Progress in Case-Based Planning

DANIEL BORRAJO, Universidad Carlos III de Madrid
ANNA ROUBÍČKOVÁ, Free University of Bozen-Bolzano
IVAN SERINA, University of Brescia

Case-Based Planning is an approach to Automated Planning that tries to save computational effort by reusing previously found solutions. In 2001, Spalazzi published a survey of work in CBP; we present here an updated overview of systems that contributed to the evolution of the field or addressed some issues related to planning by reuse in a novel way. The paper presents relevant planners, so that readers gain an insight into the operation of these systems. This analysis will allow readers to understand the approaches both in the quality of the solutions and in the complexity of finding them.

Categories and Subject Descriptors: I.2.8 [**Problem Solving, Control Methods, and Search**]: Plan execution, formation, and generation; I.2.6 [**Learning**]: Analogies; I.2 [**Artificial Intelligence**]

General Terms: Algorithms

Additional Key Words and Phrases: Case-Based Planning, Automated Planning

ACM Reference Format:
Daniel Borrajo, Anna Roubíčková and Ivan Serina, 2014. Progress in Case-Based Planning. *ACM Comput. Surv.* V, N, Article A (January YYYY), 35 pages.
DOI : http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Automated Planning deals with the task of finding ordered sets of actions that allow a system to transform an initial state to a state satisfying a goal specification [Ghallab et al. 2004]. These sets of actions are referred to as plans in the deterministic setting and policies in the non-deterministic setting. In deterministic planning (the focus of this paper), finding a plan, or even deciding its existence, has been shown to be PSPACE-complete unless severe restrictions take place [Bäckström and Nebel 1995; Bylander 1994]. Therefore, extensive research has been devoted to efficiently obtaining plans. Over time, many alternative approaches have risen to improve the efficiency of planning systems, such as domain-independent heuristics [Bonet and Geffner 2001; Hoffmann and Nebel 2001] or learning domain-dependent knowledge in various forms [Jiménez et al. 2012; Zimmerman and Kambhampati 2003]. In this paper, we focus on learning by Case-Based Reasoning (CBR) applied to Automated Planning, an approach also known as Case-Based Planning (CBP), or *planning by reuse*.

The fundamental observation in CBP is that for many real domains the types of problems that should be solved do not vary much and tend to recur. Thus, one can expect that previous solutions to similar problems will be useful when solving new problems. For example, this is true when a new problem involves very similar goals, or starts from a similar initial state to a previously solved one. This situation commonly appears due to a slight variation of goals during plan execution or due to execution failures. Then, it might be more efficient to adapt the plan in execution rather

We would like to thank Manuela Veloso for her useful comments on previous versions of this paper. This work has been partially supported by Spanish MICINN projects TIN2008-06701-C03-03 and TIN2011-27652-C03-02.
Author's addresses: Daniel Borrajo, Departamento de Informática, Universidad Carlos III de Madrid; Anna Roubíčková, Free University of Bozen-Bolzano; Ivan Serina, Department of Information Engineering, University of Brescia

DOI : http://dx.doi.org/10.1145/0000000.0000000

than to re-plan from scratch. In the extreme, one might even base the whole planning process on the modification of plans, an approach named planning from *second principles* [Nebel and Köhler 1995]. Indeed, CBP does not need to generate a plan from scratch; instead, it exploits the knowledge in plans that were generated before whenever it is advantageous. This leads to several technical as well as theoretical challenges, such as how to define and check for similarity, how to adapt a previous solution, or how to estimate efficiency and measure quality of the reuse.

A (purely) case-based system is greatly dependent on the level of reusability of the previously solved instances; this dependency is captured by two assumptions commonly made in the case-based community: problems tend to *recur* (and hence there is a chance that the additional work on retrieving, adapting and storing solutions will eventually pay off); and the world is *regular*, which means that similar problems have similar solutions (so that if one starts from a solution to a similar problem, the solution will be found with little computational effort).

