Least Common Subsumer Trees for Plan Retrieval

Antonio A. Sánchez-Ruiz\textsuperscript{1} and Santiago Ontañón\textsuperscript{2}

\textsuperscript{1} Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid (Spain)
antsanch@fdi.ucm.es
\textsuperscript{2} Computer Science Department
Drexel University
Philadelphia, PA, USA 19104
santi@cs.drexel.edu

Abstract. This paper presents a new hierarchical case retrieval method called \textit{Least Common Subsumer Trees} (LCS trees). LCS trees perform a hierarchical clustering of the cases in the case base by iteratively computing the least-common subsumer of pairs of cases. We show that LCS trees offer two main advantages: First, they can enhance the accuracy of the CBR system by capturing regularities in the case base that are not captured by the similarity measure. Second, they can reduce retrieval time by filtering the set of cases that need to be considered for retrieval. We present and evaluate LCS trees in the context of plan retrieval for plan recognition, and present procedures for both assessing similarity and computing the least common subsumer of plans using refinement operators.

Keywords: Case-based reasoning, case retrieval, plan similarity

1 Introduction

Similarity assessment, case retrieval and case base organization are some of the most studied topics in Case Based Reasoning (CBR). In this paper we extend our previous work regarding similarity assessment in structured representations \cite{Sanchez-Ruiz11,Sanchez-Ruiz12a,Sanchez-Ruiz12b} and particularly focus on retrieval of plans. We present both a similarity measure for plans using refinement operators and a new hierarchical organization of the case base, \textit{LCS trees}, based on computing the least-common subsumer (LCS) of pairs of plans.

An LCS tree is a tree where the leaves are the cases in the case base, and each node is the LCS of its children. The intuitive idea of LCS trees is that they partition the case base into hierarchical clusters, based on whether cases subsume or not each of the nodes in the tree. Thus, LCS trees offer two main advantages with respect to standard linear case-retrieval: First, they can enhance the accuracy of the CBR system by capturing regularities in the case base that are not captured by a similarity measure; and second, they can reduce retrieval time by
filtering the set of cases that need to be considered for retrieval. Compared to other tree-based case-retrieval approaches, LCS trees thus do not only focus on efficiency, but also on accuracy.

Although in this paper we focus on similarity between plans, LCS trees can be used with any structured representation as long as there exists a mechanism to compute the least-common subsumer of two domain entities. In this sense, the similarity measures presented in our previous work, based around computing the LCS [17, 15, 18], and which we extend in this paper with a similarity measure between plans, can be used to build LCS trees for other representation formalisms, such as description logics or feature terms.

The rest of the paper is organized as follows. Section 2 introduces our plan representation. Section 3 describes a refinement operator for partial plans, which is then used in Section 4 to introduce a similarity measure between plans that also computes their least-common subsumer. Section 5 introduces the concept of LCS tree and explains how to use it to hierarchically organize the case base. In Section 6, we perform a empirical evaluation of both the similarity measure based on refinements and the LCS tree structure. The paper closes with related work, conclusions and directions for future research.

2 Plans and Partial Plans

In the context of this paper, we will consider a plan to be a sequence of actions. For example, consider a maze navigation domain where the only available domain actions are move-forward, turn-right, and turn-left. An example plan with 3 actions is:

$$\langle \text{move-forward, turn-right, move-forward}\rangle$$

Actions are usually formalized using preconditions and effects that describe, respectively, the conditions required to execute them and the changes that they produce. In this paper, however, we will compare plans only from a structural point of view (the actions involved, the order in which they are executed, and their parameters) without considering the preconditions or effects of the actions.

A partial plan is a compact way to describe sets of plans used in partial order planning [10]. A partial plan $$\pi = (A_\pi, <_\pi)$$ consists of a set of actions $$A_\pi = \{a_1, a_2, \ldots, a_n\}$$ and a set of ordering constraints of the form $$a_i < a_j$$, denoting that action $$a_i$$ is executed before action $$a_j$$.

