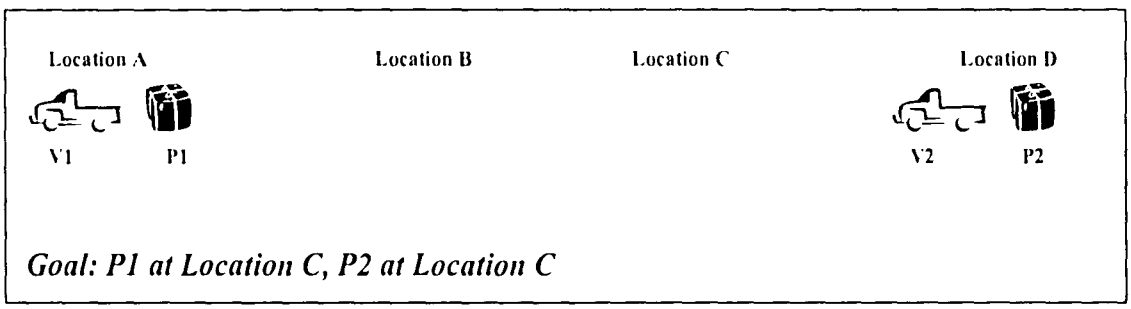

Figure 1: Planning problem in the logistics domain

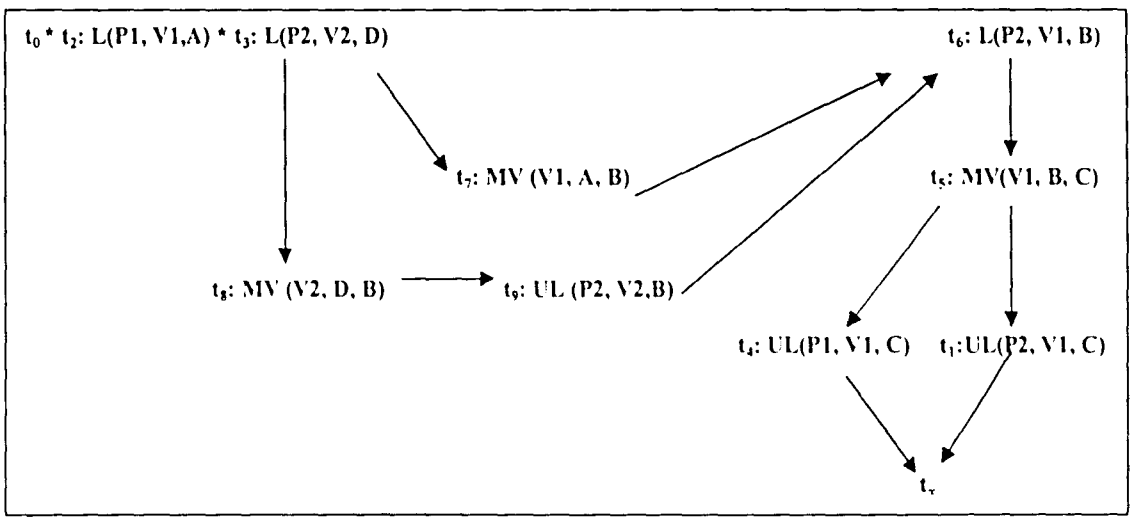

Figure 2: Partial plan solving the problem in Figure 1

2.3 Total Order Planning

Having looked at partial order plans, we now describe an alternative kind of planning called total order planning. Total ordering planning can be represented by the 4-tuple $<T, O, B, L>$, which is similar to the definition mentioned in Section 2.2, except that the set of constraints, L, here has only contiguity constraints. That is, total order plans specify a linear sequence of steps and the order of execution of these steps have to be maintained in accordance with the sequence. For example, a total order solution plan for the planning problem mentioned in Figure 1 is shown in Figure 3 below.

$$t_0 * t_2 * t_3 * t_7 * t_8 * t_9 * t_6 * t_5 * t_4 * t_1 * t_\infty$$

Figure 3: Total order solution plan for the planning problem in Figure 1

2.4 Universal Classical Planning

In this section, we define a popular planning framework called the Universal Classical Planner, upon which we build to form the TransUCP planner.

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, 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**

ii. **Backward State Space Plan Refinement**

iii. **Plan Space Plan Refinement**

2.4.1 Forward state space plan refinement

A **head step** of a partial plan is defined as a step t_j of the plan where $t_0 * t_1 * \ldots * t_j$ and there is no step t` such that $t_j * t`$. The sequence of steps $t_0 * t_1 * \ldots * t_j$ 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. Figure 4 shows the pseudo-code for a forward state space planner.

Function **RefinePlanForwardStateSpace** (Plan P)
Returns: Plan

//Operator Selection
Non-deterministically select one of the following:

Non-deterministically select a step t_i from head-fringe of P, such that all preconditions of the operator O (that takes the head fringe to t_i) are satisfied in head fringe of P.

OR

Non-deterministically select an operator O from the operator library, such that all preconditions of the operator O are satisfied in head state t of the plan. Create a new step name t_j.

//Operator Application
Let t_j be the step selected above. Add the contiguity constraint ($t_i * t_j$) to P where t_j is the current head step and associate this step with the purpose (Step Added, t_j, forward state).

Return P

Figure 4: Forward State Space Planning

2.4.2 Backward state space plan refinement

A **tail step** of a partial plan is defined as the step of t_j of the plan where $t_j * t_{j-1} * \ldots * t_\infty$ and there is no step t` such that t`* t_j. The sequence of steps $t_j * t_{j-1} * \ldots * t_\infty$ 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. Figure 5 shows the pseudo-code for a backward state space planner.

Function **RefinePlanBackwardStateSpace** (Plan P)
Returns: Plan

//Operator Selection
Non-deterministically select one of the following:

Non-deterministically select a step t_i from tail-fringe of P, such that
a. None of the effects of the operator O (that takes the t_i to the tail state) negate the facts in the tail state, and,
b. At least one effect of the operator O is present in the tail state

OR

Non-deterministically select an operator O from the operator library, such that
a. None of the effects of the operator O negate the facts in the tail state and
b. At least one effect of the operator O is present in the tail state of P.

Create a new step name t_i.

//Operator Application
Let t_i be the step selected above. Add the contiguity constraint (ti * t_j) to P where t_j is the current tail step and associate this step with the purpose (Step Added, t_i, backward state).

Return P

Figure 5: Backward State Space Planning

2.4.3 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. Once a flaw is selected, it is rectified using the following methods.

If it is an open condition ($\rightarrow^{Q}$ t), it is resolved by

- Selecting an existing step t` in the plan and ordering it before t such that, t` has Q as a subset of its post-conditions.
- Selecting one of the available operators and adding it to the plan as a new step t` and ordering it before t such that, t` has Q as a subset of its post-conditions.

There can be more than one way to reorder the existing 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, all of which have that particular flaw rectified. As a result, the RefinePlanSpace algorithm, shown in Figure 6, returns a list of plans as opposed to a single plan. After the constraints to be added are decided, the corresponding binding constraints also have to be added to the plan. This again, can be done in more than one way and results in multiple plans. Figure 7 shows this AddLinksAndBindings function.

```
Function RefinePlanSpace (Plan P, Flaw F)
Returns: List of plans

Define L: a local list of plans
If F == null THEN
   F = Select a flaw from P

IF F is an open precondition of the form (→^Q t_j)
THEN
        For each step t_i currently in P
        Do
          IF t_i can be ordered before t_j, and t_i adds a condition unifying with Q THEN
          Add the plans returned by AddLinksAndBindings (t_i, Q, t_j, P) to a local list L

        For each operator O whose post conditions contains a condition unifying with
        Q
        Do
          Add a step t_k, which applies the operator O to P with the purpose (Step
          Added, t_k, plan space)
          Add the constraints (t_k > t_j), (t_k > t_∞) and (t_0 > t_k) to P.
          Add the plans returned by AddLinksAndBindings (t_k, Q, t_j, P) to a local list L

        Return all the plans in the lists L

Else return ResolveThreat (F, P)
```

Figure 6: Plan Space Planning – Resolving an open condition

If the selected flaw is a threat, it is handled by a "Resolve Threat" function. Given a threat of the form (t_k, ti→^{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 8 shows the pseudo-code for the ResolveThreat function.