Since the first works on planning, there have been many approaches in the line of CBP [Alterman 1990; Hammond 1990]. However, the theoretical research [Liberatore 2005; Nebel and Köhler 1995] revealed that CBP is not capable of improving over the generative planning approach in terms of worst-case complexity and the first enthusiasm faded. Even though these results are true in the worst-case scenario, these studies did not fully exploit the restrictions stemming from the case-based settings (such as regularity of the world and recurrence of similar problems). In some settings, the use of a case base may indeed lead to significant improvements, as it has been discussed using complexity analysis [Au et al. 2002; Kuchibatla and Muñoz-Avila 2006], or from a practical point of view [van der Krogt and de Weerdt 2005; Veloso and Carbonell 1993]. Thus, CBP offers a potential heuristic mechanism for handling intractable problems.

Our goal is to describe the progress of domain-independent CBP and some of the main alternatives explored over time. By *domain independence* we understand the possibility to use the same system in many different domains, without changing its code. Nevertheless, some of the planners require more information about the domain than others, such as those based on Hierarchical Task Networks (HTN). In this paper, we will mainly focus on domain-independent planners that take as input STRIPS domain and problem representations. For completeness, we will also cover some planners based on HTN representations and some systems that deal with other planning tasks, such as planning and execution. Spalazzi [2001] wrote a survey that provided an exhaustive taxonomy of all case-based planners up to 2001, based on the different CBP features. Complementary to his work, other surveys can be found in [Cox et al. 2005; Muñoz-Avila and Cox 2008; Veloso et al. 1996]. The difference of this paper with those surveys is that they: compare in depth only very few of the most important planners [Veloso et al. 1996]; are short papers [Cox et al. 2005]; or focus on only one aspect, such as plan adaptation [Muñoz-Avila and Cox 2008]. We describe each system as a whole and we outline how and where it fits into the taxonomy suggested by Spalazzi, as opposed to his work, whose focus was on possible implementation alternatives of the case-based steps. We also cover new planners that were built after Spalazzi's study. In the final section, we present an updated version of Spalazzi's tables with the recent works.

We identify the interaction among the different phases of the CBP process and present successful implementation choices both from theoretical and application points of view. As the planners under consideration were developed over the last twenty years, it is virtually impossible to perform an experimental comparison. Instead, we examine them from a theoretical point of view, we discuss the differences among them and show their behaviour on an example. We use a running example throughout the whole section to highlight the differences among the planners.

In the rest of this section, we provide a necessary background in Automated Planning, which is a subfield of Artificial Intelligence (AI), we describe the standard case-based methodology, which is used in some fields of AI to implement a reuse of previous experiences, and we show how it can be applied to Automated Planning. Then, we provide an example which will be used throughout the rest of the paper to show the behaviour of various CBP systems.

1.1. Planning Formalisation

The task of automated planning is commonly defined in its propositional setting by the following decision problem.

Definition 1.1. An instance of a **propositional planning task** is a tuple $\Pi = \langle \mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ where:

— $\mathcal{F}$ is a finite set of ground atomic propositional formulae;
— $\mathcal{A}$ is a finite set of actions;
— $\mathcal{I} \subseteq \mathcal{F}$ is a set of propositions that are true in the initial state; and
— $\mathcal{G} \subseteq \mathcal{F}$ is a set of literals specifying the goals.

In order to solve planning tasks, planners need to receive these instances as inputs. Given that fully defining the tasks can be quite cumbersome, the planning community has defined a standard modelling language, PDDL (Planning Domain Definition Language) [Gerevini et al. 2009]. It allows users to specify planning models as two inputs (files): a domain, that includes a set of predicates and a set of actions; and a problem, that includes the specific objects involved in the planning task, the initial state and the goals. More formally:

Definition 1.2. A **planning domain** is a tuple $\mathcal{D} = \langle P, Op \rangle$, where

— P is a finite set of predicates;
— Op is a finite set of operators, where each operator $o \in Op$ is a rule $pre(o) \Rightarrow post(o)$, where $pre(o)$ and $post(o)$ are the following sets:
  — $pre(o) \subseteq P$ are the operator's preconditions,
  — $post(o) \subseteq P$ are the operator's postconditions, or effects. They are divided in positive effects *add*(o) (become true when the operator is applied) and negative effects *del*(o) (become false when operator is applied).