For example, consider the following partial plan consisting of 3 actions:

$$\pi_1 = (\{a_1, a_2, a_3\}, \{a_1 < a_3\})$$

According to this partial plan, $$a_1$$ has to be executed before $$a_3$$, but nothing is said about the relative execution order among the other actions. The usual interpretation of $$\pi_1$$ is that it implicitly describes the set of plans consisting of those 3 actions in which $$a_1$$ appears before $$a_3$$, i.e., $$\langle a_1, a_2, a_3 \rangle$$, $$\langle a_1, a_3, a_2 \rangle$$ and
We say that a partial plan is consistent when the order induced over the actions by $<$ does not contain cycles.

In this work we will extend the usual interpretation of a partial plan $\pi$ to include all the plans consisting at least of the actions in $\pi$ as long as the actions appear in an order compatible with the ordering constraints in $\pi$. For example, using the new interpretation the previous partial plan $\pi_1$ implicitly describes $\langle a_1, a_2, a_3 \rangle$, $\langle a_1, a_3, a_2 \rangle$ and $\langle a_2, a_1, a_3 \rangle$, but also plans like $\langle a_1, a_2, a_3, a_4 \rangle$ or $\langle a_5, a_1, a_3, a_2, a_4 \rangle$. The set of plans implicitly represented by a partial plan is the coverage of the partial plan. Note that, if we do not limit the length of the plans, the coverage of a consistent partial plan is always an infinite set of plans.

**Definition 1.** Given two partial plans $\pi_1$ and $\pi_2$, we say that $\pi_1$ subsumes $\pi_2$, denoted as $\pi_2 \sqsubseteq \pi_1$, and meaning that $\pi_1$ is more general than $\pi_2$ iff $\text{coverage}(\pi_2) \subseteq \text{coverage}(\pi_1)$.

This subsumption relation between partial plans induces a semi-lattice where the most general plan $\top = (\emptyset, \emptyset)$ (a plan with no actions) is the root, and partial plans get more specific as we move away from the root. We have described two ways to make a plan more specific: adding a new action, or adding a new ordering constraint. We call plan refinement to this operation of transforming a plan into one that is more specific.

Until now we have considered actions in plans as atomic entities, but actions are in fact ground instances of planning operators. Planning operators use typed variables to describe sets of related actions. For example, consider a transportation domain that defines the following operator to move between different locations:

$$\text{move}(\text{from} : \text{location}, \text{to} : \text{location})$$

This operator implicitly describes all the actions that result from replacing each of those parameters with a constant compatible with the type location. If we extend the definition of partial plan to allow partially instantiated operators instead of ground actions, we can define additional plan refinements based on specializing the operator parameters. In particular, we can specialize types ($\text{move}(\text{from} : \text{store}, \text{to} : \text{location})$, we can unify variables ($\text{move}(x : \text{location}, x : \text{location})$) and we can replace typed variables with compatible constants ($\text{move}(\text{bakery1, airport1})$).

The ideas of plan refinement, described in detail in the following section, and plan subsumption, form the basis of both the similarity measure (Section 4) and LCS trees (Section 5) presented in this paper.

## 3 Refinement Operators for Partial Plans

This section briefly summarizes the notion of refinement operators and the concepts relevant for this paper (see [12] for a more in-depth analysis of refinement operators). Refinement operators are defined over quasi-ordered sets.
**Definition 2.** A quasi-ordered set is a pair \((S, \leq)\), where \(S\) is a set, and \(\leq\) is a binary relation among elements of \(S\) that is reflexive \((a \leq a)\) and transitive \((if a \leq b \text{ and } b \leq c \text{ then } a \leq c)\).

If \(a \leq b\) and \(b \leq a\), we say that \(a \approx b\), or that they are equivalent. Concerning partial plans, the set of partial plans together with the plan subsumption operation (Definition 1) form a quasi-ordered set. A downward refinement operator is defined as follows:

**Definition 3.** A downward refinement operator over a quasi-ordered set \((S, \leq)\) is a function \(\rho\) such that \(\forall a \in S : \rho(a) \subseteq \{b \in S | b \leq a\}\).

In the context of this paper, a downward refinement operator generates elements of \(S\) which are “more specific” (the complementary notion of upward refinement operator, corresponds to functions that generate elements of \(S\) which are “more general”, but are not used in this paper). A common use of refinement operators is for navigating sets in an orderly way, given a starting element.

