Appl Intell
DOI 10.1007/s10489-012-0404-6

A case-based approach to heuristic planning

Tomás de la Rosa · Angel García-Olaya · Daniel Borrajo

Abstract Most of the great success of heuristic search as an approach to AI Planning is due to the right design of domain-independent heuristics. Although many heuristic planners perform reasonably well, the computational cost of computing the heuristic function in every search node is very high, causing the planner to scale poorly when increasing the size of the planning tasks. For tackling this problem, planners can incorporate additional domain-dependent heuristics in order to improve their performance. Learning-based planners try to automatically acquire these domain-dependent heuristics using previous solved problems. In this work, we present a case-based reasoning approach that learns abstracted state transitions that serve as domain control knowledge for improving the planning process. The recommendations from the retrieved cases are used as guidance for pruning or ordering nodes in different heuristic search algorithms applied to planning tasks. We show that the CBR guidance is appropriate for a considerable number of planning benchmarks.

Keywords Case-based reasoning · Automated planning · Search algorithms

T. de la Rosa (✉) · A. García-Olaya · D. Borrajo
Departamento de Informática, Universidad Carlos III de Madrid, Av. Universidad 30, Leganés, Madrid, Spain
e-mail: trosa@inf.uc3m.es

A. García-Olaya
e-mail: agolaya@inf.uc3m.es

D. Borrajo
e-mail: dborrajo@ia.uc3m.es

Published online: 10 January 2013

1 Introduction

Automated Planning is an area of the Artificial Intelligence focused on finding solutions to problems in a domain-independent way. A planning problem is a computational task whose inputs are a set of actions that can be applied in the environment, an initial state and a set of goals to be reached. The aim of Automated Planning is to find a plan that transforms the initial state of the environment into a different state where the goals are met. Automated Planning systems have been successfully applied to real world problems such as planning space missions [21], management of fire extinctions [4], control of underwater vehicles [20] or web service compositions [18]. Heuristic search in the state space is one of the most successful approaches to solve automated planning problems. Heuristics in planning are computed in a domain-independent way, using information extracted from the domain definition, the state to be evaluated, and the goal set. These heuristics usually estimate the distance between the current state and a state where goals are achieved. The estimate is used as guidance for selecting the next node in a search algorithm. Many heuristic planners integrate this basic idea. They derive heuristics from the relaxation of action negative effects such as [15] and [3]; or from causal graphs, such as [13]. Some other researchers have developed techniques for complementing or modifying these planners, such as [5, 22, 26].

Although heuristic planners perform reasonably well in many benchmark domains, they suffer from scalability problems due to the computational cost of computing the heuristic. Indeed, these planners spend most of the planning time computing the heuristic function, since they have to obtain an estimate for every node in the search process. Usually, a single heuristic estimation can be computed in polynomial time. However, the total planning time grows exponentially

because the number of expanded states grows exponentially too and the search algorithm needs to evaluate all of them, including non-interesting ones (i.e., nodes that do not appear in any solution plan). The problem of useless node evaluations are exacerbated in domains where the heuristic function is misleading or has poor quality. To tackle this problem different learning techniques have been used to improve the performance of planners. In fact, the community has a renewed interest for applying machine learning techniques to state-of-the-art planners. An example of this interest is the Learning Track organized in the 2008 International Planning Competition (IPC). The system CABALA [8], one of the competitors, was based on the ideas presented next.

In this article we present a learning technique that acquires domain-dependent knowledge from previously solved plans, with the aim of helping planning algorithms to avoid useless node evaluations. We have developed this technique using a Case-Based Reasoning (CBR) approach. We store the experience provided by solving previous easy problems, and then, when a new problem is being solved, we retrieve the most similar situation in order to recommend most promising nodes for the search algorithm to explore. These recommendations allow search algorithms to skip some useless node evaluations and therefore speed up the search. We also present the evaluation of this approach in eight planning benchmarks. The results show that the technique is useful for improving heuristic planners, especially when they are using greedy search algorithms.

In the following sections we describe the approach, organized as follows. The next section explains the planning model. Then, we formalize the representation used for domain-dependent knowledge. Afterward, we explain the cycle of reasoning from a CBR perspective [1]. In that section, we explain how the learned knowledge is used in different heuristic algorithms. Then, we present an experimental evaluation of the approach in eight planning benchmarks. Finally, we discuss related work and present our conclusions.

2 Planning model

The base planning model of this approach is the STRIPS formalism. In a domain $\mathcal{D}$, the state space $\mathcal{S}$ for a problem, and the set of actions $\mathcal{A}$ are defined in terms of predicate symbols $\mathcal{Y}$, operators (action schemas) $\mathcal{O}$ and the set of object constants $\mathcal{C}$ given for a specific task. A state $s \in \mathcal{S}$ is the set of facts (state literals) that are true in a given instant. Each fact in s is an instantiation of a predicate symbol $y \in \mathcal{Y}$ in the form $y(c_1, \ldots, c_n)$ where $c_i \in \mathcal{C}$. An operator $op \in \mathcal{O}$, that has m parameters, can instantiate into action $a \in \mathcal{A}$ in the form $op(c_1, \ldots, c_m)$ with $c_i \in \mathcal{C}$. Additionally, each action $a \in \mathcal{A}$ is defined in terms of three sets of facts $(pre(a), add(a), del(a))$, where $pre(a)$ are the preconditions that should be true for the action to become applicable, $add(a)$ are the facts that become true when the action is executed, and $del(a)$ are the facts that are no longer true after the action execution. If an action a is applied in a state s, denoted by $apply(a, s)$, the resulting state will be $s' = (s - del(a)) \cup add(a)$.

A planning task in $\mathcal{D}$ is defined as the tuple $(\mathcal{C}, I, G)$, where $\mathcal{C}$ is the set of constants for instantiating $\mathcal{S}$ and $\mathcal{A}$, $I \in \mathcal{S}$ is the initial state and G is the set of facts (a partially defined state) that represent the goals. Solving a planning task implies finding a plan $\pi = \{a_1, \ldots, a_n\}$ such that, if applied consecutively starting from the initial state, it leads to a state where all the goals are true, i.e., $\mathcal{G} \subseteq apply(a_n, apply(a_{n-1}, \ldots, apply(a_1, \mathcal{I})))$.

Moreover, we define additional functions to handle the case base representation used later, namely:

- $pred(l) = y$ is the predicate symbol of the fact l
- $arg(l) = (c_1, \ldots, c_n)$ are the arguments of the fact l
- $opt(a) = op$ is the operator of the action a
- $argop(a) = (c_1, \ldots, c_m)$ are the parameters of the action a

Some parts of our model depend on features from the relaxed plan heuristic [15]. In general, heuristics based on delete relaxation compute their estimation by considering a relaxed version of the planning task, which is the same planning task, but using a variation of the action set $\mathcal{A}$ (i.e., it does not contain the $del(a)$ set). In particular, the aim of the relaxed plan heuristic is finding a relaxed plan π^+ that solves the relaxed planning task. The relaxed plan is extracted from a *Relaxed Planning Graph* (RPG). An RPG is a sequence of proposition and action layers $P_0, A_0, P_1, A_1, \ldots, P_{n-1}, A_n, P_n$. The first proposition layer, P_0, contains the propositions in the state to be evaluated, s. Then, each action layer contains all applicable actions given the previous proposition layer. The positive effects of all actions in a layer are included in the next proposition layer. Thus, each proposition layer contains the set of achieved propositions so far. Hoffmann and Nebel [15] showed that a relaxed plan can be extracted from an RPG in polynomial time. The relaxed plan extraction algorithm selects in each proposition layer the facts marked as goals and then selects actions from the previous layer that achieve the goals, including as new goals the preconditions of the selected actions. This algorithm corresponds to a backtracking-free version of the one used in the GRAPHPLAN planner [2]. Hoffmann and Nebel [15] also introduced the pruning strategy of the *helpful actions*. A helpful action is an applicable action (i.e., it belongs to the first action layer in RPG) that achieves a fact marked as a goal (or sub-goal) during relaxed plan extraction. For an extended description of heuristic planning refer to [12].

3 Definitions

A *typed sequence* is the basic piece of knowledge used by our CBR algorithm. We present a set of definitions first.

Definition 1 Given a state s, an **object sub-state** is the set of state literals in which the object appears as an argument. The sub-state for the object o in the state s is denoted by

$$s_o = \{l \mid l \in s \ \wedge \ o \in arg(l)\}$$

For instance, suppose that in the *Blocksworld* domain we have a state $s = \{(on\ A\ B)\ (ontable\ B)\ (clear\ A)\ (holding\ C)\}$. We say that the object sub-state for block A is $\{(on\ A\ B)\ (clear\ A)\}$.

Definition 2 A **property** of an object o in a literal l is the position of o in l. It will be represented as a predicate symbol subscripted by a number representing the argument position in the state literal.

$$U_o(l) = pred(l)_i \mid arg(l) = (c_1, \ldots, c_n) \wedge o = c_i \quad (1)$$

For instance, on_1 is the property defined for the block A in the literal $(on\ A\ B)$, and on_2 is the property for the block B. Therefore, a predicate of arity n produces n properties. This concept was originally introduced by [11], where properties are used to derive domain types.

Since properties represent object positions in state literals and each position has a particular domain type, we can use properties for referring to objects in an abstract way. Thus, we can transform an object sub-state into what we will call a typed sub-state, using the properties for the literals that belong to an object sub-state. Since two or more literals can produce the same property for an object, we use the concept of *collection* as a generalization of the concept of sets. A collection is an unordered group of elements where a particular element can appear more than once.

Definition 3 A **typed sub-state** is the collection of properties that abstracts an object sub-state. We will represent the typed sub-state for an object o in the state s as:

$$\varphi_{s,o} = \{U_o(l) \mid l \in s_o\} \quad (2)$$

In the previous example, the object sub-state for block A is transformed into the typed sub-state $(on_1\ clear_1)$. If we apply the action `(STACK C A)` to the state in the example, the new object sub-state for block A is $[(on\ A\ B)\ (on\ C\ A)]$. Therefore, the new typed sub-state becomes $(on_1\ on_2)$.

Since objects of a particular planning instance are not relevant when recognizing generalized domain transitions, we need to represent typed sub-states that abstract any valid object sub-state in the domain.

Definition 4 An **arbitrary typed sub-state** $\mathcal{T}$ is a property collection that abstracts all possible object sub-states s_o that produce equal $\varphi_{s,o}$, for an arbitrary object o.