Definition 1.3. A **planning problem** is a tuple $\mathcal{P} = \langle O, I, G \rangle$, where

— O is a finite set of objects;
— I is a set of propositions that are true in the initial state; and
— G is a set of literals specifying the goals.

Given a domain $\mathcal{D}$, and a problem $\mathcal{P}$, current planners automatically generate the equivalent propositional planning task, $\Pi = \langle \mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, by instantiating (grounding) the formulae in $\mathcal{D}$, similarly to the STRIPS formalization [Fikes and Nilsson 1971]. Using the objects in the problem, O, planners instantiate all predicates in P, obtaining $\mathcal{F}$. Also, using objects in O and the operators in Op, planners instantiate all operators' schemata to obtain all actions in $\mathcal{A}$. $\mathcal{I}$ is the initial state I and the goals $\mathcal{G}$ are G.

$\mathcal{I}$ is a fully specified state, as interpreted under the closed world assumption — the propositions present in $\mathcal{I}$ are the only propositions that are true in the initial state, and all other propositions of $\mathcal{F}$ are assumed to be false in the initial state. On the contrary, $\mathcal{G}$ is only a partially specified state, where propositions from $\mathcal{G}$ have the truth-value as specified while the other propositions have an arbitrary truth-value.

An action $a \in \mathcal{A}$ can be applied in a state s only if its preconditions $pre(a)$ are met, that is, if $pre(a) \subseteq s$. The application of a in s yields a state s' where $post(a)$ hold and the rest of the propositions remain the same as in s, formally, $s' = apply(a, s) = s \setminus del(a) \cup add(a)$.

After receiving a planning problem as input, planners generate a sequence of actions $\pi = (a_1, \ldots, a_n)$, called a plan, as output. π solves the planning task Π if the iterative application of the actions in the plan transforms the initial state into a state where the goals are true. So, there is a decision problem, PLANSAT, whose goal is to check whether there exists a sequence of actions that, when applied to the initial state $\mathcal{I}$, yields a state that satisfies the goals $\mathcal{G}$. The related optimisation problem is formalised as PLANMIN and asks whether such a sequence consists of at most k actions.

If $n \leq k$, a solution plan is also a positive instance of PLANMIN. Both PLANSAT and PLANMIN have been shown to be PSPACE-complete problems [Bylander 1994] in their general form.

1.2. Case-Based Methodology

CBR is a field of AI that uses past experiences to solve new problems, assuming that similar problems have similar solutions [Aamodt and Plaza 1994; Leake 1996; de Mántaras 2001]. CBR systems have at their disposal a set $\mathcal{L}$ of previously solved problems constituting the experience of the system, called *case base*. Every *case* is a pair consisting of a problem description and some problem solving knowledge that could help solve future similar problems. The problem description comprises any knowledge (e.g. features) that is relevant for defining a task in any representation paradigm (though all the surveyed systems here use predicate logic). The problem solving knowledge can take the form of the solution to the problem or it can contain some reasoning information, e.g., about how the solution has been found previously, or why the solution has some particular shape.

CBP is a type of CBR, which uses the stored experiences to lower the high complexity of solving the planning tasks [Cox et al. 2005; Hammond 1990]. A case in this setting is composed of a problem description (referred to as *problem* and denoted by Π) and reuse information (denoted by $\wp$). Usually, the reuse information consists of the problem solution, but some systems use other information as well.

In order to benefit from storing and reusing past solutions, a case-based system needs to efficiently implement several steps organised in the so-called case-based procedure (see Fig. 1, where the updated case base – $\mathcal{L}$ on the right – is the input to new CBR episodes). Here, we describe the objective of each of its phases and briefly outline different implementation choices that suit planning applications. The resulting categorisation was introduced by Spalazzi [2001] and is repeated here, because we use it later in the planners' descriptions.