- A refinement operator \(\rho\) is **locally finite** if \(\forall a \in S : \rho(a)\) is finite.
- A downward refinement operator \(\rho\) is **complete** if \(\forall a, b \in S | a \leq b : a \in \rho^*(b)\).
- A refinement operator \(\rho\) is **proper** if \(\forall a, b \in S b \in \rho(a) \Rightarrow a \neq b\).

where \(\rho^*\) means the transitive closure of a refinement operator. Intuitively, **locally finiteness** means that the refinement operator is computable, **completeness** means we can generate, by refinement of \(a\), any element of \(S\) related to a given element \(a\) by the order relation \(\leq\) (except maybe those which are equivalent to \(a\)), and **properness** means that a refinement operator does not generate elements which are equivalent to a given element \(a\). When a refinement operator is locally finite, complete and proper, we say that it is **ideal**.

### 3.1 A Downward Refinement Operator for Partial Plans

We will define refinement operators as a set of rewriting rules. A rewriting rule is composed of three parts: the applicability conditions of the rewriting rule (shown between square brackets), the original partial plan (above the line), and the refined partial plan (below the line). Given a partial plan \(\pi = (A_\pi, \prec_\pi)\) the following rewriting rules define a downward refinement operator \(\rho_c\):

(R1) **Add operator:**

\[
\begin{align*}
&[a \text{ is a domain operator with fresh variables}] & \pi & = (A_\pi, \prec_\pi) \\
&\pi' = (A_\pi \cup \{a\}, \prec_\pi)
\end{align*}
\]

*fresh variables* means that \(a\) contains only variables, and none of them appears in \(\pi\).
(R2) Add ordering constraint:

\[
\begin{align*}
& a_i, a_j \in A_{\pi} \land \\
& a_i \not\preceq a_j \land \\
& a_j \not\preceq a_i
\end{align*}
\]

\[
\pi = (A_{\pi}, \prec_{\pi}) \\
\pi' = (A_{\pi}, \prec_{\pi} \cup \{a_i \prec a_j\})
\]

where \( a_i \prec a_j = (a_i \prec a_j) \in \prec_{\pi} \lor (a_i \prec a') \in \prec_{\pi} \land a' \prec a_j \).

(R3) Variable unification:

\[
\begin{align*}
& x_1 : t, x_2 : t \text{ appear in } A_{\pi} \land \\
& \theta = \{x_1 : t \mapsto x_2 : t\}
\end{align*}
\]

\[
\pi = (A_{\pi}, \prec_{\pi}) \\
\pi' = (A_{\pi\theta}, \prec_{\pi\theta})
\]

(R4) Type specialization:

\[
\begin{align*}
& x : t \text{ appears in } A_{\pi} \land \\
& t' \text{ is a direct subtype of } t \land \\
& \theta = \{x : t \mapsto x : t'\}
\end{align*}
\]

\[
\pi = (A_{\pi}, \prec_{\pi}) \\
\pi' = (A_{\pi\theta}, \prec_{\pi\theta})
\]

(R5) Constant introduction:

\[
\begin{align*}
& x : t \text{ appears in } A_{\pi} \land \\
& c \text{ has type } t \land \\
& \theta = \{x : t \mapsto c : t\}
\end{align*}
\]

\[
\pi = (A_{\pi}, \prec_{\pi}) \\
\pi' = (A_{\pi\theta}, \prec_{\pi\theta})
\]

The above rules correspond to the 5 different ways to specialize a partial plan that we introduced in Section 2: adding a new planning operator, adding a new ordering constraint, unifying variables, replacing the type of a variable with a more specific type, and replacing a variable with constants of the same type (rules R3-5 use a substitution \( \theta \) to specialize the parameters). \( \rho_c \) is locally finite and complete for the set of consistent partial plans (two partial plans are equivalent if one can be obtained by just renaming variables in the other).

Although \( \rho_c \) is not ideal, in practice it behaves quite well because the number of times it can be used without specializing a partial plan is limited (and usually quite small). Note that rules R1 and R2 always specialize the plan, and rules R3-5 do not produce a specialization only in very particular scenarios (when a type has only a subtype in the domain model or when there is only one constant compatible with a type).