For instance, the arbitrary typed sub-state $\mathcal{T} = (on_1\ clear_1)$ represents all states where any block is on another block and clear. The typed sub-state $\varphi_{s,o}$ is related to an object o, but T is related to all objects having the same typed sub-state.

In a domain, we can identify common transitions of a given object and associate them with the applied action, assuming that the object is a parameter of the action. Thus,

Definition 5 A **typed sub-state transition** is a structure of the form

$$\langle \mathcal{T}, op, \mathcal{T}' \rangle$$

representing an observed transition generated when a typed sub-state $w_{o,s}$ is transformed into $w_{o,s'}$ after applying an action a such that $op = opt(a)$. The action a is generalized to its operator op since the other parameters are not relevant for the object perspective of the representation.

Definition 6 A **typed sequence** $\mathcal{Q}$ is a sequence of ordered pairs of the form

$$\mathcal{Q} = \{(\mathcal{T}_0, \emptyset), (\mathcal{T}_1, op_1), \ldots, (\mathcal{T}_n, op_n)\}$$

that represents an object-centered abstraction of a plan, in which each pair is an arbitrary typed sub-state with the operator used to reach that typed sub-state.

A typed sequence is a way of summarizing the typed sub-state transitions. For an object o and a plan $\pi = \{a_1, a_2, \ldots, a_n\}$, we can generate a typed sequence where $\mathcal{T}_0 = \varphi_{I,o}$ is the typed sub-state for the initial state, and each $\mathcal{T}_i$ is the typed sub-state obtained from the state s reached when the action a_i is applied.

Since typed sequences focuses on particular objects, not all actions in a solution plan may affect that object. Thus, we define a void operator (*no-op*) that substitutes the real one in case the object is not a parameter of the corresponding applied action. Therefore, in any sequence step $(\mathcal{T}_i, no\text{-}op)$, we assume that $\mathcal{T}_i = \mathcal{T}_{i-1}$ and the action a_i is not relevant for the object generating the sequence. Additionally, we use an integer, together with the *no-op*, to indicate the number of times that irrelevant actions have been applied. This allows us to maintain a compact representation of typed sequences. Any action producing a *no-op* at the end of a sequence is not taken into account, since the rest of the plan has no relation with the particular object and there is no need to collect any additional transitions for the object.

Figure 1 shows a complete typed sequence for an object (block A) of type *Block* in the *Blocksworld* domain. The

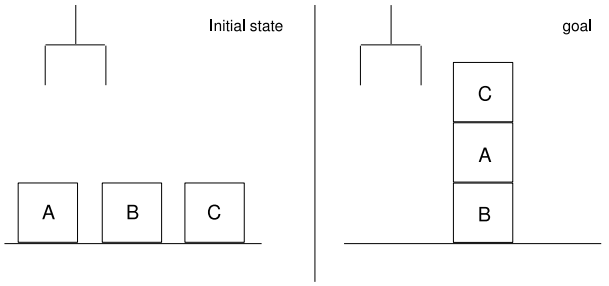
first pair in the sequence has the typed sub-state corresponding to the initial state and it has no associated action. The other pairs are generated as explained before. The action `(PICKUP C)` produces a *no-op* action, because it does not affect block *A* and the typed sub-state remains the same.

Definition 7 Given a relaxed plan $\pi^+ = \{a_1, \ldots, a_n\}$, the **relaxed plan footprint** for an object o is the sequence of pairs $\{(\mathcal{O}_{1,o}, \mathcal{T}_1), \ldots, (\mathcal{O}_{n,o}, \mathcal{T}_n)\}$ where $\mathcal{O}_{i,o}$ represents the set of operators that affect object o in the layer $i-1$ of the RPG and add some properties for the collection $\mathcal{T}_i$ in layer i. It is computed as

$$\mathcal{O}_{i,o} = \left\{ opt(a) \mid a \in \pi^+ \wedge a \in A_{i-1} \wedge o \in argop(a) \right\}$$

Each $\mathcal{T}_i = \varphi_{s_i^+,o}$ is the collection of added properties where $s_i^+ = \{add(a) \mid a \in \pi^+ \wedge a \in A_{i-1}\}$.

The relaxed plan footprint is a compact way of partially representing a typed sequence. Thereby, the footprint can serve as a retrieval key, since it can be computed at the beginning of the planning process (i.e., after computing the heuristic estimation of the initial state). The relaxed plan footprint has two main differences with a normal typed sequence. First, it only keeps track of added properties through consecutive RPG layers in the relaxed plan of the initial state. Second, it stores the associated operators by layers. Therefore, there is no precedence relation among them. Figure 2 shows the computation of the relaxed plan footprint for blocks *A*, *B* and *C* of the example in Fig. 1. The relaxed plan for the initial state coincides with the real plan, even though the relaxed plan footprint differs from the typed sequence generated for the example.



Plan	Typed Sequence (Block A)
Initial State	$[(clear_1\ ontable_1), \emptyset]$
0: (PICKUP A)	$[(holding_1),\ pickup]$
1: (STACK A B)	$[(clear_1\ on_1),\ stack]$
2: (PICKUP C)	$[(clear_1\ on_1),$ `no-op`$]$
3: (STACK C A)	$[(on_1\ on_2),\ stack]$

Fig. 1 An example of a typed sequence in the *Blocksworld* domain

4 CBR cycle

In this section we explain how typed sequences are used in a complete CBR cycle [1]. This cycle comprises retaining, retrieving, reusing and revising the cases of a particular domain. The learning system has been developed as a trainer/advisor module that supports a heuristic planner during the search process, as shown in Fig. 3. This system was built within SAYPHI, a planning and learning framework in which different learning techniques can be integrated with a common planner [7, 8].

The system flow is as follows. A set of training problems is solved by the heuristic planner, and the solutions are used to generate typed sequences and populate the case base (i.e., the "Retain" process in Fig. 3). In order to solve new problems, modified versions of the heuristic search algorithms request similar sequences that match the initial state, goals and relaxed plan footprint of the new problem (i.e., the "Retrieval" process). The retrieved sequences are used to recommend node ordering or pruning according to the states that reproduce the same transitions encoded in the sequences (i.e., the "Reuse" process). Finally, new solutions and an evaluation score are used to recognize which types are useful in the case base and update it accordingly (i.e., the "Revision" process). Next, we give an explanation of each phase.

4.1 Retain

This process manages the generation and maintenance of the case base. Its main tasks consist of (a) the generation of typed sequences from a set of training problems, (b) the case storage in the case base structure and (c) the maintenance,

Fig. 2 An example of how to compute the relaxed plan footprint

	$A_0 \in \pi^+$	s_1^+	$A_1 \in \pi^+$	s_2^+
Relaxed Plan	(PICKUP A)	(*holding A*)	(STACK A B)	(*on A B*) (*clear A*)
	(PICKUP C)	(*holding C*)	(STACK C A)	(*on C A*) (*clear C*)
Footprint A	PICKUP	$holding_1$	STACK,STACK	$on_1, clear_1, on_2$
Footprint B	$\emptyset$	$\emptyset$	STACK	on_2
Footprint C	PICKUP	$holding_1$	STACK	$on_1, clear_1$

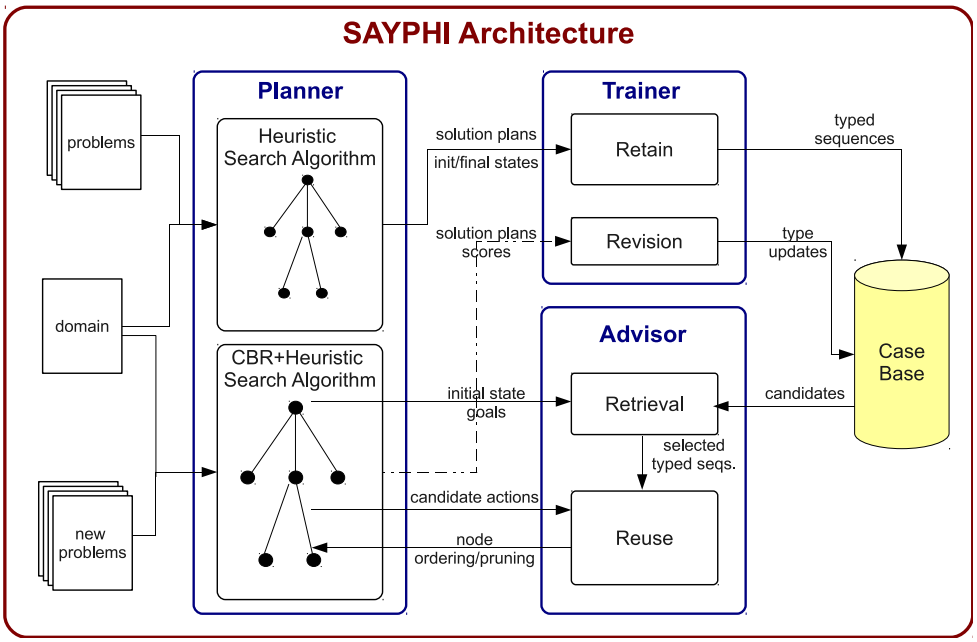


Fig. 3 SAYPHI: a heuristic planner supported by a CBR system

which implies merging repeated sequences, deleting useless ones, etc. From a more general point of view, all these tasks represent the training phase of the learning system.

4.1.1 Storage

This task stores the generated sequences in domain-dependent structures for later use. A case for a particular domain $\mathcal{D}$ is a tuple $\{i, \mathcal{F}, \mathcal{Q}\}$, where i is a case number, $\mathcal{F}$ is the relaxed plan footprint, and $\mathcal{Q}$ is the typed sequence. The relaxed plan footprint is computed after evaluating the initial state with the heuristic function of the relaxed plan. Relaxed plan footprints collect information that can help to recognize similar sequences that differ from each other in their middle steps.

4.1.2 Generalization

Once cases have been generated, a generalization process tries to merge new sequences with previous ones. For this purpose, we define the concept of equal and equivalent sequences.

Definition 8 Two typed sequences are **equal** if all their ordered pairs $(\mathcal{T}_i, op_i)$ are equal.

Definition 9 Two typed sequences are **equivalent** if when removing in both sequences all pairs $(\mathcal{T}_i, op_i)$ such that $op_i = no\text{-}op$, the resulting sequences are equal.