Function **AddLinksAndBindings** (Step t_i, Condition Q, Step t_j, Plan P)
Returns: List of plans

Define L: a local list of plans
For each set of bindings B causing t_i to assert Q
DO
  P_0 = a copy of P
  Add a new link $t_i \rightarrow^Q t_j$ to P_0
  Add the ordering constraint $t_i < t_j$ to P_0, associated with purpose (establish link, $t_i \rightarrow^Q t_j$)
  Add B to P_0, tagged with R
  Add the plan P_0 to local list L

Return all the plans in the list L

Figure 7: Plan Space Planning – Adding the appropriate constraints

Function **ResolveThreat** (Threat (t_k, $ti \rightarrow^Q t_j$), Plan P)
Returns: List of plans

Define L: a local list of plans
IF t_k can be consistently ordered before t_i
THEN
  P_0 = a copy of P
  Add the constraint $t_k < t_i$ to P_0, associated with the purpose (protect ((t_k, $ti \rightarrow^Q t_j$)))
  Add the plan P_0 to local list L

IF t_k can consistently be ordered after t_j
THEN
  P_0 = a copy of P
  Add the constraint $t_j < t_k$ to P0, associated with the purpose (protect ((t_k, $ti \rightarrow^Q t_j$)])
  Add the plan P_0 to local list L

For each set of bindings B that prevents t_k's effects from unifying with Q
DO
  P_0 = a copy of P
  Add constraints $t_i < t_k$ and $t_k < t_j$ to P_0, both associated with the purpose (protect ((t_k, $ti \rightarrow^Q t_j$))
  Add B to P_0, tagged with R
  Add the plan P_0 to local list L

Return all new plans in the list L

Figure 8: Plan Space Planning – Resolving a threat

In the pseudo-codes illustrated in figures 4 through 7, we have introduced the term "***purpose***" and mentioned ***"associating a purpose"*** with steps or constraints. These terms will be explained in the ensuing chapters of this document.

Figure 9 illustrates the working of a universal classical planner. The plan is represented using a directed graph, where each node (shown as a dashed circle) is a partial plan. Dashed arrows indicate ordering constraints while solid arrows indicate contiguity constraints. The figure shows us the first four refinements done on the plan.

The planner initially starts with a null plan (containing only the initial and final state). As mentioned previously, during each iteration, the UCP can choose one of the above three refinements and modify the partial plan according to the selected refinement strategy.

In the figure below (Figure 9), the planner performs plan space refinement in the first iteration. Subsequent iterations perform backward state space, forward state space and plan space refinements, in the specified order, to the plan. The nodes in the figure are numbered to specify the sequence in which they were added to the partial plan.

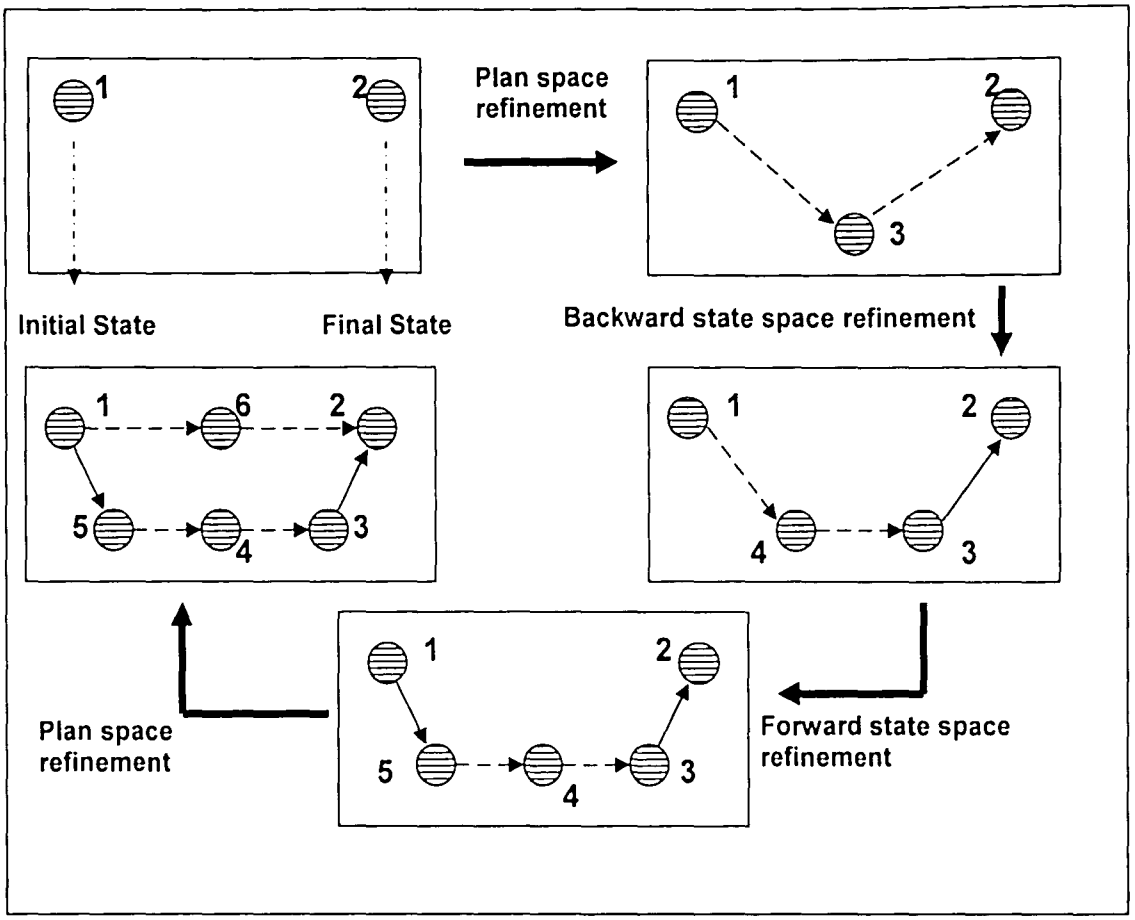

Figure 9: Universal Classical Planning

2.5 Transformational Analogy

Cased-based planning, in general, is viewed upon as to be done using two different approaches – ***transformational analogy*** and ***derivational analogy***. It is important to understand the underlying ideas and paradigms of these approaches as they are instrumental to fully comprehend the results of the work done, which are explained in later sections of this document.

Transformational analogy is a problem-solving technique in which a pre-selected plan is modified to solve a new problem (Carbonell, 1983). By looking for a similar

solution and *copying* it to the new situation, making suitable substitutions where appropriate, transformational analogy transforms it into the target solution. Possible modifications to the plan include

- Removing step(s) from the plan
- Adding new step(s) to the plan
- Changing the parameter(s) of the steps in the plan (binding constraints)
- Addition/removal of ordering/contiguity constraints in the plan

Since the starting point of a planner adopting transformational analogy method is the retrieved solution plan for a previously solved planning problem from the case-base, a transformational planner does not build a plan solution "from scratch". That is, given a solution plan, it starts modifying and refining it until it solves the current planning problem at hand. The modification and refinement techniques followed may vary from one planner to the other. 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. TransUCP, the planner which is the primary focus of this thesis, is a transformation planner and can be looked upon as an extended version of the SPA system.

To illustrate the idea better, consider the following example. If we have a solution plan for transporting the package P_1 from New York to Boston and solution uses the route New York → Rhode Island → Boston. Now, if a new package needs to be transported

from New York to some city in New Hampshire, a planner using transformational analogy would most probably reuse the original solution and would build the plan (route) from Boston to New Hampshire. The figure below (Figure 10) illustrates this example. The route on the left is the solution plan retrieved from the case-base and the one on the right is what a planner, adopting transformational analogy methods, would have come up with as a solution plan. The last leg of this route (shown as a dotted connector), could have been generated from first principles.

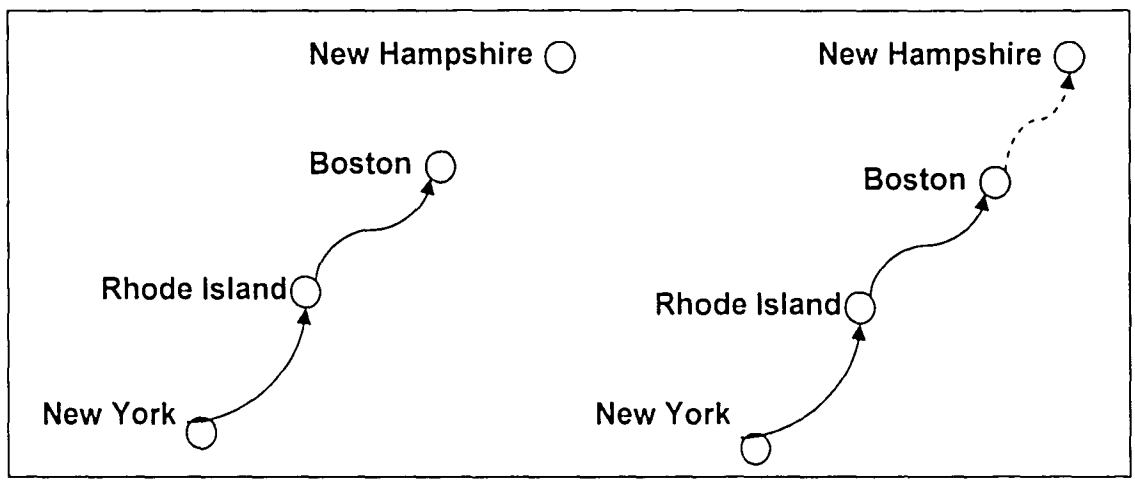

Figure 10: Transformational Analogy Example

2.6 Derivational Analogy

The second methodology used with case-based planning is derivational analogy. Derivational analogy stores problem solving decisions and their justifications in the source plan and replays them for the target (Carbonell, 1986; Veloso, 1994). That is, given a planning problem and a solution plan to a previous problem, a derivational planner looks at the history of the problem solution, the steps involved and the decisions taken at every stage that led into those steps being added into the solution plan.