4 Refinement-based Similarity between Plans

Given the refinement operator \( \rho_c \), we can measure similarity between plans following the recent idea of refinement-based similarity measures [15, 18, 17], illustrated in Figure 1, and based on the following intuitions:
First, given two partial plans $\pi$ and $\pi'$ such that $\pi' \subseteq \pi$, it is possible to reach $\pi'$ from $\pi$ by applying a complete downward refinement operator $\rho$ to $\pi$ a finite number of times, i.e. $\pi' \in \rho^*(\pi)$.

Second, the number of times a refinement operator needs to be applied to reach $\pi'$ from $\pi$ is an indication of how much more specific $\pi'$ is than $\pi$. The length of the chain of refinements to reach $\pi'$ from $\pi$, which will be noted as $\lambda(\pi \xrightarrow{\rho} \pi')$, is an indicator of how much information $\pi'$ contains that was not contained in $\pi$. It is also an indication of their similarity: the smaller the length, the higher their similarity.

Third, given any two partial plans, their least common subsumer (LCS) is the most specific partial plan which subsumes both. The larger the LCS (more actions and more ordering constraints), the more the two partial plans share.

Given two partial plans $\pi_1$ and $\pi_2$, the LCS can be computed by starting with the most general plan $\top$ and refining it with the refinement operator until no more refinements can be applied that result in a partial plan that still subsumes $\pi_1$ and $\pi_2$.

The LCS of two partial plans contains all that is shared between them, and the more they share the more similar they are. $\lambda(\top \xrightarrow{\rho} LCS(\pi_1, \pi_2))$ measures the distance from the most general partial plan, $\top$, to the LCS and, thus, it is a measure of the amount of information shared by $\pi_1$ and $\pi_2$. Similarly, $\lambda(LCS(\pi_1, \pi_2) \xrightarrow{\rho} \pi_1)$ and $\lambda(LCS(\pi_1, \pi_2) \xrightarrow{\rho} \pi_2)$ measure the amount of information specific to each partial plan and that is not shared.

Using these ideas, the similarity between two partial plans $\pi_1$ and $\pi_2$ can be measured as the ratio between the amount of information contained in their LCS and the total amount of information contained in $\pi_1$ and $\pi_2$. These ideas are collected in the following formula:

$$S_\rho(\pi_1, \pi_2) = \frac{\lambda_1}{\lambda_1 + \lambda_2 + \lambda_3}$$
the previous similarity measure. This transformation is quite straightforward
from them into the most specific partial plans that cover them and then use
thus the similarity is 0. On the other hand, if two plans share nothing, then
\( \lambda \) is 1. In order to compute the similarity between (non-partial) plans, we first trans-
form them into the most specific partial plans that cover them and then use
the previous similarity measure. This transformation is quite straightforward
since the most specific partial plan that covers a plan \( p = \langle a_1, \ldots, a_k \rangle \), is
\( \pi = \langle \{ a_1, \ldots, a_k \}, \{ a_1 < a_2, \ldots, a_{k-1} < a_k \} \rangle \), i.e., a partial plan with the
same actions that defines ordering constraints between consecutive actions.

Figure 2 shows a detailed example of the similarity assessment between 2
plans using the procedure described in this section.