Thus, all pairs of equal or equivalent sequences are merged in order to generate a new sequence. Figure 4 shows the algorithm for the merging process. If the function *equivalent-sequences* returns a new sequence, it is stored in the case base and the input sequences are removed. If the new sequence is completely new (i.e., function equivalent-sequences returns *null*), it is stored as is in the case base.

equivalent-sequences $(\mathcal{Q}, \mathcal{Q}')$: Q_m typed sequence or null

$\mathcal{Q} = \{(\mathcal{T}_0, \emptyset), \ldots, (\mathcal{T}_n, a_n)\}$: Sequence in the case base
$\mathcal{Q}' = \{(\mathcal{T'}_0, \emptyset), \ldots, (\mathcal{T}'_m, a'_m)\}$: New sequence

$i = 0$; $j = 0$; $Q_m = \{\}$
while ($\mathcal{T}_i \neq$ **null and** $\mathcal{T'}_j \neq$ **null**)
  if $(\mathcal{T}_i, a_i) = (\mathcal{T'}_j, a'_j)$ **then**
    add $(\mathcal{T}_i, a_i)$ to Q_m
    $i = i + 1$; $j = j + 1$
  elseif $(a_i = no\text{-}op)$ **then**
    $i = i + 1$
  elseif $(a'_j = no\text{-}op)$ **then**
    $j = j + 1$
  else
    return null
return Q_m

Fig. 4 The algorithm for merging two sequences

4.2 Retrieval

Given a new planning task $(\mathcal{C}, I, G)$ for the domain $\mathcal{D}$, the retrieval process searches the case base for the most similar cases to this task. Given that each planning state contains information on several objects, we retrieve one case for each

object in the planning task. For each object o in C, the features used by the similarity function are:

- the object type of o
- the typed sub-state generated from the initial state: $\varphi_{I,o}$
- the typed sub-state generated from the set of goals: $\varphi_{G,o}$
- the relaxed plan footprint computed for the initial state: $\{(\mathcal{O}_{1,o}, \varphi_{s_1^+,o}), \ldots, (\mathcal{O}_{n,o}, \varphi_{s_n^+,o})\}$

The object type is used to pre-select the subset of cases corresponding to its type, so it is not directly used to compute the similarity metric.

4.2.1 Similarity metric

The similarity metric used for typed sub-state comparisons takes into account that the number of properties of an object can vary depending on the size of the planning instance. Accordingly, we define three levels or degrees of matching to indicate how similar two typed sub-states are. We represent typed sub-states $\mathcal{T}$ as pairs (X, m), where X is the set of non repeated properties in $\mathcal{T}$, and $m(x)$ is the function representing the number of times the element $x \in X$ appears in $\mathcal{T}$. For instance, in the *Logistics* domain, if the object sub-state for the truck *Tr1* is $s = \{(at\ Tr1\ Loc1)\ (in\ Pkg1\ Tr1)\ (in\ Pkg2\ Tr1)\}$ the corresponding typed sub-state is $(at_1\ in_2\ in_2)$. This typed sub-state can be written as $((at_1\ in_2), (1\ 2))$ under this representation.

Definition 10 Given two typed sub-states $\mathcal{T}_1$ and $\mathcal{T}_2$, expressed as pairs in the form (X_1, m_1) and (X_2, m_2), we define the matching level as follows:

1. There is a **total match** of $\mathcal{T}_1$ with $\mathcal{T}_2$, represented as $\mathcal{T}_1 \subseteqq \mathcal{T}_2$, when all properties in $\mathcal{T}_1$ are present in the same amount in $\mathcal{T}_2$. Formally, this relation is defined as

$$\mathcal{T}_1 \subseteqq \mathcal{T}_2 \leftrightarrow X_1 \subseteq X_2 \wedge \forall x \in X_1 m_1(x) = m_2(x)$$

2. The relation $\mathcal{T}_1 \subsetneqq \mathcal{T}_2$ represents a **partial matching** of $\mathcal{T}_1$ with $\mathcal{T}_2$, when all properties in $\mathcal{T}_1$ are present in $\mathcal{T}_2$, but not all of them appear the same number of times. Formally, this relation is defined as

$$\mathcal{T}_1 \subsetneqq \mathcal{T}_2 \leftrightarrow X_1 \subseteq X_2 \wedge \exists x \in X_1 m_1(x) \neq m_2(x)$$

3. The relation $\mathcal{T}_1 \nsubseteqq T_2$ represents that $\mathcal{T}_1$ has no matching with $\mathcal{T}_2$ because not all properties in $\mathcal{T}_1$ are in $\mathcal{T}_2$. Formally, this relation is defined as

$$\mathcal{T}_1 \nsubseteqq \mathcal{T}_2 \leftrightarrow X_1 \nsubseteqq X_2$$

These relations are not commutative (i.e., $\mathcal{T}_1 \subseteqq \mathcal{T}_2$ does not imply $\mathcal{T}_2 \subseteqq \mathcal{T}_1$). A typed sub-state generated for an object in the new problem should be equal to the first/last step in the stored sequence, or a subset of this step, but not the opposite. This is because typed sub-state of goals could be partially defined (goals are partially defined states), but the last steps of a stored sequence have the complete typed sub-state. For instance, in the *Blocksworld* domain, the last step in the sequence generated for block B in Fig. 1 is $[(on_2\ ontable_1)$, *stack*$]$. However, if a block in a new problem does not need to be on another block in the goals, the goal typed sub-state is (on_2) if the *on-table* predicate is not specified as a goal, which normally occurs. Therefore, considering the defined matching levels for the retrieval, $\mathcal{T}_1$ will refer to a typed sub-state generated from an object in the new problem, and $\mathcal{T}_2$ will refer to a typed sequence in a stored sequence.

Relations in Definition 10 can be extended to relaxed plan footprints. Since relaxed plan footprints keep properties and operators by layers, in the extension it is relevant to take empty layers into account.

Definition 11 Two relaxed plan footprints have a **total match** if each pair $(\mathcal{O}_{i,o}, \mathcal{T}_i)$ is equal in both footprints. Two relaxed plan footprints have a **partial match** when they have a total or a partial match for each pair $(\mathcal{O}_{i,o}, \mathcal{T}_i)$, after removing in both sequences pairs such that $\mathcal{O}_{i,o} = \emptyset \wedge \mathcal{T}_i = \emptyset$.

The similarity metric is defined in terms of the matching level for the initial and goal sub-states. The matching level for the relaxed-plan footprints is used to break ties if two cases obtain the same value for the similarity metric with the new case. A tie in the similarity metric normally occurs when two sequences have the same initial and goal typed sub-states. Then, the tie needs to be broken taking into account the intermediate steps, partially abstracted by the relaxed plan footprint. Given an object o of the new planning task (C, I, G), the similarity metric is computed with Eq. (3).

$$\mathcal{M}(o, \mathcal{Q}_i) = 2W(\varphi_{I,o}, \mathcal{T}_0) + W(\varphi_{G,o}, \mathcal{T}_n) \tag{3}$$

where $\mathcal{Q}_i$ is the sequence i in the case base of the same type of o. The function W returns a numerical value for distinguishing relations described in Definition 10. The match for the initial typed sub-states is more relevant in the equation because goal typed sub-states are partially defined. Moreover, in any given planning task this value is 0 for many objects (i.e., the object does not appear in any goal). For simplicity we use the following equivalence for the matching relations.

$$W(\mathcal{T}_1, \mathcal{T}_2) = \begin{cases} 2 & \text{if } \mathcal{T}_1 \subseteqq \mathcal{T}_2 \\ 1 & \text{if } \mathcal{T}_1 \subsetneqq \mathcal{T}_2 \\ 0 & \text{if } \mathcal{T}_1 \nsubseteqq \mathcal{T}_2 \end{cases}$$

If there is a tie in the similarity metric, the function W is applied to the footprints of each sequence, and the one with greater value is selected.

sequence-selection ($\mathcal{B}$, $\mathcal{C}$, I, G): $\mathcal{RQ}$ replay-table

$\mathcal{B}$: domain case base.
$\mathcal{C}$: Set of constants (objects) in the planning task.
I, G: Initial state and goals.

$\mathcal{RQ} = \emptyset$
for each o in $\mathcal{C}$
  $\mathcal{K} \leftarrow \{k = \{i, \mathcal{F}_i, \mathcal{Q}_i\} \mid k \in \mathcal{B}$ and
    k is of the same type as $o\}$
  $\mathcal{K}' \leftarrow \{k' \mid k' = \arg\max_{\{i,\mathcal{F}_i,\mathcal{Q}_i\}\in\mathcal{K}} \mathcal{M}(o, \mathcal{Q}_i)\}$
  $\mathcal{F}' \leftarrow$ *rp-footprint*$(o,$ *relaxed_plan*$(I, G))$
  $k_{best} = \arg\max_{\{i,F_i,\mathcal{Q}_i\}\in\mathcal{K}'} W(\mathcal{F}_i, \mathcal{F}')$
  Add k_{best} to $\mathcal{RQ}$
return $\mathcal{RQ}$

Fig. 5 Algorithm for ranking and selecting typed sequences for a new problem

4.2.2 Ranking and selection

The goal of the retrieval process is to build the *replay table*. It is defined as the set of pairs $\langle \mathcal{T}_i, o_i \rangle$, where $\mathcal{T}_i$ are the retrieved sequences, and o_i is the object it was retrieved for. Figure 5 shows the pseudo-code for building the *replay table*. For each object in the new problem, a subset of sequences maximizing the similarity metric is selected. Then, from this subset, the sequence with the best match in the relaxed plan footprint is added to the replay table as the retrieved sequence for the object. The function *rp-footprint* computes the relaxed plan footprint of the initial state for the object o according to Definition 7.

4.3 Reuse

The idea of reusing the knowledge kept in the typed sequence consists of replaying the typed sub-state transitions during the forward state search. The retrieved sequences are managed through the replay table, which keeps track of the current step being followed for each sequence, as in previous work on case-based planning [25]. The advice given by typed sequences is usually independent of the search algorithm. Accordingly, we use the concept of a "recommended node" to recognize a promising successor suggested by the CBR component; that is, it reproduces a sub-state transition for an object in the replay table. We denote the replay table with the symbol $\mathcal{RQ}$ and the function *next_step*$(\mathcal{RQ}, o)$ returns the next pair $(\mathcal{T}_i, \mathcal{OP}_i)$ of the sequence retrieved for object o. Given a state s, the successors are represented as pairs (s', a), where $s' = apply(a, s)$. In order to recognize whether a successor replays the current transition in the retrieved sequence for an object o, we define a function:

$$\Gamma(\mathcal{RQ}, o, s', a) = \begin{cases} 1 & \text{If } next\_step(\mathcal{RQ}, o) = (\mathcal{T}_i, \mathcal{OP}_i) \\ & \quad \wedge\, o \in argop(a) \wedge opt(a) = \mathcal{OP}_i \\ & \quad \wedge\, W(\varphi_{s',o}, \mathcal{T}_i) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Definition 12 A **recommended node** is a candidate node that reproduces the typed sub-state transition in the sequence retrieved for a parameter (object) of the applied action in the node. Formally, a successor (s', a) is a recommended node if:

$$\exists o \in argop(a) \mid \Gamma(\mathcal{RQ}, o, s', a) = 1$$

Definition 13 The **recommendation ratio** for a successor is the level of recommendation of a node related to the total number of action parameters of that node. Formally, the recommendation ratio for a successor (s', a) is defined with the function

$$r_{cbr}(\mathcal{RQ}, s', a) = \frac{\sum_{\forall o \in argop(a)} \Gamma(\mathcal{RQ}, o, s', a)}{|argop(a)|}$$

The recommended ratio is used for ranking successors in search algorithms, with the aim of prioritizing those nodes that replay the most sequences.