Derivational analogy makes use of this information to arrive at a solution for the current planning problem.

Unlike a transformational planner, which starts from the solution plan from the case-base, a derivational planner has, as starting point, a null-plan and starts building from there on. It tries to reuse the same decisions as those used in the case-plan, thereby probably replicating some of the steps used in the plan. But it does not copy or reuse the steps from the solution plan directly.

As opposed to transformational analogy, which only looks at the final solution of the case-plan, a derivational planner looks at how the problem was solved. To further elucidate, derivational planners makes use of solution derivation, also known as derivational traces, which are sequences of planning decisions that have led to previous solution plans, and are reused when solving a new problem (Veloso & Carbonell, 1993). Derivational traces are decision sequences that point towards the decision choices that the planner must make to generate the solution plans (Muñoz-Avila, Nau & Au, 2002).

3 TransUCP

In this chapter, we explain the details of the TransUCP algorithm and its constituent elements and sub-functions. The main idea behind the TransUCP algorithm is to solve the planning problem by using transformational analogy with the Universal Classical Planning framework. The inputs to the algorithm are a planning problem and the case library. TransUCP returns the solution plan or a failure message if it could not generate one.

3.1 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 or 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 t_j 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 t_j 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, $t_i \rightarrow^Q t_j$)).

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 ($\rightarrow^Q t_j$).

For example, in the plan shown in Figure 16, the step t_2: L (P1, V1, A), which is added to the plan during forward space refinement, has the purpose tag *Purpose (Step Added, t_2, forward state)*.

In the plan shown in the figure, the goal is to move the packages P1 and P2 at location D. Therefore, one of the open conditions that the final state t_∞ had was ($\rightarrow^Q t_\infty$) where Q is the condition that the package P1 should be in location D. To nullify this open condition, the step t_4: UL (P1, V1, C) has been added during the plan space refinement. Therefore, this step t_4 has the following purpose tag *Purpose (establish link, $t_4 \rightarrow^Q t_\infty$)*.

3.2 The Algorithm

The inputs to the algorithm: the planning problem and the case library, are given to the starting function, TransUCP. This function returns the solution plan or a failure message if it could not generate one. The planning problem is specified by a 2-tuple <*Initial State, Goal State*>, where the initial state is the set of conditions which are true at the start of the problem and the goal state is the set of all the desired conditions that need to be true in final state of the problem. Figure 11 shows the pseudo code of this function.

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 document 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, through the function AdjustExactly. 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, the plan returned by AdjustExactly, called here as AdjustedPlan, might have open conditions and threats. 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, <AdjustedPlan, UP> and <AdjustedPlan, DOWN> to the PlanPool. The purpose of the direction indicators, UP and DOWN is discussed in detail in momentarily. This collection of plans, PlanPool, is passed to the function TransformPlan, which returns the solution plan. Figure 12

illustrates the pseudo code of the function TransformPlan.

TransformPlan function is called recursively until a solution plan is arrived at or a failure is returned, which happens when the PlanPool is extinguished. This function makes non-deterministic choices at various points. It takes PlanPool as an argument and chooses a plan from it non-deterministically and deletes it from the PlanPool. It then checks if it is a solution, and if so, returns it. If not, it checks for the direction pointer of the plan P.

```
Function 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
```

Figure 11: TransUCP Algorithm

Progressive Refinements

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. The three kinds of refinements used in the universal classical planning algorithm (Section 2.2) constitute the progressive refinements. Each progressive refinement done to a plan has a purpose tag associated with it.

```
Function 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

  //Progressive Refinements
  If D == DOWN
  THEN
        Non-deterministically, select any one of

        1. P' = RefinePlanForwardStateSpace (P)
              Check if P' is a solution. If yes, return P'.
              Add <P', DOWN> and <P', UP> to PlanPool

        2. P' = RefinePlanBackwardStateSpace (P)
              Check if P' is a solution. If yes, return P'.
              Add <P', DOWN> and <P', UP> to PlanPool

        3. RefinePlanSpace (P)
              For each plan P_i returned by RefinePlan (P, null) do
                    Check if P_i is a solution. If yes, return P_i.
                    Add <P_i, DOWN> to PlanPool

  //Non-progressive Refinements
  ELSE IF (D == UP)
  THEN
        Add all elements of RetractRefinement (P) to PlanPool

  //Recursive Invocation
        TransformPlan (Initial, Goal, PlanPool)
```

Figure 12: Pseudo code – TransformPlan

In the TransUCP function, all plans with direction indicators DOWN would go through progressive refinements and all those which have UP as direction pointers are refined non-progressively.

If the direction pointer of the plan selected in TransformPlan is DOWN, it performs progressive refinements on it. It does this by non-deterministically choosing one of the three possible refinements to be applied. The three kinds of refinements used in the universal classical planning algorithm (Section 2.4) constitute the progressive refinements.

Once a particular refinement strategy is chosen, all the plans returned by that refinement function are added to the PlanPool. These plans have the appropriate purpose tags associated with the refinements done on them. TransformPlan is then called in a recursive fashion until a solution plan or a failure is encountered.

Non-progressive Refinements

All refinements made to a plan that are not progressive refinements are termed as non-progressive. Non-progressive refinements basically undo one or more of the progressive refinements done to the plan previously. All those plans in PlanPool that have UP as the direction pointer go through non-progressive refinements.

If the direction pointer is UP, TransformPlan calls the function RetractRefinement with the selected plan as an argument. RetractRefinement selects, non-deterministically, one of the purpose tags in the plan and retracts the refinement associated with that purpose tag. Basically, this 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 thesis. A good heuristic to be followed for this selection process can be found in Hanks and Weld (1996). The progressive refinements to undo can be any of the three kinds of refinements used in the universal classical planning algorithm (Section 2.4).

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_1 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. Figure 13 shows the pseudo-code of the RetractRefinement function.

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

Define L: a local list of plans

R = select a purpose
(F, P0) = RemoveStep(R, P)
Add the tuple (P0, UP) to local list L

IF purpose in R was forward state space refinement
THEN
        For each plan P1 returned by RefinePlanForwardStateSpace (P0)
        Do
                If P1 does not map onto P, add <P1, DOWN> to list L.

Else IF purpose in R was forward state space refinement
THEN
        For each plan P1 returned by RefinePlanBackwardStateSpace (P0)
        Do
                If P1 does not map onto P, add <P1, DOWN> to list L.

Else IF purpose in R was partial plan space refinement
THEN
        For each plan P1 returned by RefinePlanSpace (P0, F)
        Do
                If P1 does not map onto P, add <P1, DOWN> to list L.