\[
\begin{align*}
\text{Constants} & \quad p_1 : \text{location}, p_2 : \text{location}, b_1 : \text{box}, b_2 : \text{box} \\
\text{Plans to partial plans} & \\
\langle \text{take}(b_1), \text{move}(p_1, p_2) \rangle & \rightarrow \pi_1 : \langle \{ a_1 : \text{take}(b_1), a_2 : \text{move}(p_1, p_2) \}, \{ a_1 < a_2 \} \rangle \\
\langle \text{move}(p_1, p_2), \text{drop}(b_2) \rangle & \rightarrow \pi_2 : \langle \{ b_1 : \text{move}(p_1, p_2), b_2 : \text{take}(b_2) \}, \{ b_1 < b_2 \} \rangle \\
\text{Refinements from} \quad \top \quad \text{to the LCS} & \\
0 : (\{\}, \{\}) & \\
1 : (\{c_1 : \text{move}(\langle x_1 : \text{location}, x_2 : \text{location} \rangle), \{\}\}) & \\
2 : (\{c_1 : \text{move}(\langle p_1, x_2 : \text{location} \rangle), \{\}\}) & \\
3 : (\{c_1 : \text{move}(\langle p_1, p_2 \rangle), \{\}\}) & \\
\text{Refinements from} \quad \text{the LCS} \quad \text{to} \quad \pi_1 & \\
1 : (\{c_1 : \text{move}(\langle p_1, p_2 \rangle, c_2 : \text{take}(\langle x_3 : \text{object} \rangle), \{\}\}) & \\
2 : (\{c_1 : \text{move}(\langle p_1, p_2 \rangle, c_2 : \text{take}(\langle x_3 : \text{object} \rangle), \{ c_2 < c_1 \} \}) & \\
3 : (\{c_1 : \text{move}(\langle p_1, p_2 \rangle, c_2 : \text{take}(\langle x_3 : \text{box} \rangle), \{ c_2 < c_1 \} \}) & \\
4 : (\{c_1 : \text{move}(\langle p_1, p_2 \rangle, c_2 : \text{take}(\langle b_1 \rangle), \{ c_2 < c_1 \} \}) & \\
\text{Refinements from} \quad \text{the LCS} \quad \text{to} \quad \pi_2 & \\
1 : (\{c_1 : \text{move}(\langle p_1, p_2 \rangle, c_2 : \text{drop}(\langle x_4 : \text{object} \rangle), \{\}\}) & \\
2 : (\{c_1 : \text{move}(\langle p_1, p_2 \rangle, c_2 : \text{take}(\langle x_4 : \text{object} \rangle), \{ c_2 < c_1 \} \}) & \\
3 : (\{c_1 : \text{move}(\langle p_1, p_2 \rangle, c_2 : \text{take}(\langle x_4 : \text{box} \rangle), \{ c_1 < c_2 \} \}) & \\
4 : (\{c_1 : \text{move}(\langle p_1, p_2 \rangle, c_2 : \text{take}(\langle b_2 \rangle), \{ c_1 < c_2 \} \}) & \\
\text{Similarity} \quad S_p(\pi_1, \pi_2) = \frac{3}{3+4+4} = 0.27
\]

\[
\lambda_1 = \lambda(\top) \xrightarrow{\ell} LCS(\pi_1, \pi_2) \\
\lambda_2 = \lambda(LCS(\pi_1, \pi_2)) \xrightarrow{\ell} \pi_1 \\
\lambda_3 = \lambda(LCS(\pi_1, \pi_2)) \xrightarrow{\ell} \pi_2
\]

Intuitively, if two plans are identical, then \( \lambda_2 \) and \( \lambda_3 \) are 0, and thus the
similarity is 1. On the other hand, if two plans share nothing, then \( \lambda_1 \) is 0, and
thus the similarity is 0.

Fig. 2. Similarity assessment example (refinements are underlined).
5 Least Common Subsumer Trees

The retrieval task in a CBR system involves finding the most relevant cases in the case base to solve a given query. In our system the case base is a repository of plans annotated with the type of problem they try to solve. Given a plan query, a standard way to perform retrieval is to find the nearest neighbors using a similarity measure. The standard approach to find the nearest neighbor is to perform linear search, but efficient approaches such as \( kd \)-trees [21] or cover trees [1] exist that can find the nearest neighbor in near logarithmic time.

In this paper, we present a new hierarchical structure that we call an LCS tree (Least Common Subsumer tree) to perform retrieval. The LCS tree exploits the fact that our refinement-based similarity measures not only provides a numerical value of similarity but an explicit description of the structure shared between two plans (the LCS). In this section, we describe how to use this additional information to hierarchically organize the plan base. Thanks to this additional information, our hierarchical structure not only helps in decreasing retrieval time, but also exploits the LCS information to increase performance by retrieving better cases. An LCS tree is defined in the following way: each node in the tree is a partial plan; each node that is not a leaf is the LCS of its children. Thus, a given node is always subsumed by all of its ancestors, and a given node always subsumes all of its descendants.