4.4 Advising heuristic search

The control knowledge given by the typed sequences can be integrated within any heuristic search algorithm. There are two basic strategies of using retrieved sequences:

- **Pruning Strategy**: a recommended node (if exists) can be directly selected, discarding other successors. This acts similar to a state-action policy.
- **Ordering Strategy**: a recommended node can be preferred for evaluation in a greedy algorithm that distinguishes between node generation and node evaluation.

In the following sections we present how these strategies can be integrated into three search algorithms commonly used in planning.

4.4.1 CBR hill-climbing

The first straightforward application of the pruning strategy consists of directly selecting a recommended node whenever advice is available among the current state successors. For instance, after a node expansion in the hill-climbing algorithm, if a successor is recommended, it is selected and no other evaluations are performed. If there is no advice, all nodes are evaluated and the one with the best heuristic estimation is selected. Figure 6 shows the complete pseudo-code for this algorithm.

Figure 7 shows an example of how typed sequences are replayed in CBR hill-climbing when solving a problem in the *Blocksworld* domain. The top of the illustration shows the configuration of the initial and final state. At the bottom, there is a plan that solves the problem along with the four retrieved sequences, one for each block in the problem. In this domain, all objects are of type *Block*. Next, we explain how the replay occurs. The search tree in this algorithm only considers *helpful* actions. The search can also be followed using the search tree in Fig. 8.

CBR-Hill-Climbing $(I, G, \mathcal{R}Q)$: *plan*

I, G: Initial state and goals
$\mathcal{R}Q$: replay table

$s \leftarrow I$; *solved* = **false**
while ($s \neq$ **null and not** *solved*)
  if $G \subseteq s$
    then *solved* = **true**
  else
    $S' \leftarrow$ expand-node(s)
    if $\max r_{cbr}(\mathcal{R}Q, s', a) > 0$ **then**
      $s \leftarrow \arg\max_{s' \in S'} r_{cbr}(\mathcal{R}Q, s', a)$
      evaluate $h(s)$
    else
      evaluate $h(s'), \forall s' \in S'$
      $s \leftarrow \arg\min_{s' \in S'} h(s')$
if *solved*
  return *path* (I, s)
else
  return *failure*

Fig. 6 A hill-climbing algorithm guided by CBR recommendations

1. The only two successors of the initial state, `(UNSTACK A B)` and `(UNSTACK C D)`, have both recommended ratio 1, because their two arguments match the transitions in their respective sequence. Both actions are correct for finding an optimal plan. The search selects the action `(UNSTACK A B)` since it appears first after the node expansion.
2. Action `(PUTDOWN A)` matches the third step in the sequence retrieved for block *A*. It has ratio 1. The action `(STACK A C)` has ratio 0, because it does not match the transition for block *A* or block *C*.
3. Actions `(PICKUP B)` and `(UNSTACK C D)` have ratio 1. The second one is preferred because in this action two sequences are being followed (i.e., sequences for block *C* and *D*). When a tie in *r* occurs, it is solved preferring the action that replays more sequences.
4. Action `(PUTDOWN C)` is selected because it has ratio 1. `(STACK B C)` has ratio 0. The action `(STACK C A)` has ratio $\frac{1}{2}$ because block *C* does not match, but the current transition of block *A* does, since it is expecting

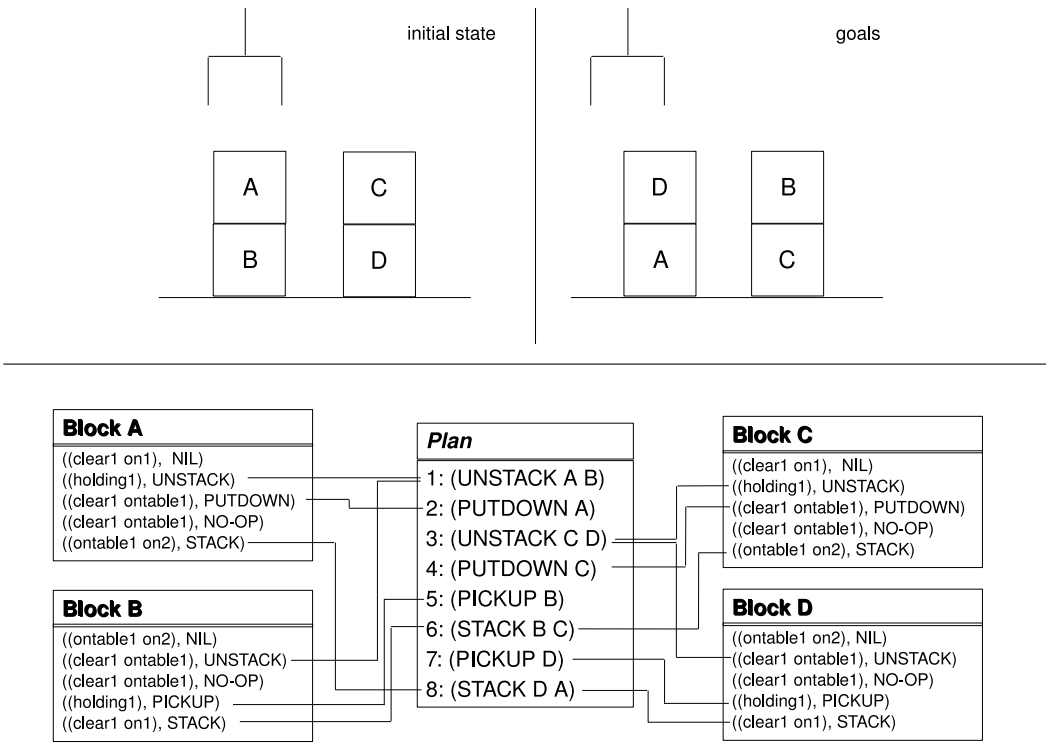


Fig. 7 Example of replaying typed sequences in the CBR Hill-Climbing algorithm

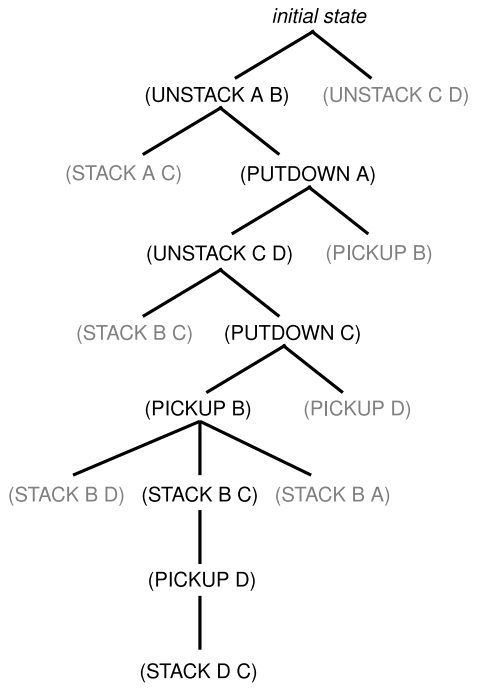


Fig. 8 The search tree resulting for the replay process of the example in Fig. 7. *Nodes* in *gray* are helpful actions not selected by the recommendations

to have a block on it in the transition $\{(ontable_1\ on_2), stack\}$.

5. Actions `(PICKUP B)` and `(PICKUP D)` have ratio 1. In this situation we can clearly appreciate how the pruning of the helpful actions technique chooses a small and coherent set of successors (i.e., picking up block *A* or *C* are not considered as successors). Both actions in the helpful action set are correct. Action `(PICKUP B)` is selected because of the expansion order.
6. Action `(STACK B D)` has ratio $\frac{1}{2}$, because block *B* matches, but block *D* does not. Actions `(STACK B C)` and `(STACK B A)` have ratio 1. In this situation, the right choice is selected by chance (i.e., it appears first in the successors). Action `(STACK B A)`, even with ratio 1, could produce a wrong choice in the search. This action appears as helpful because it reaches the sub-goal of having the robot arm empty, which is needed later.
7. Action `(PICKUP D)` with ratio 1 is the only successor.
8. Action `(STACK D C)` with ratio 1 is the only successor. With this action the problem goals are reached.

4.4.2 CBR EHC

This algorithm modifies the *Enforced Hill-Climbing* algorithm (EHC) [15] to take advantage of the greedy feature in terms of node evaluations. EHC performs a breadth-first search from a node s until it finds a node s' that has a better heuristic value. This process repeats iteratively until finding the goals or running out of successors. States in the same breadth level are evaluated in the order given by the parent node expansion. This means that if in a certain breadth level, a state improving the heuristic estimation is evaluated first, the rest of evaluations can be avoided. The modification for EHC that integrates CBR recommendations affects the evaluation node ordering. After a node expansion, each successor is evaluated only if it is a recommended node. Otherwise, the evaluation is postponed and the node is placed into a *delayed list*. If none of the recommended nodes improves the current heuristic estimation, the rest of successors are evaluated and the algorithm continues normally.

This modification to EHC seems interesting because it keeps the same layout for the algorithm, giving the opportunity of avoiding useless node evaluations. The main drawback is that CBR-EHC still relies upon the heuristic function. Therefore, the domain topology and the existence of different types of *plateaus* [14] will affect the final performance of this algorithm. Suppose that EHC needs to expand the breadth-first search until a certain depth d in order to escape a plateau. If there is only one node at depth d that improves the heuristic estimation, the evaluation of this node (if marked as recommended) can avoid the rest of evaluations. On the other hand, if all nodes at depth d are plateau exits, matching states for finding a CBR advice becomes a waste of time. Accordingly, only an experimental evaluation can determine the benefit of this technique in certain domains.