Return all plans, direction pairs collected in list L.
```

Figure 13: Pseudo code - RetractRefinement

Figure 14 clearly illustrates the flow of the TransUCP planner. TransUCP takes a plan from the case base, adjusts it and adds it to the plan pool. Only when the plan pool is empty is when the planner reports failure. Otherwise, it selects a plan from the pool and depending upon the direction pointer, performs the appropriate refinement. If it is DOWN, it performs a progressive refinement, selecting non-deterministically from the three. If not, it calls RetractRefinement, refining non-progressively. In both cases, the resulting plans are added to the plan pool. This process is called recursively until a

solution plan is found when success is reported and the planner terminates.

This completes the explanation of the TransUCP algorithm. In the subsequent section, the working of the algorithm is elucidated with examples.

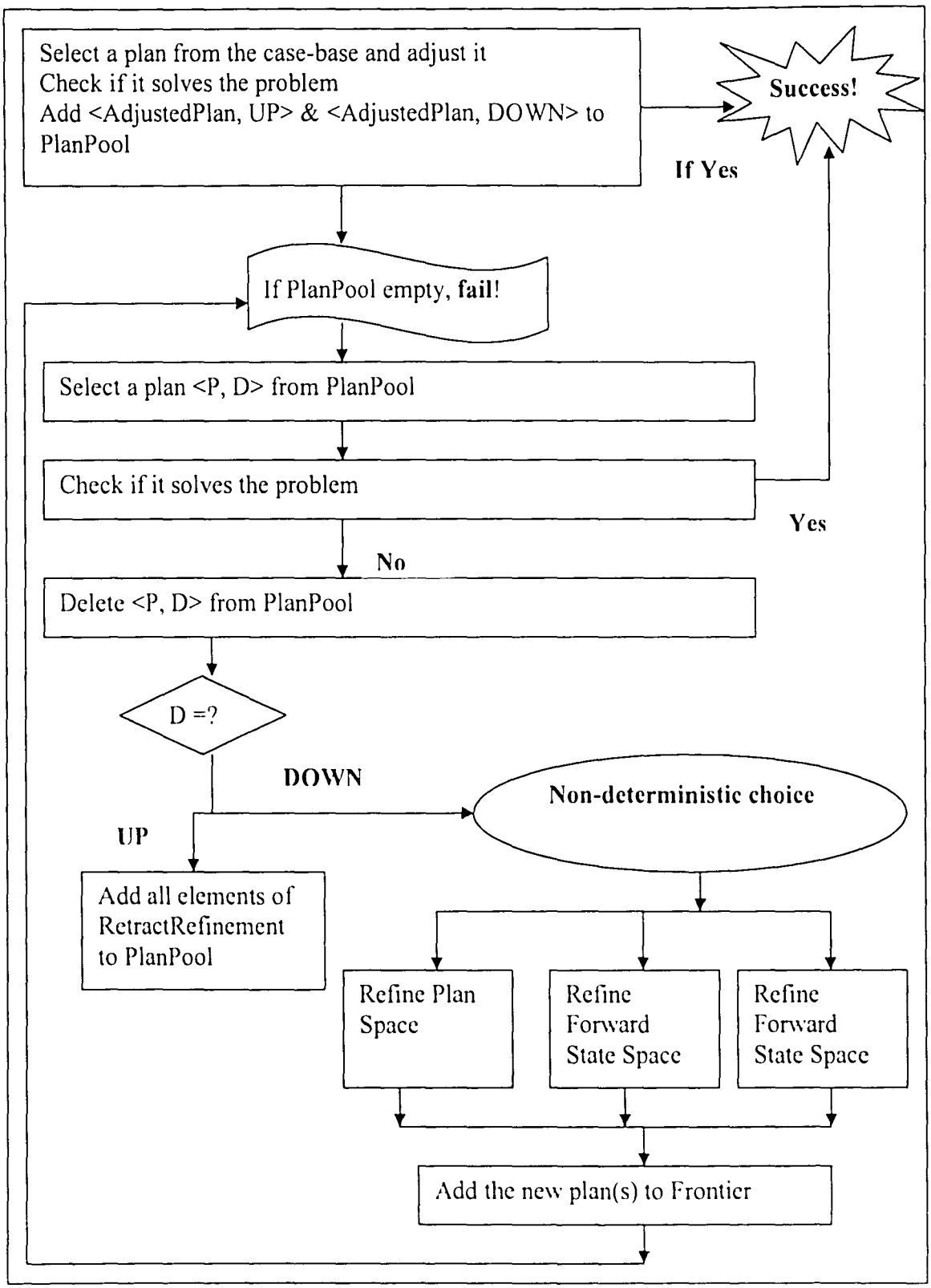

Figure 14: TransUCP Flow Chart

3.3 Example of TransUCP

In this section, we explain the working of the TransUCP with an example. 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 (shown in Figure 15) and its solution plan described in Figures 16. We take this as the input case retrieved from the case library and try to solve the new planning problem, shown in Figure 17, using TransUCP.

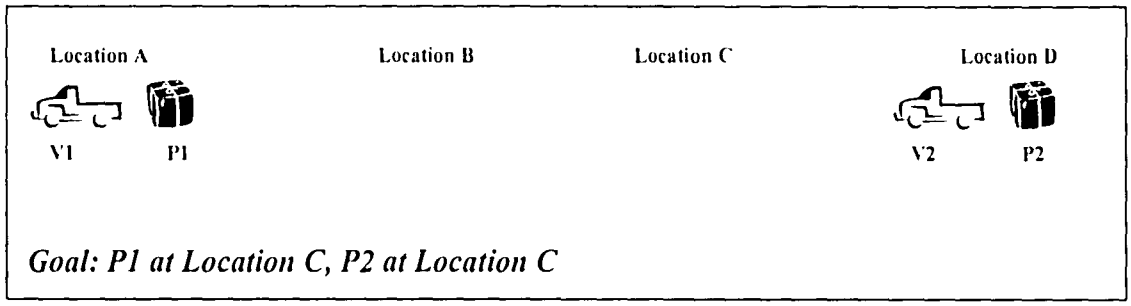

Figure 15: Planning problem in the logistics transportation domain

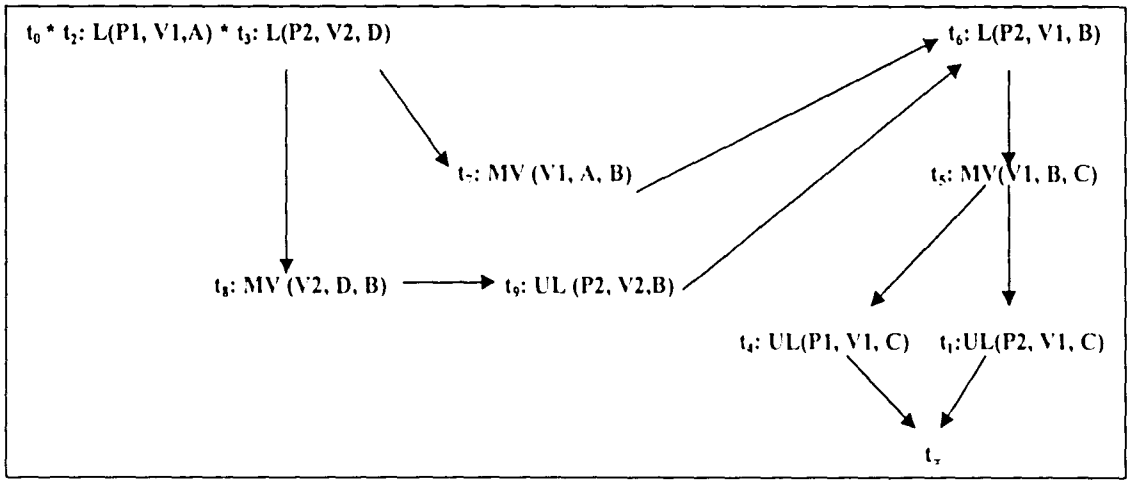

Figure 16: UCP generated solution plan for the problem in Figure 15

The solution plan show in Figure 16 has been generated by UCP. In the new problem to

be solved, shown in Figure 17, we have the same goals, i.e. to relocate package p1 and p2 into location C. The difference is that this time there is no truck in location D.

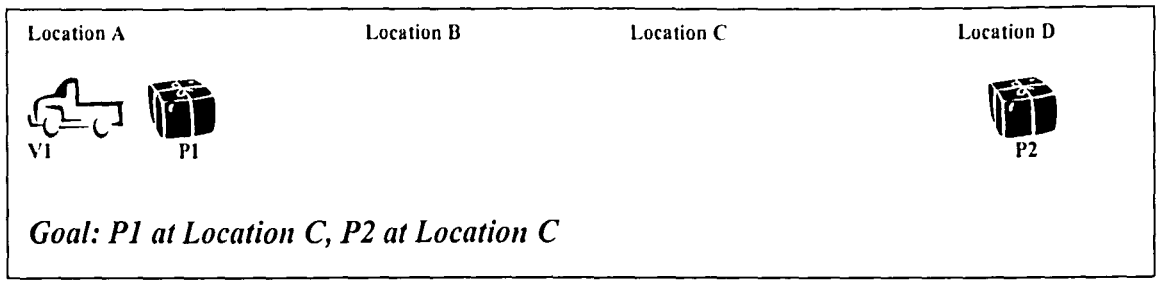

Figure 17: Planning problem to be solved by TransUCP

When this problem is passed to TransUCP, it retrieves a plan from the case library, a planning problem and its solution that is similar to the current problem at hand. Let us assume that the ones retrieved are those shown in Figure 15 and 16.

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 18. 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 <P_1, UP> and <P_1, DOWN> to the PlanPool and this pool is passed to TransformPlan. Let us assume that TransformPlan non-deterministically chooses the pair

<P_1, DOWN> from the plan pool to refine. Since the direction pointer is DOWN, it ha to perform a progressive refinement on the plan P_1. That is, it can choose between forward state, backward state and plan space refinements to perform on P_1.

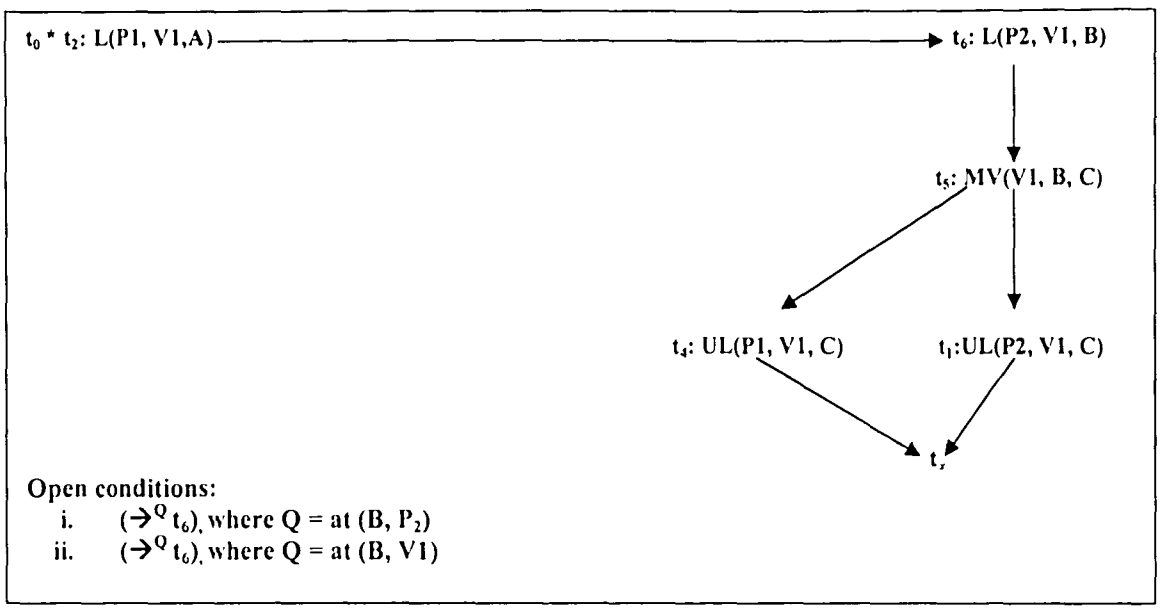

Figure 18: Partial plan P_1 after initial adjusting

Assuming without the loss of generality that the refinement strategy chosen is forward state space refinement, the control is now passed on to the RefinePlanForwardStateSpace function. It has to choose an existing step within the head fringe (in this case only step t_6) or add a new step, such that, the preconditions of the step to be added are satisfied in the header of the plan. If it is chosen to add a new step, a new step t_8 MV (V1, A, D) can be appended to the current head step t_2 with the contiguity constraint t_2 * t_8 as all the preconditions of t_8 are satisfied at t_2. The resulting plan, labeled as P_2, is shown in Figure 19 and the 2-tuple <P_2, DOWN> is added to the PlanPool. After t_8 has been added to P_2, the flaws of P_2 are analyzed and these are also shown in Figure 19.

During the second pass of TransUCP, the PlanPool contains the pairs (<P1, UP>, <P_2, DOWN>, <P_2, UP>). If the pair chosen by TransUCP to refine was <P_2, DOWN> and the progressive refinement chosen was partial plan space refinement, the control is now passed to the RefinePlanSpace function.