Algorithm 1 Construction of the LCS tree.

```
function BuildLCSTree(PlanBase)
    T = ∅
    for all Plan ∈ PlanBase do
        T = T ∪ { Tree(PartialPlan(Plan)) }
    end for
    while |T| ≠ 1 do
        MSP = PairsOfMostSimilarTrees(T)
        for all (t₁, t₂) ∈ MSP do
            T = T \ {t₁, t₂} ∪ {Tree(t₁, LCS(t₁, t₂), t₂)}
        end for
    end while
    return only tree in T
end function
```

Algorithm 1 shows how to build a LCS tree from a collection of plans. The algorithm builds the tree from the leaves to the root and consists of two different stages. The first stage (lines 3-5) transforms each plan from the input into a partial plan using the approach described in Section 4, and then generates a set \( T \) with one tree leaf containing each of the partial plans. The second part of the algorithm (lines 6-11) matches the trees than contain the most similar partial plans in pairs (according to our similarity measure) and adds them as children of a new tree whose root contains the LCS of both plans. That way, each pass
through the loop generates a new level of the tree with approximately half of the previous level nodes. At the end of the process we obtain a binary tree in which each node contains a partial plan which subsumes (in more general than) all the partial plans in the nodes below. And the leaves of the tree contain the partial plans corresponding to the original plans in the plan base.

Figure 3 shows an example LCS tree generated with Algorithm 1 from a set of three plans. As can be seen, in the LCS tree, each node subsumes all of its descendants, and, in particular, the root subsumes all the plans in the case base. An interesting aspect of LCS trees is that if there is a set of features that a large number of plans in the case base share, then those would most likely be captured by some node in the tree as part of the LCS. This is interesting, since if there is some feature that is shared amongst a large set of plans, probably means that that feature is important. Thus, in a sense, the LCS tree can be likened to a hierarchical clustering technique [9] that computes a description (the LCS) of each cluster, or to a subset of the concept lattice that would be built by Formal Concept Analysis [7]. Given a new problem, we can determine whether the new problem belongs to one of these clusters or not via subsumption by the appropriate LCS. This idea defines our retrieval algorithm.

The retrieval is shown in Algorithm 2. There are two intuitions behind this algorithm. First, similar plans will be stored in nearby leaves of the tree because the more similar the plans are the closer they will be to their LCS node. Second, the differences between the LCSs that are in the higher levels of the tree are likely to be differences that are of key importance, since they are the ones that distinguish larger clusters of plans. Basically, given a plan query the algorithm looks for similar plans in the subtrees containing a partial plan which subsumes (more general than) the query. We need to make several considerations in order to explain the operation of the algorithm:
Algorithm 2 Retrieve partial plan candidates.

1: function RETRIEVE(LCSTree, Query, MaxDepth)
2:     if IsLeaf(LCSTree) or Depth(LCSTree) ≥ MaxDepth then
3:         C = PARTIALPLANSINLEAVES(LCSTree)
4:     else
5:         C = ∅
6:         for all SubTree of LCSTree do
7:             if IsLeaf(SubTree) or PARTIALPLAN(SubTree) subsumes Query then
8:                 C = C ∪ RETRIEVE(SubTree, Query, MaxDepth)
9:         end if
10:     end for
11:     if C = ∅ then
12:         C = C ∪ PARTIALPLANSINLEAVES(LCSTree)
13:     end if
14: end function
15: return C

– The sets of plans that are subsumed by two children of a given node do not necessarily have to be disjoint. For example, in Figure 3, a plan that has a move action, then a drop, and then a take would be subsumed by both children of the root node. Therefore, during retrieval we might have to visit both of them. In practice, the tree is useful to filter an important number of plans (the leaves of the tree), but in the worst scenario the search remains linear (line 6).

– The algorithm does not only return a plan but a sequence of candidates. During the search process it is possible to reach a node such that none of its children subsume the query. In this situation the retrieval algorithm returns all the plans in the leaves of the current tree (lines 11-13) as candidates. The list of candidates is processed afterwards linearly to find the most similar plan among them to the query.