4.4.3 CBR WBFS

The *Weighted Best-First Search* (WBFS) is another algorithm commonly used in forward heuristic planning. The CBR version tries to use the typed sequence recommendations to break ties in the evaluation function $f(n) = g(n) + w_h \times h(n)$. In order to keep the same layout for the algorithm, we just modify the evaluation function with the formula:

$$f(n) = g(n) + w_h \times \big(h(n) + 1 - h_{cbr}(n)\big)$$

where $h_{cbr}(n)$ returns the recommended ratio computed for the successor (s', a) in n. The idea consists of modifying the value that $h(n)$ contributes to the standard formula. Thereby, a recommended node (if any) will be preferred to continue the search. The new evaluation function is only useful when $w_h > 1$ because otherwise the algorithm will not take advantage of the new ordering imposed to $f(n)$. Let us consider an example with an admissible heuristic. If s_g is a goal state, all generated nodes that hold $f(n) < g(s_g)$ should be evaluated before stopping the search, therefore it does not make sense to order the evaluation of these nodes. Although the relaxed plan heuristic is non-admissible, some experimental evaluations have shown a similar effect. Nevertheless, WBFS with

$w_h > 1$ has the effect (which increases with bigger w_h) of not reconsidering specific nodes while it is descending in the search tree. This means that the node ordering is relevant and preferring more promising nodes could affect the final performance.

CBR-WBFS has some differences with greedy algorithms as CBR-Hill-Climbing or CBR-EHC. In greedy search algorithms, a single replay table can keep track of the current step of sequences. However, in CBR-WBFS, nodes being expanded can belong to different paths, so it is necessary to keep track of the current step of the replay table followed in each path. Therefore, in WBFS-CBR, each node holds a set of references indicating the current step being followed at that node.

4.5 Revising typed sequences

This process consists of revising which types in a case base contribute to the improvement of the CBR planning algorithms. The refinement of domain types is relevant to the CBR cycle because sequences of certain types can be meaningless. For example, in transportation domains such as *Logistics*, objects being transported can generate sequences that store knowledge like {*Load-in-Vehicle*, *no-ops*, *Unload-from-Vehicle*}. This information clearly stores the process of moving packages. However, the domain type for places (*location*) can generate sequences with transitions that do not hold anything relevant in the problem. Package locations are represented with predicate *at*, in the form (*at object place*). Thus, the typed sequences generated for the object *place* will only keep different occurrences of the at_2 property, and these type of sequences do not necessarily contribute to a right advice during the replay process.

Accordingly, we have designed a refinement process that evaluates how a domain case base performs in comparison to the same case base but removing sequences of specific types. Figure 9 shows the pseudo-code that determines the best set of types for a domain given a validation set of problems V and the planning algorithm α (i.e., one of the CBR variations presented in the previous section). The function *evaluate_set* runs the problems in V with the algorithm α using the sequences of types in T_j. The function *score* gives the score to T_j types using the performance time metric used in the IPC-2008, explained in the next section. The *Revising-Types* algorithm reproduces a hill-climbing algorithm using *score* as the evaluation function. The algorithm ends when the best set of types is equal to the set selected in the previous iteration.[1]

Revising-Types $(\mathcal{B}, T, V, \alpha)$: *types*

$\mathcal{B}, T$: domain case base and domain types
V: validation set
α: planning algorithm

$T' \leftarrow T$; $T_{best} \leftarrow \emptyset$
while ($T' \neq T_{best}$ **and** $T' \neq$ **null**)
  $T_{best} \leftarrow T'$
  candidates $\leftarrow \{T'\}$
  for each $t_i \in T'$
    $T_{-t_i} \leftarrow T' - t_i$
    Add T_{-t_i} to *candidates*
  for each $T_j \in$ *candidates*
    evaluate_set (V, α, T_j)
  $T' \leftarrow \arg\max_{T_j \in candidates}$ score (V, T_j)
return T_{best}

Fig. 9 The algorithm for refining domain types

Finally the case base B is updated by removing all sequences that belong to types not present in the outcome of the *revising-types* function.

5 Experimental evaluation

This section presents the experiments we have done to evaluate the case-based approach to heuristic planning. We have evaluated this approach in eight STRIPS benchmarks. Two of them (*Matching Blocksworld* and *Parking*) were used in the IPC-2008 Learning Track. The rest of them (*Blocksworld*, *Logistics*, *Depots*, *Mystery'*, *Satellite*, *Rovers*) are benchmarks commonly used in the planning community. These six domains belong to five different domain classes according to Hoffmann's domain topology [14],[2] which guarantees a reasonable domain variety for evaluation. All domains have random problem generators available in IPCs webs. These generators were used to create different problem sets during training and test phases.

5.1 Training phase

For each domain we generated a training set of 20 problems using the random problem generators. These problems are of small size so that it is possible to find a good quality plan in reasonable time. Each problem was solved with EHC and then, the solution was refined with a *Depth-First Branch and Bound* (DFBnB) using a time bound of 60 seconds. The best-cost solution found by DFBnB was used to generate the

[1] As the evaluation metric used in IPC-2008 takes into account the performance of other planners/configurations (relative scoring), the best set of types of an iteration is included in the candidates of the next iteration.

[2] The *Matching Blocksworld* and *Parking* domains have not been classified in the domain topology because [14] was published prior to the IPC 2008 Learning Track.

typed sequences in order to populate the domain case base. The time for generating typed sequences is negligible with respect to the time bound (60 s) for solving each problem. The computational cost of generating a typed sequence is linear in the plan length, and the number of sequences is bounded by the number of objects. We have also generated a validation set of 20 problems for type refinement. After a case base was generated with the training set, the *revising-types* algorithm was executed to obtain the best set of types. Then, we removed from the case base all typed sequences belonging to a type not present in the set of best types.

5.2 Evaluation metrics

For each domain we have computed the number of solved problems. We have also evaluated the different planning algorithms with the metrics used in the IPC-2008 Learning Track. These metrics measure planning performance in terms of quality (plan length) and CPU Time. Final scores in both metrics are the sum of points given to each problem in a particular domain. For the quality metric, a planner configuration receives N_i^*/N_i points, where N_i^* is the minimum number of actions in any solution returned by a planner for the problem i, and N_i is the number of actions returned by the evaluated configuration for the problem i. If the evaluated planner does not solve the problem, it receives 0 points for this problem. The score for a problem is in the range of [0, 1]. Likewise, the time metric is computed giving a planner configuration T_i^*/T_i points, where T_i^* is the minimum time used by any planner for solving the problem i, and T_i is the time used by the evaluated planner for solving the problem i. The planner receives 0 points if it does not solve the problem.

5.3 Test phase set up

For the *Matching Blocksworld* and *Parking* domains we have used the 30-problem sets from the *target* distribution given to the competitors. These distributions have the same difficulty as the sets used in the competition. For the rest of the domains, we have built a test set of 30 problems of incremental size, using the random problem generators. We have not used the competition sets because, on the one hand, they have a different number of problems, which makes the comparison with the current metric unfair. On the other hand, the improvements in planning techniques and hardware processing speed have made those sets less attractive for evaluation purposes. For instance, the set used for the *Logistics* domain in 2000 had 41 goals as the hardest problem. The set used in this evaluation has a maximum of 120 goals.

For the evaluation we used a time bound of 900 seconds for each problem, as did the IPC 2008 Learning Track. The whole experiment was run in an Intel Core 2 Quad CPU with 3 Gb of memory-bound for the process. All tested configurations share the same parsing and pre-processing functions, as well as the relaxed plan heuristic. Therefore, the real difference comes from the algorithms. The planning algorithms tested in the evaluation were:

- **HC**: The hill-climbing algorithm with the helpful actions pruning and a chronological backtracking when the search process runs out of candidates.
- **CBR-HC**: The CBR Hill-climbing algorithm, implemented with the same features of HC.
- **EHC** The Enforced Hill-climbing algorithm with the helpful actions pruning.[3]
- **CBR-EHC** The CBR-EHC algorithm implemented with the same features of EHC.
- **WBFS** The Weighted Best-First Search without helpful actions pruning. It uses a $w_h = 3$. Bonet and Geffner [3] reported that w_h values between 2 and 10 do not produce significant differences.
- **CBR-WBFS** The CBR-WBFS algorithm implemented with the same features of WBFS.

5.4 Results

Table 1 presents the number of solved problems in the eight evaluation domains for all tested configurations. The best

[3] This planning configuration differs from FF search algorithm, because we do not activate the WBFS when EHC fails. This allows us to recognize in which circumstances EHC is performing well alone.

Table 1 Solved problems for the evaluated planning configurations

Domains	HC	CBR-HC	EHC	CBR-EHC	WBFS	CBR-WBFS
Blocksworld	17	**18**	9	8	8	9
Matching-bw	0	0	0	0	**6**	**6**
Parking	21	**25**	12	12	7	7
Logistics	16	21	**26**	25	7	8
Mystery'	9	20	25	**27**	22	23
Depots	25	27	**29**	**29**	21	22
Satellite	27	**29**	19	22	10	9
Rovers	**28**	27	22	27	9	8
Total	143	**167**	142	150	90	92

Table 2 Scores for CPU Time metric

Domains	HC	CBR-HC	EHC	CBR-EHC	WBFS	CBR-WBFS
Blocksworld	10.21	**14.41**	1.34	2.24	2.33	2.53
Matching-bw	0.0	0.0	0.0	0.0	3.25	**6.00**
Parking	14.64	**23.59**	1.15	4.17	0.53	0.51
Logistics	2.30	12.58	**19.45**	17.77	0.25	0.40
Mystery'	4.34	17.91	10.74	**21.64**	2.41	2.84
Depots	12.04	18.41	17.97	**22.32**	2.55	3.16
Satellite	10.69	**25.75**	8.76	16.58	0.22	0.17
Rovers	18.93	**26.55**	10.32	21.07	0.37	0.27
Total	73.15	**139.20**	69.73	105.79	11.91	15.88

Table 3 Scores for quality metric

Domains	HC	CBR-HC	EHC	CBR-EHC	WBFS	CBR-WBFS
Blocksworld	6.31	8.48	7.46	6.60	7.64	**8.53**
Matching-bw	0.00	0.00	0.00	0.00	5.77	**6.00**
Parking	13.12	**15.67**	10.09	10.48	6.76	6.29
Logistics	4.94	10.47	**25.49**	20.82	7.73	8.80
Mystery'	6.97	16.10	20.86	**25.72**	20.91	20.84
Depots	7.56	7.96	25.60	**26.86**	20.31	21.09
Satellite	21.87	**22.73**	16.91	20.97	9.79	8.78
Rovers	22.21	21.99	21.39	**26.22**	8.88	7.81
Total	83.28	103.83	127.68	**141.18**	86.86	87.13

result per row is presented in bold. The last row shows the sum of solved problems for each configuration. CBR-HC was the algorithm that solved the most problems. Note also that each CBR algorithm solved more problems than its standard version. CBR-HC solved 24 more problem than HC, which shows the benefit of the CBR pruning strategy. CBR-EHC solved 8 more problems than EHC, revealing a discrete improvement when using the ordering strategy in greedy search. However, CBR-WBFS only solved 2 more problems than WBFS, indicating that CBR is more appropriate for greedy search algorithms than for best first techniques.