Let us assume that the flaw selected by this function to resolve is $(\rightarrow^{Q} t_6)$, where Q = at (B, *truck*). It can resolve this open condition by either reordering steps, i.e. by adding ordering constraints to the plan or by adding a new step to the plan and ordering it before t_6. Either way, the condition Q has to be made true before t_6 is carried out. We can assume that a new step t_9 : MV (V1, A, B) was added and ordered before t_6 with the ordering constraint $t_8 < t_9 < t_6$. This resolves the flaw and results in plan P_3 shown in Figure 20. 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 and is shown in the figure.

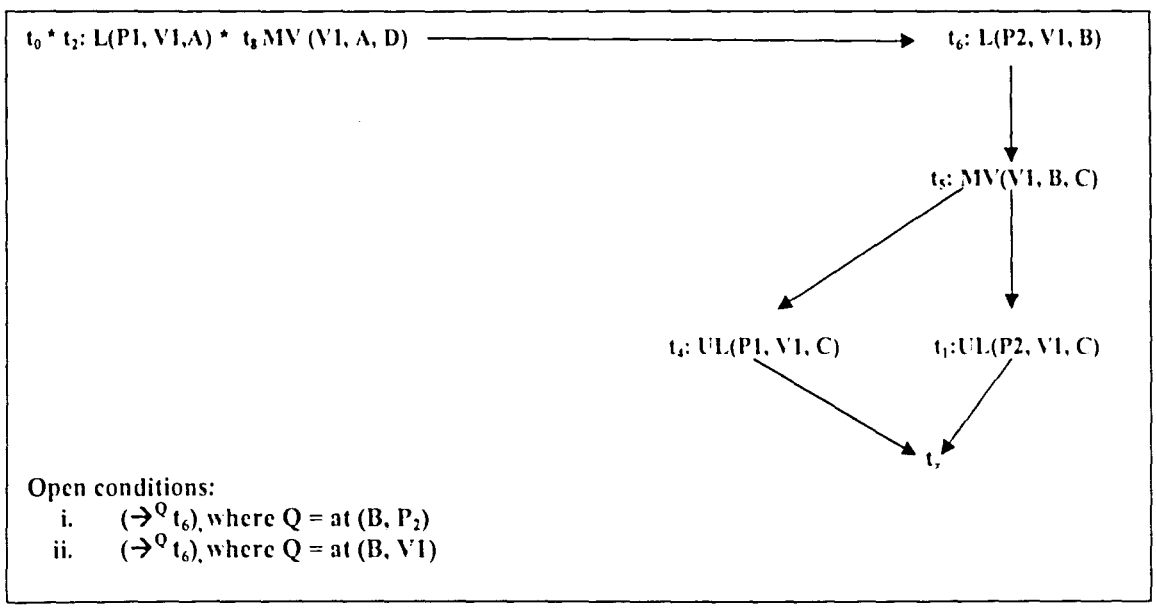

Figure 19: Partial plan P_2

Continuing in this manner TransUCP continuously keeps refining the partial plan and searches for the first solution node. During each pass, it may choose to perform progressive refinements and add steps or it may choose to perform non-progressive refinements and retract some of the decisions previously made. It reports the first partial plan that it encounters that satisfies the required conditions for being a solution plan and returns it. Figure 21 shows one of the possible solution plans returned by TransUCP.

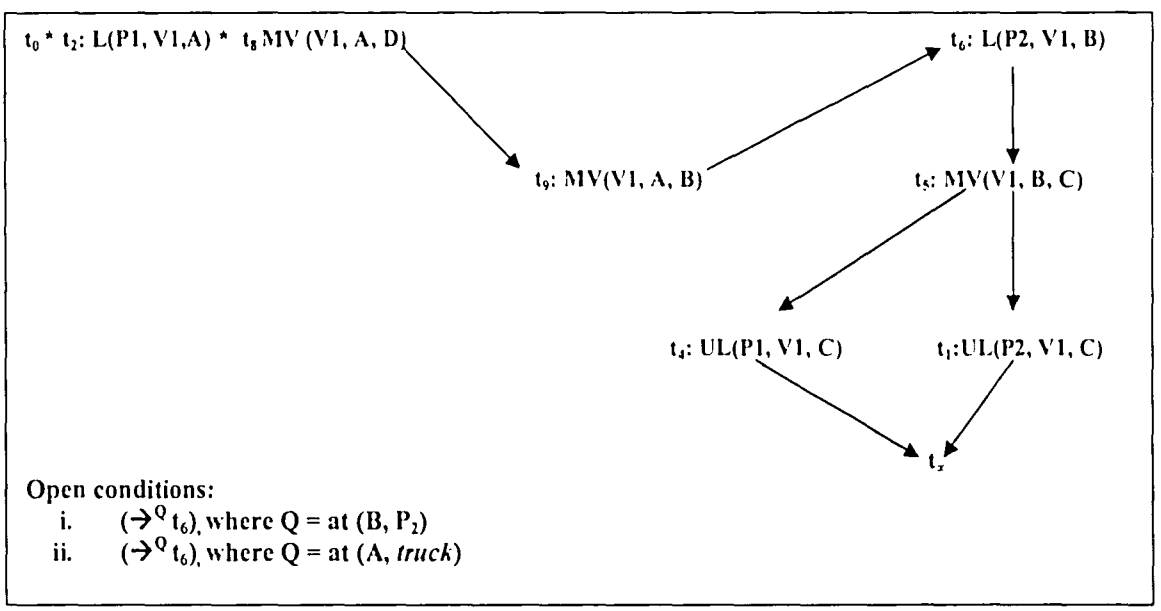

Figure 20: Partial plan P_3

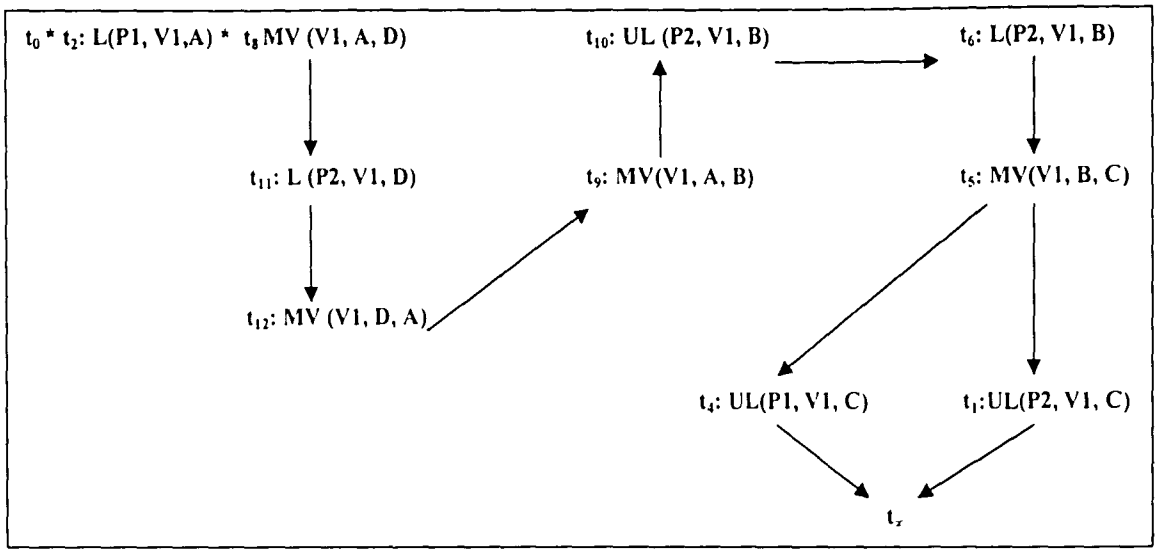

Figure 21: Solution plan generated by TransUCP

4 Complexity Results and Search Space

In this section, we explain how the TransUCP performs and explores the search space to find the solution nodes. Further, the performance of the algorithm is analyzed and it is proved that this framework does not satisfy the conditions of being a conservative planner. By doing so, it is also proved that transformational analogy does not come under the worst-case complexity of Nebel and Koehler (1995).

4.1 Search space

TransUCP generates a solution for a given planning problem by treating the solving procedure in effect as a searching problem. It searches through the plan search space, defined below, to find the solution nodes.

State Search Space

A state search space can be defined as necessarily being a graph (Weld, 1994), where:

- Each node represents a state of the world
- Arcs between nodes A and B indicate that there is an operator which takes the world from one state (represented by A) to the next (represented by B). The edges would be in-directed if the actions are reversible; else, they would be directed.
- The solution plan to a planning problem represented thus would be a path (a sequence of connected nodes), starting from the initial node and ending at the final node.

One of the advantages of solving a planning problem using a state search space is that a variety of search algorithms can be used: breadth first search or A* search to name a few. Another observation to be made is that all the possible combinations of valid states that the world can be are represented as nodes in the graph. This would make the graph very large.

Figure 22 shows a sample state search space. The domain used is that of the STRIPS system (Weld, 1994). In this domain there are three blocks, named A, B and C, and they are originally in the initial state as shown in the figure. The desired final state is to place them as A on top of B and B on top of C and C is on the table, as shown in figure. Individual nodes in the graph show the different states of the world and the solution plan is any path from the initial state node to the final state node. One such solution path is shown in dashed lines.

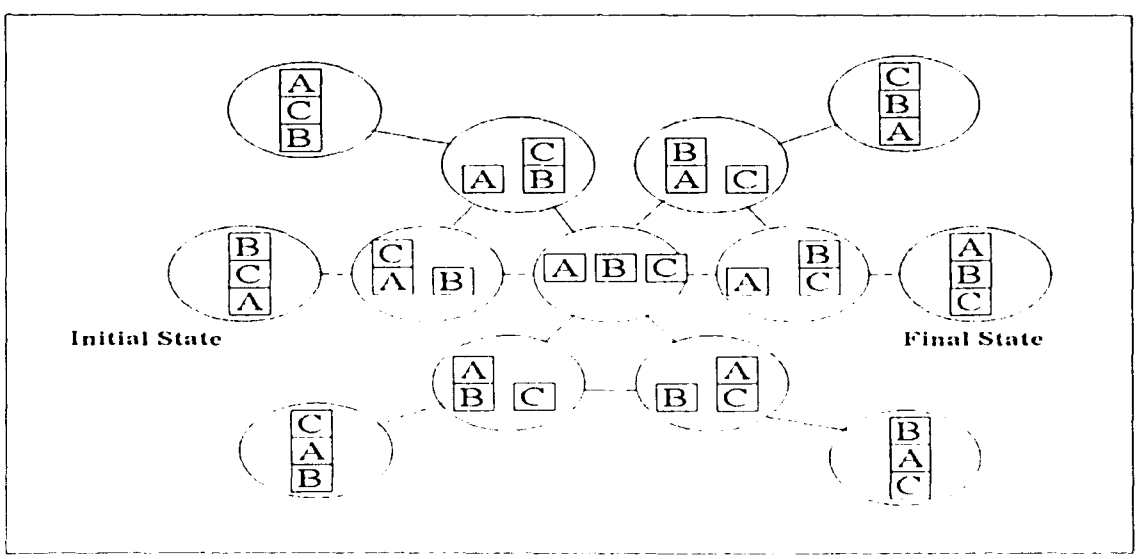

Figure 22: State Search Space of STRIPS (figure taken from Weld, 1994)

Plan Search Space

A plan search space can be defined as necessarily being a directed graph, where:

- Each node represents a partial plan