– Although both the similarity measure and the tree construction are based on the idea of subsumption, there is no guarantee that the resulting list of candidates will contain the nearest neighbor to the query. The list of candidates will contain, however, some of the most similar plans in the tree and, as we discuss in Section 6, it outperforms nearest neighbor retrieval in our experiments.

– The parameter MaxDepth limits the maximum depth of the search and it is useful to configure the behavior of the algorithm (line 2). Smaller values of MaxDepth produce larger sets of candidates (the probability of missing the most similar case decreases) and increases the retrieval time, while larger values of MaxDepth produce smaller sets of candidates and improve the retrieval times. So we can use this parameter to balance the desired level of accuracy and performance.
In order to evaluate the approach presented in this paper, we used two datasets, commonly used in the plan recognition literature: Linux and Monroe. The Linux dataset [4] consists of 457 plans, classified into 18 different classes, where each class represents one of 18 different goals that a Linux user can be trying to achieve (create a file, compress a folder, etc.). Each plan contains between 1 to 60 actions (with an average of 6.1), and the actions are Linux command-line commands including their parameters. The Monroe dataset [2] contains plans for a disaster management domain. It consists of 5000 plans classified into 10 different classes, and each plan contains between 1 to 29 actions (with an average of 9.6). Plans in the Linux domain correspond to recorded traces of human users (real data) while plans in the Monroe domain were generated using a modified version of the SHOP2 planner [14] (synthetic data).

We performed two sets of experiments. In the first one, we evaluated the performance of LCS trees in the context of a nearest neighbor classifier, when trying to predict the correct class for each of the plans in each of the datasets. In a second experiment, we evaluated the performance of our LCS trees in determining the class of a plan when only part of the plan can be observed (i.e., when we can only observe the first few actions of the plan). This second experiment is relevant for many tasks, such as intrusion detection or plan recognition (where we want to identify a plan before it is actually completed). All of our experiments were performed using a leave-one-out method, where to classify one plan, we compared it against all the other plans in our training set.

In both experiments we compared LCS trees using our refinement-based similarity measure $S_\rho$, against linear standard retrieval using both $S_\rho$ and a collection of other similarity measures:

- **Random**: just returns a random number between 0 and 1.
- **Jaccard**: given two plans, their similarity is determined using the Jaccard index, i.e., the number of common actions divided by the total number of different actions in both plans.
- **Edit-distance (action-level)**: assuming that a plan is a sequence of symbols, and where each action is a different symbol, we compute the distance between two plans $p$ and $q$ as the edit distance [13] between these sequences.
- **Edit-distance (symbol-level)**: this similarity is the same as the previous one, but where consider that each action name, and each action parameter is a different symbol. So, this distance is more fine-grained.
- **Edit-distance (character-level)**: finally, going one step further, we convert every plan to a string of characters, used to compute the edit distance (at a character level). The rationale behind this distance is that similar parameters or actions tend to have similar string representations, and thus, without having to add additional domain knowledge to our planning domains about the relation between their actions or parameter types, we can compute similarity at an even finer granularity level.
Table 1 shows the performance in both domains. The Linux domain is well known to be a very hard domain (the original paper presenting the Linux domain reports precision and recall values of 0.351 and 0.236 respectively, when detecting the plan class) because the dataset contains real data. Nevertheless our refinement-based similarity measure $S_\rho$ outperforms all the other similarity measures significantly. Additionally, when we compare the linear and the hierarchical plan base using LCS trees, we realize that the LCS trees not only substantially improve the retrieval time (reduced to a 29.5% of the original time with $MaxDepth = 5$), but also improve the accuracy. The explanation is that, in this domain, the filtering performed by the LCS tree, equivalent to hierarchical clustering, captures additional information that the similarity measure does not.

The Monroe domain is an easier domain than the Linux one, and the literature on plan recognition reports prediction accuracies between 95% to 99% for different methods (for example, [3] report a precision of 95.6% in this dataset). Again, we observe that our refinement-based similarity outperforms all other similarity measures, reaching an accuracy of 99.8%. Edit distance at the character level is the similarity measure that came closer, reaching an accuracy of 97.8%. Another interesting result is that using LCS trees and $MaxDepth = 5$ we are able to reduce the retrieval time to 35.7% of the original time and we only loose a few tenths of accuracy.