We give a detailed analysis for each domain further in this section. Table 2 shows the final scores using the time metric. Each domain can receive from 0 to 30 points. Getting the score of 30 means that the planner solved thirty problems in the test set and no other planner solved any problem in less time. The last row presents the sum of the scores obtained in each domain. CBR-HC was the algorithm that got the top score for the time metric. As in solved problems, each CBR algorithm obtained better results than its normal algorithm, especially for greedy search algorithms (in those algorithms the CBR version got double the score than its corresponding non-learning version). Even when the number of solved problems is the same or fairly similar, CBR-HC and CBR-EHC could solve the same problems in less time. The main reason for this improvement is the reduction in the number of evaluations during the search. Likewise, Table 3 shows the scores using the quality metric. Again, each CBR algorithm improved its standard version. In this case CBR-EHC got the top score. This difference arises, as we will see later in this section, because in some domains HC and HC-CBR obtained plans of poor quality.

Table 4 Node evaluation average for problems solved by Hill-Climbing and CBR Hill-Climbing

Domains	Common	HC	CBR-HC
Blocksworld	15	3823.5	**3235.0**
Matching-bw	0	*	*
Parking	21	3270.9	**2681.4**
Logistics	16	11016.4	**7139.8**
Mystery'	9	73412.1	**39.7**
Depots	25	**1836.5**	**1859.5**
Satellite	27	4039.8	**1902.7**
Rovers	26	1931.0	**1329.9**

Table 4 presents the node evaluation average obtained by HC and CBR-HC in problems solved by both configurations (indicated by the "common" column). CBR-HC evaluated fewer nodes in six out of eight domains on average. We do not show a comparison in the *Matching Blocksworld* domain, because no configuration solves any problem. Likewise, Table 5 shows the node evaluation average for EHC and CBR-EHC in problems solved by both configurations.

Table 5 Node evaluation average for problems solved by EHC and CBR-EHC

Domains	Common	EHC	CBR-EHC
Blocksworld	7	3543.1	**2176.4**
Matching-bw	*	*	*
Parking	10	10456.3	**4847.5**
Logistics	12	**1592.6**	4643.9
Mystery'	25	**1120.1**	2647.6
Depots	29	3627.4	**1531.2**
Satellite	19	1992.6	**1055.1**
Rovers	22	2629.1	**1036.4**

Table 6 Node evaluation average for problems solved by WBFS and CBR-WBFS

Domains	Common	WBFS	CBR-WBFS
Blocksworld	8	**2104.5**	3491.5
Matching-bw	6	20878.2	**7791.9**
Parking	6	11479.3	**7049.3**
Logistics	7	19724.4	**7396.3**
Mystery'	22	16136.5	**12872.3**
Depots	21	20676.5	**10898.7**
Satellite	9	**5859.1**	8636.9
Rovers	8	**5585.8**	10867.1

In this case, CBR-EHC was better in five out of eight domains. Finally, Table 6 shows the node evaluation average for WBFS and CBR-WBFS. Although CBR-WBFS was better in five domains, we do not consider these results as relevant as before since in many domains the number of commonly solved problems is small when compared to the problem set size. In the following sections we explain the results for each domain separately.

5.4.1 The Blocksworld domain

The Blocksworld domain is a well-known domain in automated planning. The tasks consist of configuring towers of blocks and placing them with an arm robot. Although it is one of the oldest benchmarks, it is still considered a challenging one. The domain only has the *Block* type, therefore the refinement process does not make any changes. The case base only contains 26 sequences, revealing that from a *Block* perspective, the set of possible transitions is restricted. Figure 10 shows the possible transitions between different typed sub-states in the domain. For instance, suppose that a block on the table with a block on top (sub-state 5) should finish on top of another block and clear (sub-state 2). To reach this goal, it is necessary that the block becomes clear and then held by the arm, before reaching the final position. This example matches blocks *B* and *D* on the example in Fig. 7. Using this knowledge, one can deduce a set

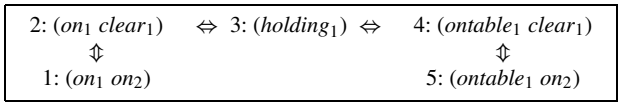


Fig. 10 Possible transitions between typed sub-states in the *Blocksworld* domain

of invariants for the domain. Nevertheless, typed sequences have the advantage of only keeping the knowledge appearing in a training example. For instance, the typed sequence with transitions (1-2-3-4-3-2) represents a common behavior for a block, which continuously appears in the training examples. By contrast, the sequence with transitions (5-4-3-2-3-4), even though it can be deduced from a domain analysis, never appears in a training example (i.e. in a good quality plan, one does not pick a block up from the table, put it on another block, and then place it back again on the table).

Regarding the results, CBR-HC solved one more problem than HC, but both algorithms produced plans of poor quality. On the other hand, the rest of algorithms solved fewer problems, just with a small improvement on CBR-WBFS, which obtains the top score for the quality metric. Although typed sequences seem to capture the right knowledge for this domain, using them at planning time does not offer a significant scalability improvement. First, there is a strong goal dependency in this domain, which can not be handled using an object-perspective approach as ours. Second, CBR algorithms still rely on the heuristic function, which is fairly misleading in this domain. In order to take advantage of the key knowledge stored in *Blocksworld* sequences it is necessary to complement the search with an additional technique such as a goal agenda or pre-computed landmarks [16].

5.4.2 The matching Blocksworld domain

This domain is a variation of the *Blocksworld* where each block has a polarity, either positive or negative. There are also two arms with polarity. The main difference with the standard *Blocksworld* is that if a block is held by an arm of wrong polarity, the block becomes damaged and no other block can be placed on the top of it. Blocks get damaged when they are placed by a wrong polarity arm, but not when picked up from the table or unstacked from other block. This fact makes recognizing dead-ends a difficult task. Even though there are alternative representations for this domain, it was designed in this form, to analyze difficulties that arise with the relaxed plan heuristic (i.e., the relaxed task never damages a block, thus both the relaxed plan and the heuristic estimation are wrong).

As in *Blocksworld*, tasks consist of placing towers of blocks in specific configurations. The refinement process kept the *Hand* type and left out the *Block* type. This does not make much sense because *Block* sequences capture the

right knowledge, including the properties $block\_positive_1$ and $block\_negative_1$ in their corresponding typed sub-states. So, the learned knowledge is useless at planning time because of two reasons. First, *Block* sequences can not recognize UNSTACK and PICK-UP actions as dangerous, because in those steps blocks do not lose the $solid_1$ property, the one that indicates the block is not damaged. The block loses this property in a following action if the used arm is of the opposite polarity. The second reason is that the relaxed planning graph does not take into account that blocks can be damaged (this appears in the delete lists, not considered in the computation of the relaxed planning graph). Therefore, the relaxed plan is rather different than the needed one, leaving the set of helpful actions incorrect. Thus, trying to recommend an action from a set of wrong ones is just a waste of time. Even with *Hand* sequences, the results reveal that algorithms using helpful actions pruning could not solve any problem. WBFS and CBR-WBFS solve the same number of problems, but CBR-WBFS gets top scores for time and quality metrics.

5.4.3 The parking domain

This domain comprises a set of curb locations where a set of cars can be parked. Each curb has the restriction of having at most two cars parked. The tasks consist of moving from one configuration of parked cars to another configuration of parked cars. The domain used in the IPC-2008 Learning Track has a flaw. A valid state can lead to an invalid state after applying the action MOVE-CURB-TO-CAR using the same constant for the moved car and the destination car (i.e. the car where the car is moved to). For our tests we used a modified version of this domain including the precondition (*not* (= *?car ?cardest*)) in the action. Thereby, no invalid states can be reached from a valid one. In the competition, heuristic planners could recognize these invalid states as dead-ends. Nevertheless, the evaluation of these nodes is useless because it is not possible to reach the goals from them. The main consequence of the flaw is the overhead caused by generating and evaluating the invalid states. We preferred to work with the fixed domain, since that way we can give a better interpretation of the scalability improvement achieved by the learning techniques. The refinement process does not discard any type in this domain. Sequences from both *Car* and *Curb* types were used for the test phase. CBR-HC performed very well in this domain, solving four more problems than HC. CBR-HC also got top scores for time and quality metric. Getting 15.67 points in quality with 25 solved problem reveals the algorithm did not get good quality plans. However, this ratio is acceptable when compared with other domains. The rest of the algorithms solved fewer problems, but each CBR algorithm performs better than its standard version. The improvements in the three

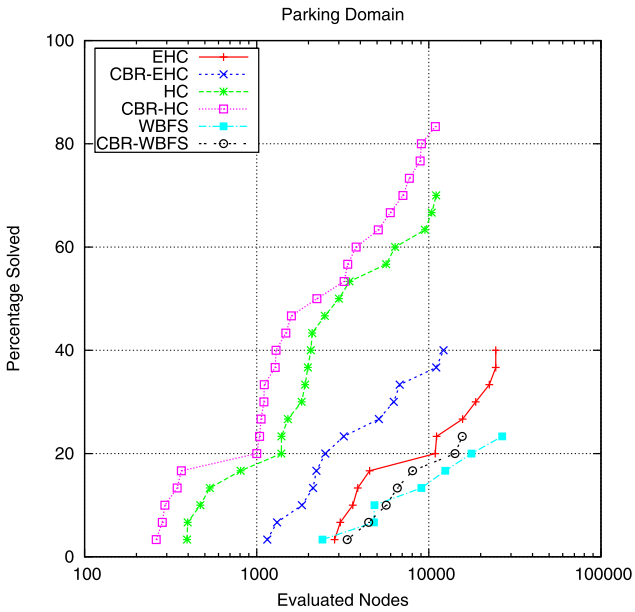


Fig. 11 Percentage of solved problems in increasing evaluated nodes in the *Parking*domain

CBR algorithms are due to a great reduction in the number of evaluated nodes, as can be appreciated in Fig. 11. The *x-axis* (in logarithmic scale) represents the number of evaluated nodes. Each point in the *y-axis* indicates the percentage of solved problems, when evaluating at most x nodes per problem. Thus, left-most lines indicate a better performance.