- A directed edge from node A to node B represents that the partial plan B results by performing one or more refinements on partial plan A.
- The solution plan to a planning problem represented thus would be a node, which represents a partial plan that solves the planning problem.

One of the advantages of using plan search space as a representation format is that domain dependent heuristics can be applied in the search process for the solution nodes. Like state search space, the plan search space graph can also be very large.

4.2 Traversal of the search space by TransUCP

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, for the purposes of this document, referred to as the parents of the node and similarly, the nodes resulting from performing progressive refinements are referred to as the children of the node.

It is to be noted that the graph being searched does not necessarily have to be 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 the problem (initial and goal states) to be solved, TransUCP first modifies the case plan so as to match its initial and goal states to those of the given problem. Once this plan adaptation 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 23). This node would be an inner node in the graph. In the figure, nodes marked Node A, Node B and Node C are shown as the probable solution nodes which represent plans that qualify to be solutions.

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. 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.

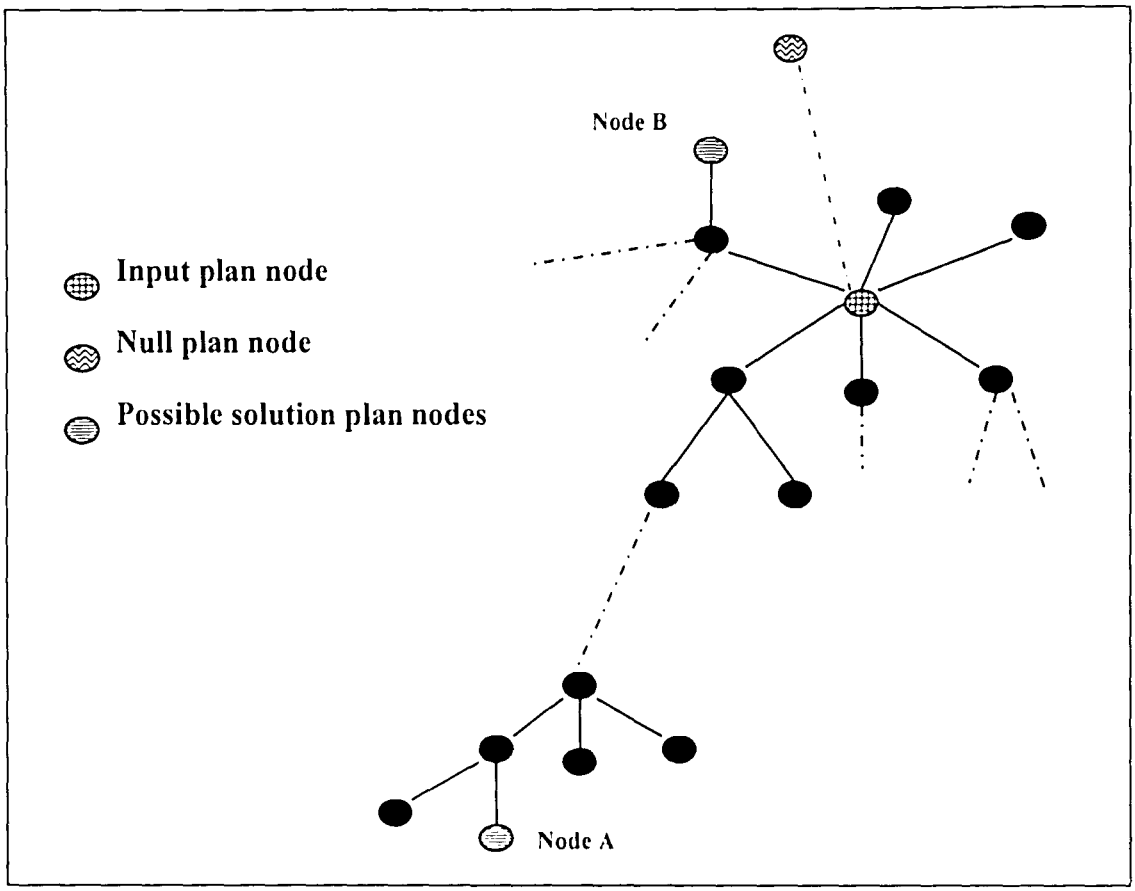

Figure 23: Graph Traversal by TransUCP

4.3 Properties

We shall now analyze the properties of TransUCP and check if it is conservative in the sense of (Nebel & Koehler, 1995). We will make use of the definitions below, taken directly from (Nebel & Koehler, 1995), for this analysis.

Definition 1: PLANSAT is the following decision problem: given an instance of the planning problem Π, does there exist a plan Δ that solves Π?

Definition 2: A **conservative** approach to plan modification is one that solves the following **plan modification problem**: given a planning-problem instance Π_I and a plan Δ that solves another instance Π, produce a plan Δ_I that solves Π_I by minimally modifying Δ.

Definition 3: MODSAT is the following decision problem: Given a planning-problem instance Π_I, a plan Δ that solves another instance Π, and an integer k, does there exist a plan Δ_1 that solves Π_1 and contains a sub plan of Δ of at least length k?

We intend to show that TransUCP does not use a conservative plan modification approach, as defined above, to find the solution plan for a given planning problem.

Before proceeding, the following property of TransUCP is brought into focus. 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 23). 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 23 is returned as the solution node by the planner.

The primary claim of this thesis is now proved.

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

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 23). 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.

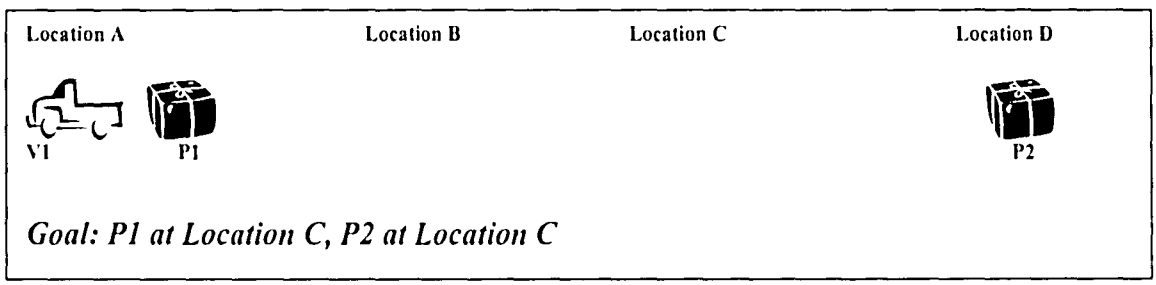

Figure 24: Planning problem to be solved by TransUCP

Consider the planning problem instance shown above in Figure 24 and its solution generated by TransUCP in Figure 25 below. The case plan used by TransUCP is shown in Figure 26. 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 27 solves the planning problem in Figure 24 and is minimally modified from the case plan shown in Figure 26. The highlighted steps show those that have been reused from the original plan.

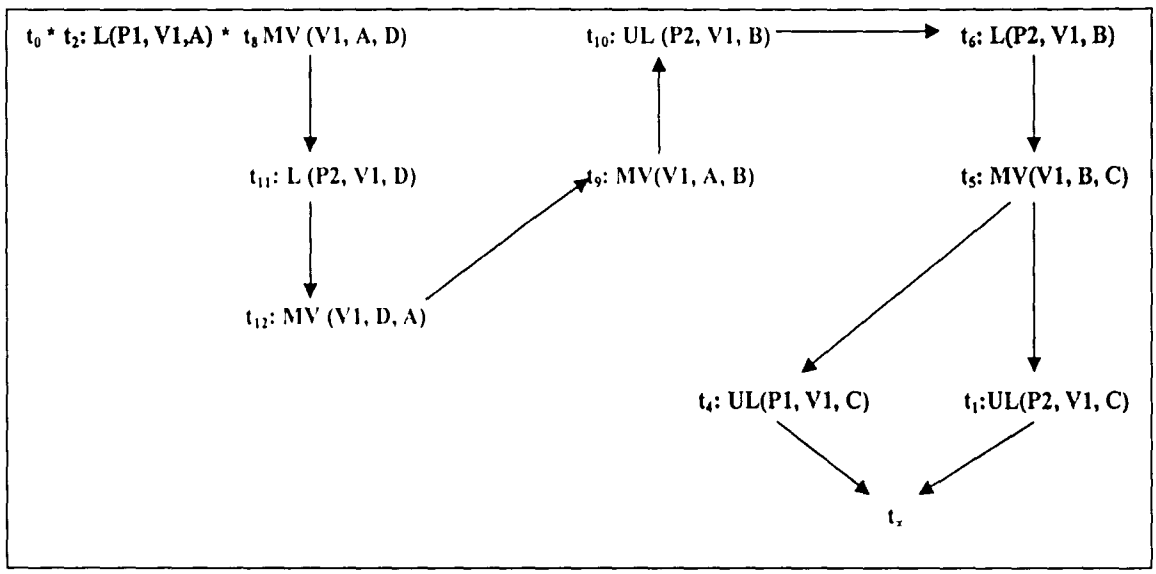

Figure 25: Solution plan generated by TransUCP

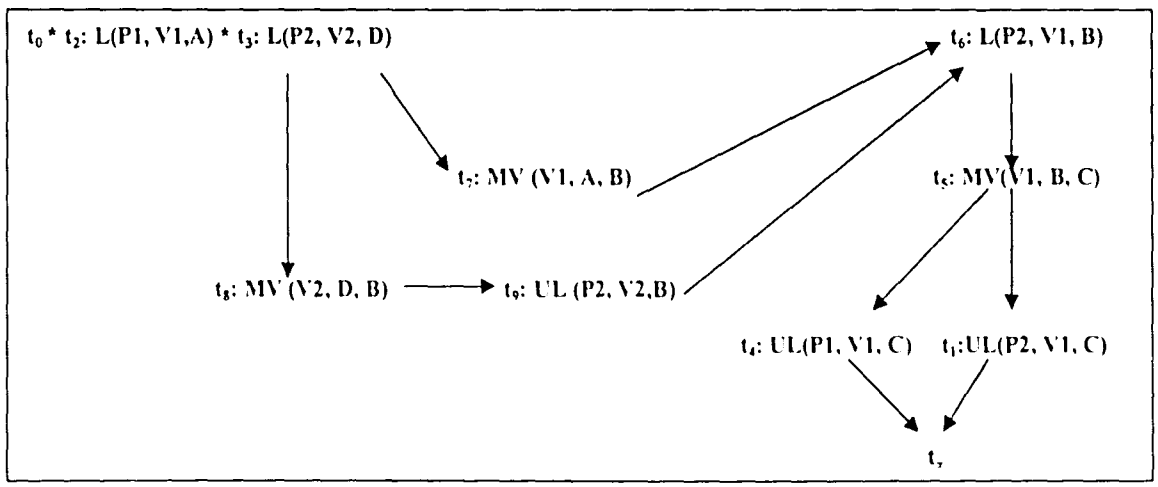

Figure 26: Case plan solution reused by TransUCP

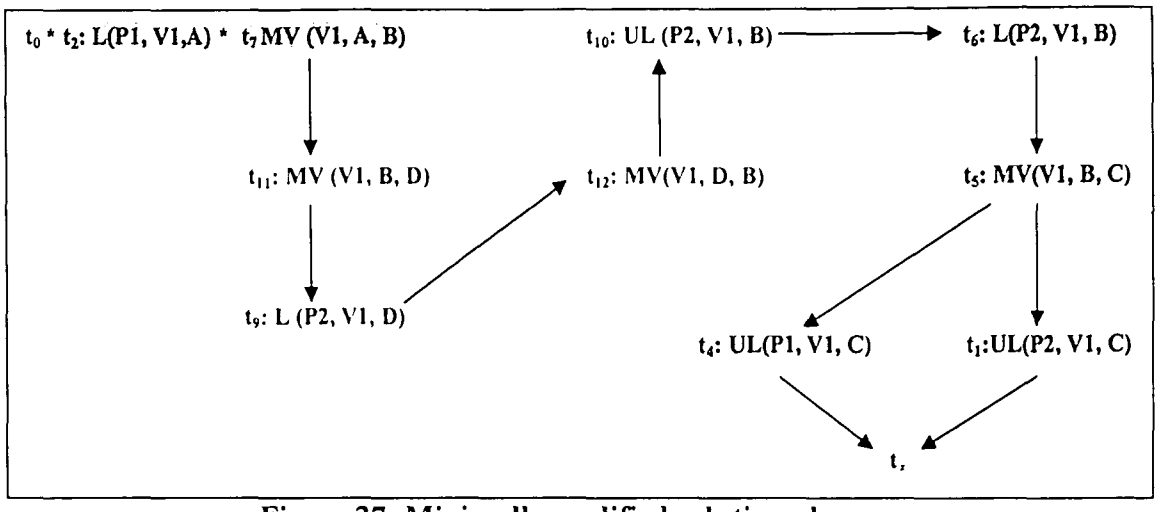

Figure 27: Minimally modified solution plan