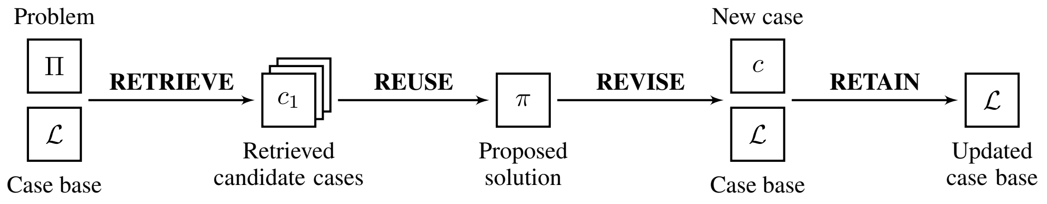

Fig. 1. The case-based procedure.

1.2.1. CBR Procedure. In this section, we will cover the main steps of the CBR process: retrieve, reuse, revise, and retain. When a case-based system faces a new problem, it queries the case base to find analogous cases (*Retrieve*). There are several ways to identify such cases: *associative* retrieval uses the features of the cases to assess the suitability of a case; e.g., by using a similarity metric [de la Rosa et al. 2013; Hanks and Weld 1995]. *Hierarchical* retrieval uses a hierarchical ordering of the features, proceeding from the more general to the more specific ones; e.g., the indexing-based retrieval [Serina 2010; Tonidandel and Rillo 2002; Veloso 1994]. Goals, initial states and/or encountered failures are the most frequent features used for retrieval purposes. Domain-dependent planners may implement *model-based* retrieval, where the domain-dependent knowledge may be used to generalize the current problem to find a better match among the cases. All these techniques can also be combined using *hybrid* retrieval. In planning applications, a key issue is objects' identification. It is highly probable that cases, even from the same domain, are grounded over different sets of objects, or that the grounding was implemented differently, resulting in different naming conventions in different cases. Thus, the first step of the retrieval phase often consists of fitting the objects of the two problem descriptions to each other as closely as possible. Depending on the implementation, the task may be realised either as unification [Hanks and Weld 1995; Kambhampati and Hendler 1992; Köhler 1996], or as object matching [de la Rosa et al. 2013; Nebel and Köhler 1995; Serina 2010].

The objective of the *Reuse* step is to repair faults of the retrieved solution when applied to the current problem. As this step may modify the previous solution, it is also referred to as *adaptation*. Different approaches have been considered in the literature for plan adaptation. Spalazzi [2001] identifies the following categories. The first group of approaches uses *non automated adaptation*; the system simply copies the stored solution, or lets the user adapt it manually. The systems that attempt to automatically produce applicable solution plans can be divided into those that perform transformational, derivational or hybrid adaptation. *Transformational* adaptation attempts to repair the retrieved plan by means of constraint satisfaction, heuristics, plan generation, merging or recursive case-based approaches. Some of these systems attempt to reuse the pieces of the previous plan that are still valid and modify, remove, or add new actions for the previous actions that fail in the new conditions [Hammond 1990; Hanks and Weld 1995; Kambhampati and Hendler 1992]. Others perform more dynamic plan changes using a deterministic [Likhachev and Koenig 2005; van der Krogt and de Weerdt 2005] or stochastic [Borrajo and Veloso 2012; Fox et al. 2006] sequence of plan modification operators that try to repair the failed plan. Alternatively, the system can try to replay decisions that were successful in the search for the solution found in the case base, yielding a *derivational* adaptation [de la Rosa et al. 2013; Veloso 1994]. Another alternative consists of using a *hybrid* approach that combines several of the above mentioned techniques.