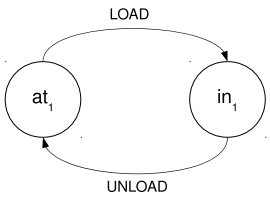
5.4.4 The logistics domain

This transportation domain is considered to be an easy benchmark. The difficulty of this domain comes when handling a large number of objects. The tasks comprise moving packages from different locations and different cities using trucks within cities and airplanes among cities. The refinement process left out the *City* and *Truck* types, keeping the *Package, Location, Airport* and *Airplane* types. In this case, the relaxed plan footprint is crucial for retrieving the right sequences. Suppose that a package in a location, producing the typed sub-state (at_1), needs to be in another location in the goal state, which also produces the same typed sub-state. Therefore, in this scenario, the only way of retrieving the correct sequence for the package is identifying the middle transitions it will have. Figure 12 shows all possible combinations of places for delivering a package in *Logistics*. Wide arrows indicate airplane movement and narrow arrows indicate truck movements. In the list, the combinations 2 and 3 correspond to the same typed sequence. The rest of the combinations produce one sequence each. Different actions for moving vehicles and properties achieved in different RPG layers make footprints distinguishable. Moreover, the *Package* type has exactly five sequences. All of them are retrieved correctly when solving new problems in *Logistics*.

1:	airport ⇒ airport
2:	location → airport
3:	airport → location
4:	location → airport → airport
5:	airport ⇒ airport → location
6:	location → airport ⇒ airport → location

Fig. 12 Different combinations of places for delivering a package in the *Logistics* domain

Fig. 13 A generic invariant for a cargo in the *Mystery'* domain



Regarding the results, CBR-HC solved five more problems than HC, but EHC got the best performance in terms of solved problems and evaluation metrics. CBR-EHC does not achieve any improvement, because the domain definition has the operators ordered in the right way such that preferring one successor for evaluation does not benefit the search process. Thus, grounded actions for loading or unloading packages from different vehicles appear before actions for moving vehicles. Actions in the first group always improve the heuristic value, so there is no need to re-order successors. Additional experiments using a *Logistics* version with the inverted actions confirmed this explanation. In this case, CBR recommendations gave the right advice for sorting actions and therefore for improving EHC.

5.4.5 The mystery' domain

This domain is a transportation domain also known as *Mprime*. The tasks comprise moving objects among different locations using vehicles of limited capacity and having fuel limitation. The refinement process selected the *Cargo* type as the only useful one. The other types (i.e., *Vehicle, Location, Space, Fuel*) were removed. Contrary to what happens in the *Logistics* domain, in the *Mystery'* domain all the transported objects have the same sequence. This typed sequence corresponds to the known invariant shown in Fig. 13.

Having this typed sequence only in the replay table forces CBR-HC to directly select action `LOAD` and `UNLOAD` whenever they appear as candidates (i.e., no other recommendation could be given). Additionally, actions for moving vehicles (`MOVE`) and actions for moving fuel (`DONATE`) are applied taking into account the heuristic estimation. This effect allows the algorithms to safely skip evaluations when actions for loading or unloading a cargo are part of helpful actions. The results show that CBR-HC solved 11 more problems than HC. In general, CBR-HC found plans with less node evaluations than HC. The great difference in the average though, was due to a single problem that CBR-HC solved with 56 evaluations, but HC evaluated more than 600 thousand nodes and took 412 seconds. Additionally, CBR-EHC solved 2 more problems than EHC and it got top scores for time and quality metrics.

5.4.6 The depots domain

The *Depots* domain is a combination of the *Logistics* and *Blocksworld* domains. The tasks are comprised of trucks transporting crates around depots and distributors. Using hoists, crates must be stacked onto pallets or on top of other crates at their final destination. As useful types, the refinement process chooses the set {*Crate, Depot, Hoist, Pallet, Truck*}. The *Distributor* type was left out. Both *Depot* and *Distributor* represent places, so we could think that if one of them is useless, the other is useless too. However, problems normally have more distributors, therefore the refinement prefer *Distributor* sequences because they produced more overhead. *Depot* sequences were not discarded because having them did not degrade the performance, however they did not provide valuable knowledge. Regarding the obtained results, EHC and CBR-EHC solved the same number of problems, but CBR-EHC got a better time score given that it reduces the number of evaluated nodes. It also obtained a slightly better result in the quality score. CBR-HC solved two more problems than HC. However, HC and CBR-HC performed poorly in terms of quality, due to goal interaction, which is relevant in this domain, as it is in the *Blocksworld* domain. WBFS and CBR-WBFS performed reasonably well, considering that they do not prune actions that are not helpful actions (i.e., they have to handle more nodes). CBR-WBFS solves one problem more than WBFS, getting slightly better scores in terms of time and quality.

5.4.7 The satellite domain

This domain comprises a set of satellites with different instruments, which can be operated in different modes. The tasks consist of taking images of certain targets in a particular format (mode). The refinement process does not mark any type as useless. Thus, all domain types (e.g. *Satellite*, *Instrument*, *Mode* and *Direction*) were used in CBR algorithms. HC-CBR solved more problems than any other algorithm, and it also obtained the best scores for the time and quality metrics. The difficulty of a problem relies on the number of objects, but tasks are essentially the same as in simple problems. Accordingly, CBR-HC uses typed sequences most of the time to directly select the next action, producing a great reduction in the number of evaluated nodes. Figure 14 shows the percentage of solved problems when increasing the number of evaluated nodes per problem. We can appreciate that both CBR-HC and CBR-EHC need much fewer nodes than their normal versions in order

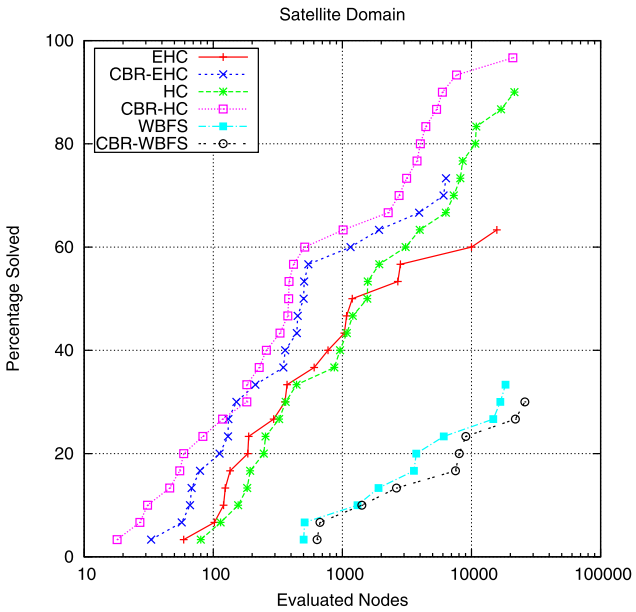


Fig. 14 Percentage of solved problems when increasing evaluated nodes in the *Satellite* domain

to solve the same percentage of problems. CBR-EHC obtains a good improvement over EHC because the node evaluation order is relevant in this domain. Actions such as CALIBRATE may be evaluated first, since it always improves the heuristic value of its parent state. However, the high branching factor makes both WBFS and CBR-WBFS a bad choice for solving problems in this domain. This branching factor is high because a satellite can point to any direction at any time.

5.4.8 The rovers domain

This domain is a simplification of the tasks performed by the autonomous exploration vehicles sent to Mars. The tasks consist of navigating the rovers, collecting samples of soils and rocks, and taking images of different objectives. The refinement process left out the *Waypoint* type and kept the rest of the types: *Camera, Lander, Mode Objective, Rover* and *Store*. This is probably because *Camera* sequences keep the information about taking images, and *Store* sequences about samples. Thus, *Waypoint* sequences only produce overhead in the recommendation computations. HC solved one problem more than CBR-HC, but CBR-HC showed a reduction in terms of time. These differences exist because CBR-HC could solve many problems faster than HC. CBR-EHC solved five more problems than EHC, revealing that the node evaluation order is also important in this domain. The reduction in the number of evaluated nodes (see Table 5) allowed the algorithm to scale better than EHC and therefore to solve more problems. It also got the top score for the quality metric. As happened in *Satellite*, WBFS and CBR-WBFS performed significantly worse due to the high branching factor imposed by not using the helpful actions.

5.4.9 Summary

The previous analysis shows that rather than having a general rule for predicting the performance of CBR algorithms, the result will depend on the domain representation and the characteristics of its search space. CBR-HC performs well in domains with a big branching factor and many solutions. In these cases, CBR-HC can achieve a big reduction in the number of evaluated nodes. This advantage reduces when problems have strong goal ordering (i.e., there is a fewer number of different solution paths). This weakness arises because retrieved typed sequences are not related to each other, and therefore the CBR algorithms have no information about which sequence is the best one to replay at each state, or which sequences should be replayed at the same time.

On the other hand, EHC-CBR is also able to reduce the number of evaluated nodes in several domains, and it is less affected by the goal ordering issue than HC-CBR. Nevertheless, EHC-CBR partially relies on the heuristic function as EHC does. Therefore, EHC-CBR will solve problems faster than EHC where EHC can lead the search to a solution. However both algorithms will perform poorly with a misleading heuristic. Finally, results showed that WBFS-CBR has no clear advantage over the other algorithms regardless of the evaluated domain. In general, altering $f(n)$ with a CBR recommendation value does not pay off if many nodes are expanded and evaluated anyway. For this reason we think that our approach is more suitable for greedy search algorithms rather than for best-first ones, such as WBFS.

6 Related work

Typed sequences are closely related to the domain invariants deduced with TIM [11]. The first versions of several domains written in PDDL (*Planning Domain Definition Language*) do not have types in their definition. Domain invariants helped to discover implicit types in the domain, which allows a GRAPHPLAN-like planner to instantiate a subset of possible actions, while taking into account the constraints that discovered types imposed. Nowadays, most domains have the object types in their PDDL definition. Domain invariants could be used to improve the guidance of a heuristic planner, but to our knowledge it has not been tried yet. On the other hand typed sequences have the advantage that they only code an observed object type transition. A finite state machine defined from TIM can encode all valid transitions, but not all of them may be part of a good plan. Additional research is needed to clarify in which domains a deduced invariant or a learned sequence could represent different guidance for a planner.