Figure 4 shows the performance of the different best similarity measures in our second set of experiments. For different plan lengths $k$ starting from 1 and up to the length of the longest plan in each dataset, we classify each plan but only using the first $k$ actions of the plan. The intuition is that when only using the first few actions, classifying the plan should be harder, and the more actions being considered, the higher the expected performance. The idea is to model the situation (common in the plan recognition literature) of trying to recognize the first actions of a plan $p$ against a library of complete plans. As expected, the classification accuracy improves quite fast as we consider more actions at the beginning and stabilizes around 10-13 actions for both $S_\rho$ and the edition...
(char) distance. In this experiments using an LCS tree not only improves the performance but, in most of the lengths, classifies better the plans than the linear plan base, illustrating again, that the LCS-based clustering used in LCS trees can capture aspects of the problem domain that the similarity measure does not.

In summary, our results point out that the refinement-based approach to plan similarity achieves very good accuracy, higher than all the other similarity measures we used in our evaluation. Additionally, this accuracy can be improved in some domains by the structure imposed by LCS trees, achieving a side effect of significantly improving retrieval times.

7 Related work

Our LCS trees resemble other approaches such as *kd*-trees [21] or cover trees [1]. However, LCS trees are not just an organization to improve the retrieval time, they serve as an additional filter to retrieve better cases. In this sense, LCS trees can be related to hierarchical clustering [9].

Concerning our similarity between plans, similarity measures for plans have been studied mostly in the field of case-based planning [5]. Most of these similarities, like the *foot-print* similarity used in the PRODIGY system [20] or the Action-Distance Guided similarity [19] are based on comparing the initial and end states when executing plans. Our similarity measure focuses on the plan
structure instead, and can be used for tasks like plan recognition in which the
goal of one of the plans is unknown.

The refinement operator presented in this paper resembles the search process
carried out by partial-order planners like UCPOP [16]. However, the space of
partial plans is traversed with a different perspective since we are not solving a
planning problem and we do not need to keep casual links between actions or
an agenda of open goals.

Our work is related to the general idea of assessing similarity between struc-
tured representations. In our previous work we presented similarity measures
for instances represented using Description Logics [17] and Feature Terms [15].
Other relevant work on structured similarity measures is that of RIBL (Rela-
tional Instance-Based Learning), an approach to apply lazy learning techniques
while using Horn clauses [6]. For a comprehensive overview on this topic, the
reader is referred to [15].

Plan similarity is also related to the area of plan recognition [11] [8], where
the goal is to identify plans, or their goals or intents. One of the main differences
with respect to our work is that plan recognition concerns with the problem of
identifying plans from an already existing plan model (e.g. a plan grammar, or a
given set of goals), whereas in the work presented in this paper we are interesting
on measuring similarity between two concrete plans.

8 Conclusions

In this paper we have introduced the Least Common Subsumer Trees (LCS
trees), a new hierarchical case retrieval method that clusters the cases according
to their least-common subsumers. LCS trees are able to capture implicit patterns
in the case base and can be used in conjunction with another similarity measures
to improve the accuracy of a CBR system. LCS trees also reduce the retrieval
time by filtering the set of cases that need to be considered for retrieval.

Since LCS trees are based on subsumption, they can be used in domains
formalized using structured representations. We have empirically shown their
advantages in the concept of plan recognition, using a nearest neighbor classifier
and a repository of annotated plans. We have also described a similarity measure
for partial plans based on refinements that can be used to compute the LCS and
to build the tree. The combination of the LCS tree to retrieve a set of candidate
plans and our similarity measure based on refinements to select the most similar
among them, achieves very good accuracy results in our experiments.

As part of our future work, we would like to study the types of patterns that
LCS trees are able to capture, as well as their relation with other hierarchical
clustering techniques, to better understand the interplay between the unsupervised
induction performed by the LCS tree and similarity-based case-retrieval.

References