The revision step (*Revise*) can be divided into two subtasks: *verification* of the adapted solution and its possible *repair*. The verification step checks for the failures that may occur during plan execution, preventing the plan from producing the expected result. It may be addressed by simulated execution, formal verification or by the execution itself, depending on what the system's repair procedures support. Consequently, Spalazzi [2001] distinguishes systems verifying the plans *in the real world* [Kambhampati and Hendler 1992], *in the model* [Veloso 1994] or *in the case base*, where the last approach searches for failures of similar problems recorded in the case base. In addition, the solution may also be evaluated externally, *by a teacher*. The teacher is a human expert often needed in domains where the correctness of the plan is crucial, such as medical diagnoses and therapy. In such settings, the automatically found solution may need to be approved by a responsible person. When a failure is discovered, the system may simply abort the plan, or attempt a repair. The repair can be then performed either *by a user* [Veloso 1994], or by the system itself (*self-repair*) [Kambhampati and Hendler 1992].

The retention phase (*Retain*) is responsible for building and maintaining the case base so that the system can retrieve cases more efficiently and the retrieved cases are of higher quality. Depending on the criteria that guide this phase, in [Leake and Wilson 1998] its authors distinguish two types of case-base maintenance techniques — the quantitative criteria (e.g., time) lead to performance-driven policies, while the qualitative criteria (e.g. coverage) lead to competence-driven policies.

In planning, the retention scheme is usually one of the extreme ones: either all solutions are included in the case base; or a pre-built case base, which is maintained fixed during the lifetime of the system, is used and the new solutions are discarded. Accordingly, Spalazzi [2001] presents only one selection strategy for deciding which solutions to keep and which to discard; it is decided *by the user*. If the new solution is to be stored, the relevant data is extracted first, but the implementation of the extraction depends on the implementation of the adaptation and retrieval, as well as on the organisation of the case base. The extracted data (that is, the new case) is then inserted into the case base in a memory-efficient manner, using either *no arrangement*, simply accumulating one case after another, or arranging the cases by *metric* information, adjusting the *index*, or leaving the task to the user to re-arrange the case base *manually*.

Muñoz-Avila studied the case retention problem in order to filter redundant cases [Muñoz-Avila 2001]. His approach was based on the observed computational effort when solving the new problem. In [Gerevini et al. 2013a] the authors formally differentiated between the *online* and *offline* approaches to the case-base maintenance problem and proposed different policies to perform the maintenance that are guided by different qualitative and quantitative criteria.

1.2.2. Complexity of CBP. In this section, we will shortly present some complexity results on CBP. The complexity of propositional planning has been widely studied in the literature. In its most general form, it was proven to be PSPACE-complete [Bylander 1994]. If severe constraints on the form of the operators are fulfilled, one can guarantee polynomial time or at least NP-completeness [Bäckström and Nebel 1995; Bäckström et al. 2012; Bylander 1994]. Possible approaches to address the high complexity of planning involve developing heuristics and learning domain-dependent knowledge from experience. CBP is one of such heuristic approaches. Regardless of the precise underlying formalisation, there are two main approaches to CBP, whose fundamental difference is in the way they adapt the stored plan to solve the current problem: conservative and generative plan adaptation.

Conservative plan adaptation tries to reuse as much of the known plan as possible [Kambhampati and Hendler 1992]. It turns out that such adaptation may be very expensive given that the identification of the "maximal reusable part" is a source of additional hardness. It may make the plan adaptation even more expensive than the traditional plan generation [Nebel and Köhler 1995] for some fragments of planning languages. Moreover, the quality of the solution strongly depends on the correspondence between the case being reused and the current problem, which is influenced by the way the case is retrieved from the case base as well as by the case base itself, or rather by its competence.

Generative plan reuse treats the case as a hint that can, in the extreme, be fully ignored and the solution can be generated from scratch [Serina 2010; Tonidandel and Rillo 2002; Veloso 1994]. Indeed, the results obtained with extensions of Universal Classical Planning (UCP [Kambhampati and Srivastava 1996]) confirm that non-conservative plan reuse is at most as hard as classical planning. Using DERUCP, an algorithm which models derivational analogy (e.g., as used by PRODIGY/ANALOGY in Sec. 5) for STRIPS-style planners, Au *et al.* [2002] show that the search space produced by derivational adaptation can be exponentially smaller compared to the search space required by transformational adaptation. TRANSUCP [Kuchibatla and Muñoz-Avila 2006] formally models transformational analogy and shows that such a modification of the UCP algorithm does not require conservativeness of the reuse. Hence, it does not deteriorate its worst-case complexity.