Our work is also related to OAKPLAN [23], a case-based planner that uses kernel functions to retrieve a set of similar problems with their associated plans. Then, one of these

plans is adapted to provide a solution. The main difference with our work is that their cases represent the whole planning problem while in our approach we can use information from different problems since we retrieve a case for every single object. Also, they perform case reuse as a local search on the retrieved plan, while we use previous cases to recommend nodes. Previously, other approaches were integrated in learning-based planners. A relevant work is PRODIGY/ANALOGY [25], where lines of reasoning are transferred to the new problem, reproducing/replaying the steps stored in the retrieved cases. Cases in ANALOGY represent the whole planning problem, therefore it is also different to our approach which is object-centered. Many other researchers have applied CBR to planning. The most relevant ones are summarized in [24] and [6]. More recently, CBR has been applied to hierarchical planning such as in HTN-MAKER [17]. HTN-Maker learns new methods for decomposing hierarchical tasks. It tries to reduce the engineering effort of designing the domain knowledge for controlling the search, which in the case of hierarchical planning is written within the action model. In our approach, the action model is fixed and the objective is to modify the search algorithm, so we can use the learned knowledge to guide the search.

Regarding other learning techniques applied to planning, the most representative vision comes from the set of participants of the IPC-2008 Learning Track. The system CABALA used typed sequences to support the generation of lookahead states from a relaxed plan. It performed poorly in the competition because the algorithm (not presented in this work) strongly depends on the quality of the relaxed plan. Other systems such as OBTUSEWEDGE [27], and ROLLER [9], performed better. These systems learn a generalized policy [19], which is a function that maps a meta-state to the action that should be applied. When it is hard to induce such a target function, a lazy learning technique such as CBR becomes an alternative for acquiring domain-dependent knowledge.

7 Conclusions

We have presented in this article an alternative approach to represent and use domain-dependent knowledge for handling scalability issues in domain-independent planning. The technique was developed within the case-based reasoning model, which involves saving, retrieving, using and revising the learned knowledge. We can summarize the contributions of this work as follows:

- Typed sequences: a compact way for abstracting plans and their corresponding state transitions from an object-centered perspective. Typed sequences can retain sub-state transitions that commonly appear in problems, which seems easier than deducing all possible sub-state transitions from a domain analysis.
- Relaxed plan footprint: a retrieval key that is used to recognize the right sequences when they have differences only in intermediate steps.
- CBR algorithms: Modifications to heuristic algorithms that allow us to integrate the knowledge given by typed sequences in a soft fashion, which also takes into account the guidance of the heuristic estimations.

Results on eight planning benchmarks showed the benefits of using typed sequences during the search process. Improvements obtained by different CBR algorithms were due to the reduction in the number of evaluated nodes. Since computing the relaxed plan heuristic is expensive in terms of time, any technique that can alleviate this burden will scale better in large problem instances. Accordingly, typed sequences can be complemented with other techniques that handle node evaluation issues. As future work we are planning to use CBR node recommendations to validate the action policy of ROLLER [9]. The new algorithm will propose a set of candidate actions to be applied using the relational decision trees, and then will select the actions that replay a typed sequence stored in the case base.

In addition, we want to research alternative representations for typed sequences, in order to enrich the information they can encode. As discussed previously, the CBR algorithms have no information of which sequence to replay first or which sequences should be replayed at the same time. We want to include information from the causal graph of the planning task [13] to handle this limitation. We also want to enrich typed sequences for handling more complex domains [10].

Finally, the planning community has some interest in synthesizing plans that contain user preferences. This idea is implemented through state preferences and giving rewards or penalties to certain actions. Thus, as future work, we plan to extend our approach by giving type sequences the possibility of acquiring user preference knowledge. As done in CBR algorithms, this information could be used to select or prefer certain actions among the search candidates.

Acknowledgements This work has been partially supported by the Spanish MEC projects PELEA: TIN2008-06701-C03-03 and PlanInteraction: TIN2011-27652-C03-02.

References

1. Aamodt A, Plaza E (1994) Case-based reasoning: foundational issues, methodological variations, and system approaches. AI Commun 7(1):39–59
2. Blum A, Furst M (1995) Fast planning through planning graph analysis. In: Mellish CS (ed) Proceedings of the 14th international joint conference on artificial intelligence, IJCAI-95, vol 2. Morgan Kaufmann, Montreal, pp 1636–1642
3. Bonet B, Geffner H (2001) Planning as heuristic search. Artif Intell 129(1–2):5–33

4. Castillo L, Fernández-Olivares J, García-Pérez O, Palao F (2006) Bringing users and planning technology together. Experiences in siadex. In: Proceedings of the 16th international conference on automated planning and scheduling (ICAPS 2006). AAAI Press, Menlo Park
5. Coles A, Fox M, Smith A (2006) A new local search algorithm for forward-chaining planning. In: Proceedings of the 17th international conference on automated planning and scheduling (ICAPS-2007), Providence, RI, USA
6. Cox M, Muñoz-Avlia H, Bergmann R (2005) Case-based planning. Knowl Eng Rev 20(3):283–287
7. De la Rosa T, Jiménez S, Borrajo D (2008) Learning relational decision trees for guiding heuristic planning. In: Proceedings of the 18th international conference on automated planning and scheduling
8. De la Rosa T, Jiménez S, García-Durán R, Fernández F, García-Olaya A, Borrajo D (2009) Three relational learning approaches for lookahead heuristic planning. In: Working notes of ICAPS 2009 workshop on planning and learning, pp 37–44
9. De la Rosa T, Jiménez S, Fuentetaja R, Borrajo D (2011) Scaling up heuristic planning with relational decision trees. J Artif Intell Res 40:767–813
10. Della Penna G, Magazzeni D, Mercorio F (2012) A universal planning system for hybrid domains. Appl Intell 36:932–959
11. Fox M, Long D (1998) The automatic inference of state invariants in TIM. J Artif Intell Res 9:317–371
12. Ghallab M, Nau D, Traverso P (2004) Automated planning, theory and practice. Morgan Kaufmann, San Mateo
13. Helmert M (2006) The fast downward planning system. J Artif Intell Res 26:191–246
14. Hoffmann J (2005) Where "ignoring delete lists" works: local search topology in planning benchmarks. J Artif Intell Res 24:685–758
15. Hoffmann J, Nebel B (2001) The FF planning system: fast plan generation through heuristic search. J Artif Intell Res 14:253–302
16. Hoffmann J, Porteous J, Sebastia L (2004) Ordered landmarks in planning. J Artif Intell Res 22:215–278
17. Hogg C, Muñoz-Avila H, Kuter U (2008) HTN-MAKER: learning htns with minimal additional knowledge engineering required. In: Proceedings of the 20th AAAI conference. AAAI Press, Menlo Park
18. Kuzu M, Cicekli NK (2012) Dynamic planning approach to automated web service composition. Appl Intell 36:1–28
19. Martin M, Geffner H (2004) Learning generalized policies from planning examples using concept languages. Appl Intell 20:9–19
20. McGann C, Py F, Rajan K, Ryan H, Henthorn R (2008) Adaptative control for autonomous underwater vehicles. In: Proceedings of the 23rd AAAI conference. AAAI Press, Menlo Park
21. Nayak P, Kurien J, Dorais G, Millar W, Rajan K, Kanefsky R (1999) Validating the ds-1 remote agent experiment. In: Artificial intelligence, robotics and automation in space
22. Richter S, Westphal M (2010) The LAMA planner: guiding cost-based anytime planning with landmarks. J Artif Intell Res 39:127–177
23. Serina I (2010) Kernel functions for case-based planning. Artif Intell 174:1369–1406
24. Spalazzi L (2001) A survey on case-based planning. Artif Intell Rev 16:3–36. citeseer.ist.psu.edu/kettler94massively.html
25. Veloso M, Carbonell J (1993) Derivational analogy in PRODIGY: automating case acquisition, storage, and utilization. Mach Learn 10(3):249–278
26. Vidal V (2004) A lookahead strategy for heuristic search planning. In: Proceedings of the fourteenth international conference on automated planning and scheduling, Whistler, British Columbia, Canada, pp 150–160
27. Yoon S, Fern A, Givan R (2008) Learning control knowledge for forward search planning. J Mach Learn Res 9:683–718

Tomás de la Rosa is a Ph.D. Assistant Professor at Unversidad Carlos III de Madrid (UC3M). He received his Ph.D. in Computer Science from the UC3M in 2010. He is member of the Planning and Learning Group (PLG) at UC3M since 2005. He has published 14 journal and conference papers in the field of Automated Planning and Machine Learning. He was co-organizer of the Workshop on Planning and Learning in the International Conference of Automated Planning and Scheduling (ICAPS) in 2009. He has participated in 10 research projects funded by European Union and Spanish research council, including 4 projects for transferring knowledge to the industry.

Angel García-Olaya is Associate Professor at Unversidad Carlos III de Madrid (UC3M) since 2004. He received his Ph.D. and B.Sc. in Telecommunications from UPM in 2004 and 1999 respectively. He also has a M.Sc. on Health Informatics from University of Athens, 1997. He is author of about 40 journal and conference papers, in the fields of Telemedicine, Automated Planning and Robotics, being awarded with a prize of innovation in Telemedicine by University of Santiago de Compostela in 2004 and with the Best Paper Award in the 2011 Conference of the Spanish Association for Artificial Intelligence. He has participated in 15 research projects funded by European Union and Spanish research council.

Daniel Borrajo is a University Professor of Computer Science (CS) at Unversidad Carlos III de Madrid (UC3M) since 1998. He received his Ph.D. in CS from Universidad Politécnica de Madrid (UPM) (1990, doctoral prize), and a B.Sc. in CS (1987) also from UPM. He has been also a pre- and post-doctoral visiting researcher at the CS Department of Carnegie Mellon University (1989, 1993–1994, 2007–2008). He has been vicedean (1996–2000 and 2004–2006) and Head of the Department (2000–2002). He founded the Plannning and Learning Group (PLG) in 2004 from the AI group at UC3M (since 1995). He has published over 160 journal and conference papers mainly in the field of machine learning and problem solving (including planning). He has served as the Program co-chair of the International Conference on Automated Planning and Scheduling (ICAPS 2013), Conference co-chair of the International Symposium on Combinatorial Search (SoCS 2011 and 2012), Conference co-chair of ICAPS 2006, and Program Chair of the Spanish Conference on Artificial Intelligence. He has been the coordinator and researcher of more than 15 european, national and regional R&D projects.