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 25), 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 earlier in

this section. 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.

4.4 Completeness

Completeness, in the sense of a planner, can be defined as consisting of the condition that every solution to the planning problem is found. It means that the planner will eventually find a solution plan for the particular planning problem, if there exists one. The completeness of TransUCP is proved with a key assumption that the partial plan search has finite boundaries, i.e. the number of nodes of the graph, N, is finite.

TransUCP searches the plan space looking for the first solution node. More importantly, it makes sure that each node is not visited more than once. It does this by verifying that the new nodes to be visited do not *map* on to the nodes visited previously. This in effect, means that the planner scans the nodes, in both the upward direction (meaning the parents of nodes) and in the downward directions (children of nodes). Therefore, if the planner is left to scan the nodes of the graph, given enough time, it will eventually end up on a solution node, given that here exists one and the graph is finitely bounded. TransUCP stops as soon as it finds the first solution node. It can be modified d so as to not stop

after finding the first solution node and keep scaling the graph till all the solution nodes are found. This proves the completeness of TransUCP under the given assumptions.

4.5 Non-determinism in 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 here, 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.

5 Implementation and Experimental Results

In this section, the implementation details of TransUCP and the results that were drawn from the experimentation are explained.

5.1 Implementation

In this section, the details of the implementation of TransUCP are described and its results are elucidated. The purpose of these experiments was to show that the counter-example shown in Section 4.3 is not an exception and that that TransUCP very rarely behaves like a conservative planner. Therefore it is unlikely that transformational analogy fits in the worst-case complexity scenario of Nebel and Koehler (1995).

The TransUCP algorithm was implemented to generate solutions in the logistics transportation domain. The code was developed and executed in Java. Planning problems in the transportation domain were generated in a random fashion by the code. In the experiments performed, the problems were randomly generated; meaning each of the planning problems contained an arbitrary number of each of the elements in the domain such as trucks and locations. Once the number of trucks and packages were decided randomly, the location of each of these elements was also decided randomly. This was done to ensure that there was no bias of any sort.

Once the planning problem was properly formulated, it was given to the TransUCP planning system to be solved. In all runs of the planner, a single solution plan was

provided as case plan. The case plan to be reused is the one from Figure 26.

The graph being searched is infinitely large and, therefore, the searching process for the solution node had to be "guided" at certain levels, owing to memory and space constraints. Guiding refers to adding appropriate heuristics so that the searching process is carried out in a more effective manner.

After running TransUCP giving as input 10 randomly generated problems and the case plan, non-minimal solution plans were generated in every run.

5.2 Experimental Results

The empirical observations and summary of the experiment performed are explained in this sun-section.