De Haan *et al.* [2013] have noted that the framework of the worst-case complexity is not particularly well suited for studying plan reuse in the CBP settings, where the system is very likely to have at its disposal a solution plan that is similar to the plan it is looking for. In the worst case, of course, even the most similar plan is not useful at all [Liberatore 2005]. Therefore, they use the framework of parametrized complexity, where the parameter captures different notions of plan similarity. Plan reuse in their work assumes that the whole stored plan is used and the system tries to find only a prefix and a suffix of actions to connect the plan with the new initial state and goals. The results show that such a plan reuse is provably tractable only if the number of actions used in the prefix and suffix of the new plan is low and the actions are selected from a (relatively) small set.

1.3. Running Example

The example used in this paper is from the Depots domain introduced in the 3rd International Planning Competition [Long and Fox 2003]. This domain can be considered a combination of two other well-known planning domains: the Transportation (or Logistics) domain and the Blocksworld domain. In Depots, trucks transfer crates of goods among several locations. Similarly to blocks in Blocksworld, the crates need to be lifted in the source location and loaded onto a truck, and after the truck arrives to its destination, it needs to be unloaded and the crates dropped onto other crates or onto pallets which efficiently serve as a Blocksworld table with limited space. The crates are moved by hoists present at each location, where the hoists play the same role as the robotic arm in Blocksworld, for loading and unloading objects to and from the trucks.

The Depots domain is *typed*, which means that not all objects can be interchanged because they are of different types. In Depots, there are types such as `Truck` or `Pallet`, but the domain can be converted to an untyped one by replacing the type statements by propositions. For instance, in the PDDL description, we could replace (`:objects truck0 truck1 – Truck`) by propositions (`is-truck truck0`), (`is-truck truck1`). This simple conversion allows us to use the same example also for the systems that do not allow typed objects.

Figure 2 depicts a planning problem in the Depots domain with the corresponding PDDL description on Figure 3a. A solution plan to this problem, consisting of loading the crates on the trucks and driving the trucks to the crates destination, where they are unloaded, can be found on Figure 4. In the remainder of this section, we assume that the planner has encountered this problem in the past and created a corresponding case, which is now stored in the case base. The new problem that the planner is trying to solve is described on Figure 3b. The main difference with respect to the previous one is in the presence of `crate2` on top of `crate1` in `depot0`, which should be also transported to `distributor0` and placed again on top of `crate1`.

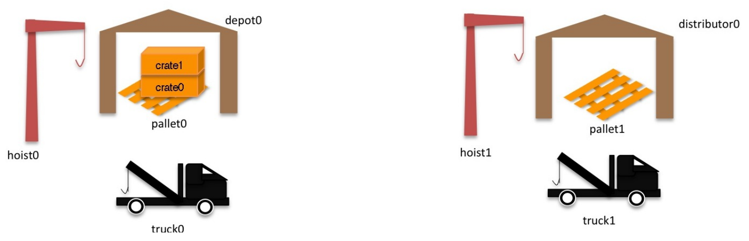

Fig. 2. Example of a problem in the Depots domain, where the initial state is captured on the picture and the goal is to move both crates to distributor0 maintaining the same stack order.

```
(define (problem case1) (:domain Depot)
     (:objects  depot0 - Depot
                distributor0 - Distributor
                truck0 truck1 - Truck
                pallet0 pallet1 - Pallet
                crate0 crate1 - Crate
                hoist0 hoist1 - Hoist)
     (:init     (at pallet0 depot0)
                (at pallet1 distributor0)
                (clear pallet1)
                (at truck0 depot0)
                (at truck1 distributor0)
                (at hoist0 depot0)
                (available hoist0)
                (at hoist1 distributor0)
                (available hoist1)
                (at crate0 depot0)
                (on crate0 pallet0)
                (at crate1 depot0)
                (on crate1 crate0)
                (clear crate1))



     (:goal     (and  (on crate0 pallet1)
                      (on crate1 crate0))))
```

```
(define (problem problem1) (:domain Depot)
     (:objects  depot0 - Depot
                distributor0 - Distributor
                truck0 truck1 - Truck
                pallet0 pallet1 - Pallet
                crate0 crate1 crate2 - Crate
                hoist0 hoist1 - Hoist)
     (:init     (at pallet0 depot0)
                (at pallet1 distributor0)
                (clear pallet1)
                (at truck0 depot0)
                (at truck1 distributor0)
                (at hoist0 depot0)
                (available hoist0)
                (at hoist1 distributor0)
                (available hoist1)
                (at crate0 depot0)
                (on crate0 pallet0)
                (at crate1 depot0)
                (on crate1 crate0)
                (at crate2 depot0)
                (on crate2 crate1)
                (clear crate2))
     (:goal     (and  (on crate0 pallet1)
                      (on crate1 crate0)
                      (on crate2 crate1))))
```

Fig. 3. PDDL description of a) the problem in Figure 2 on the left and b) a new problem on the right.

In the following sections, we describe a variety of CBP systems that have contributed to the evolution of CBP. We will not necessarily follow the chronological order. Instead, we start by describing some of the simpler systems to allow the reader to get accustomed to the case-based process. Then,

```
(lift hoist0 crate1 crate0 depot0)
(load hoist0 crate1 truck0 depot0)
(lift hoist0 crate0 pallet0 depot0)
(load hoist0 crate0 truck0 depot0)
(drive truck0 depot0 distributor0)
(unload hoist1 crate0 truck0 distributor0)
(drop hoist1 crate0 pallet1 distributor0)
(unload hoist1 crate1 truck0 distributor0)
(drop hoist1 crate1 crate0 distributor0)
```

Fig. 4. Solution plan for the problem in Figure 2.

we gradually move to more evolved systems and to systems that implement some of the steps of the case-based methodology in more elaborate ways, explaining the differences with respect to the systems described before. For each system, we present the case representation, followed by all (or part of the) four CBR steps (Retrieve, Reuse, Revise and Retain) and a discussion. To conclude the paper, we summarise the main differences among the surveyed systems and the novelty they introduced, we present several successful domain-dependent applications of CBP and discuss future directions and open problems for CBP.

2. PRIAR

PRIAR [Kambhampati and Hendler 1992] is not a self-contained case-based planner. Rather it is a case-based extension of the planner NONLIN [Tate 1977], that is based on hierarchical tasks, such as HTN [Erol et al. 1994]. HTN planners require richer domain models that involve a hierarchy of tasks and methods to achieve those subtasks.

PRIAR does not address all the steps usually present in CBP. In particular, the revision and retention phases were not fully implemented. Its focus was on the retrieval and reuse phases, that are addressed by a domain-independent algorithm. In the retrieval phase, the matching and evaluation use a similarity metric based on the estimated cost of the modification of a case. PRIAR does not feature any revision procedure; the proposed solution complies with the known domain theory given the repair strategies are correct. When execution fails and other unforeseen complications are observed, PRIAR invokes a special *replanning* procedure which puts an extra emphasis on reusing the already executed part of the plan; in fact, the initial state of the new problem is the current state of the execution instead of the former initial state.

2.1. Case representation

PRIAR extends NONLIN, which is a hierarchical planner. This fact is decisive for the choice of the case and naturally also for the design of the adaptation algorithm. While generating a plan, NONLIN also produces a *validation structure*. It is a graph structure linking the tasks' decomposition, the initial state and facts forming preconditions and effects of actions. All this information forms the plan's explanation that PRIAR stores in its cases.

Each validation links together two nodes, called *source* and *destination*, of an HTN used by NONLIN. The *effect* of the source satisfies the *applicability condition* of the destination. The applicability condition is then called *supported condition* and the effect is called *supporting effect* of the validation. Such a structure provides an explanation of each proposition in the plan, marking it as either a precondition of some actions in the plan, or as a goal of the plan.