It was noticed that as the number of elements were increased the increase in the plan search space to be searched was very large. Figure 28 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. 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. This further reinforces the claim that it is very unlikely that the solution plan node found by TransUCP for a given solution plan would be a minimally modified.

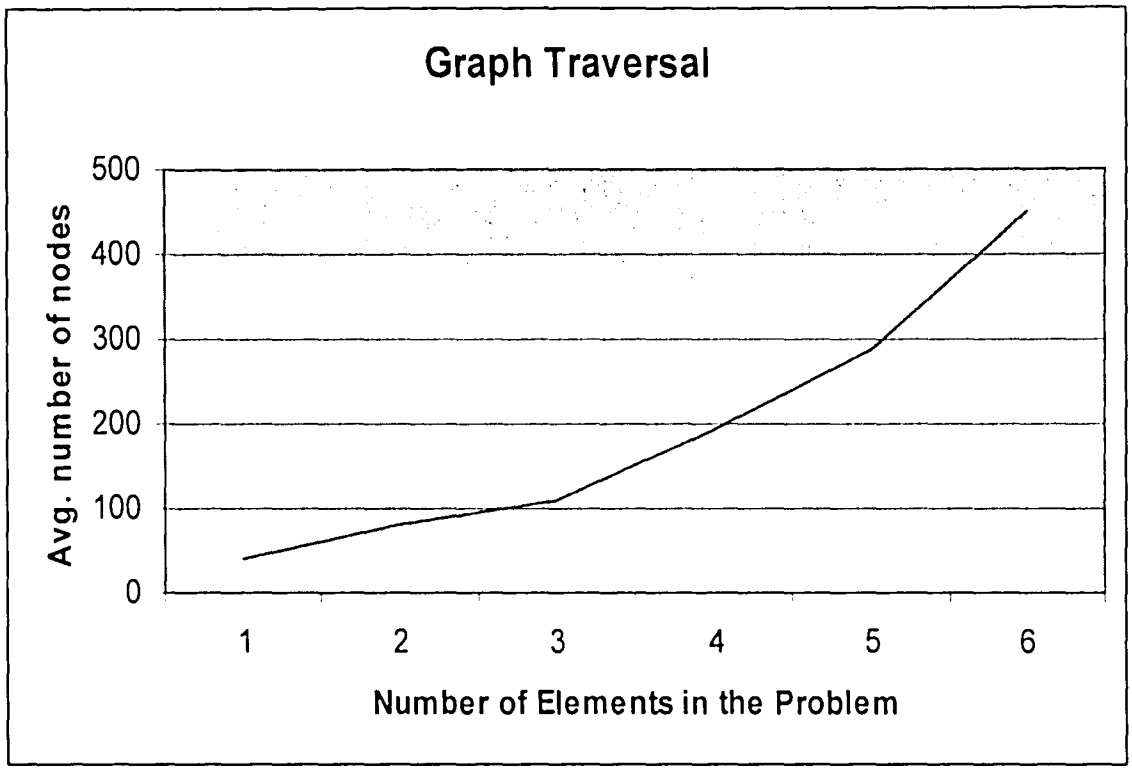

Figure 28: Number of nodes traversed in the graph

6 Conclusion

This thesis has presented TransUCP, a general and domain independent framework for transformational analogy. This framework has been built on top of the universal classical planning framework and extends the SPA system by Hanks and Weld (1996) to transformational analogy.

The framework has been implemented in Java and tested by using it to solve planning problems in the logistics domain. Using this framework, it is demonstrated that transformational analogy does not always perform conservative plan adaptation and generate minimally modified plans. This is proved by carefully constructing an example where conservative plan adaptation does not occur. Through this, it is proved that 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.

6.1 Future Work

A possible extension to the work done would be to develop a domain independent algorithm for the AdjustExactly function (Section 3.2), so that the case-plan solution which is reused can be "adjusted" can be done in a pre-defined manner, irrespective of the domain. Currently this function is dependent on the domain and the framework introduced can profit from making it domain independent.

Another avenue for extending the current work is to develop domain independent criteria to prune the plan space being searched so as to expedite the process of finding a solution node.

Bibliography

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.

Bergmann, R., Muñoz-Avila, H. & Veloso, M. (1996) *General-Purpose Case-Based Planning: Methods and Systems.* AI Communications, 9(3):128–137.

Carbonell, J.G. (1983*) Learning by analogy: formulating and generalizing plans from past experience.* in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.). Machine Learning: An Artificial Intelligence Approach (Morgan Kaufmann. Los Altos, CA.

Carbonell, J.G. (1986) *Derivational analogy: A theory of reconstructive problem solving and expertise acquisition.* In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning*, volume 2, pages 371--392. Morgan Kaufmann.

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.

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. (1996) *Design and implementation of a replay framework based on a partial order planner.* In Weld, D., editor, In: Proceedings of AAAI-96. IOS Press.

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).

Melis, E. (1995) *A Model of Analog-Driven Proof-Plan Construction.* In Proceedings of the 14th International joint Conference on Artificial intelligence, pages 182-189, Montreal.

Nebel, N. and Koehler, J. (1995) *Plan reuse versus plan generation: a theoretical and empirical analysis.* Artificial Intelligence, 76, 427–454.

Veloso, M. (1994) Planning and learning by analogical reasoning. Berlin: Springer-Verlag.

Weld D. (1994) An Introduction to Least Commitment Planning. AI Magazine, 15(4), pages 27-61. AAAI Press.

Vita

Sarat Chandra Vithal Kuchibatla Venkata was born in Karimnagar, India to Kuchibhatla Sundara Nagalakshmi and Kuchibhatla Jagan Mohan Rao in 1981. He attended National Institute of Technology (formerly known as Regional Engineering College), Tiruchirappalli, India from the fall of 1999 through the spring of 2003, receiving a B.E. in Computer Science and Engineering in 2003, and his Masters (pending) from the Department of Computer Science and Engineering in Lehigh University in 2006. He has one publication in the European Conference on Case-Based Reasoning (ECCBR06) conference.

END OF TITLE