For example, a validation $\langle$(on crate1 crate0), n_I, (lifting hoist0 crate1), $n_{L_1}\rangle$ is part of the validation structure of case1. This validation links the node of the initial state n_I, where (on crate1 crate0) is true and forms one of the preconditions of the action lift represented by node n_{L_1}. The node n_{L_1} reduces the high-level task of moving crate1 from crate0 to truck0. In the validation structure, n_{L_1} is followed by nodes that continue the reduction by loading crate1 to truck0. There are other validations linked to the node n_{L_1} to capture the rest of the preconditions of the action lift, such as a validation to ensure the availability of the hoist and the fact that the lifted crate is clear.

2.2. Retrieval

As many other case-based planners, PRIAR tries to map the objects of the stored problem (case) with objects of the current problem. As the problem is represented by a set of propositions, the mapping is realised as unification. The first step, which filters out a number of unsuitable cases, is performed by a partial unification of the objects of the stored and current problems' goal descriptions, which also provides a mapping between the case and the current problem. The second step realises a finer selection that exploits the validation structure of the case, which is *interpreted* according to the previously obtained mapping. The interpretation of the validation structure consists of renaming the objects as specified by the mapping. The interpreted validation structure can be used to address the current problem, as the objects of the structure and the current problem correspond to each other. However, just a simple renaming of objects does not ensure applicability of the validation structure in the new settings. Hence, the number and type of inconsistencies in the interpreted validation structure are estimated together with the estimated difficulty of their repair, as it is explained later. Consequently, the system can discard cases that achieve similar goals by actions that cannot be applied in the current situation. Such an approach ensures that the planner reuses a significant part of the validation structure, hence the plan modification is claimed to be *conservative*.[1] Also all the modifications are *correct*, as they never introduce a new inconsistency into the validation structure.

During the partial unification of problems' descriptions, some initial facts get annotated as *out facts* and *new facts*, while some goal facts may get annotated as *extra* or *unnecessary goals* depending on whether they are missing in the case or in the current problem. All these annotations refer to inconsistencies in the validation structure and are addressed by different repair strategies of different computational costs. For example, an *unnecessary goal* indicates a presence of an *unnecessary validation*, which can be pruned in order to simplify the validation structure. The corresponding repair strategy is to remove the validation, which is quite a simple procedure. On the other hand, some *failing validations*, which are caused by *out facts*, that is, facts that are present in the stored problem but not in the current problem, may be very difficult to repair as they imply undoing some of the planner's decisions and deviating the search for a solution plan into a different direction.

For example, the initial facts related to `crate2` in Figure 3b are *new facts* and are added into the initial state of the case described by the PDDL on Figure 3a, while (`clear crate1`) is an *out fact* which is not satisfied in the new problem and needs to be removed from the description. Note that in our example, the goals of the new problem extend the goals of the case and hence the case does not contain any *unnecessary goals*.[2] Inconsistencies of the example consist of the *new facts* (`at crate2 depot0`), (`on crate2 crate1`) and (`clear crate2`), an *out fact* (`clear crate1`), and an *extra goal* (`on crate2 crate1`). There are no *unnecessary goals*.

Using the "number and type of inconsistencies in the validation structure", PRIAR heuristically assesses the costs of repairing the validation structure. The price of the repair can be understood as a similarity metric, which can also be used in the evaluation phase. While it is common that the reuse algorithm determines the retrieval strategy, in the case of PRIAR one part of the retrieval effort (namely the mapping) is used later in the reuse phase, too. One could even view the mapping as the first step of the reuse, because it marks the inconsistencies on the case's validation structure. The adaptation engine of PRIAR directly resolves the inconsistencies without the need of first inspecting the validation structure and its consistency.

2.3. Reuse

The adaptation phase of PRIAR is realised in two steps. First, the necessary repair actions (called *refit tasks*) are suggested, as described above. Even though the effort and the number of actions needed to repair an inconsistency may be big, the suggested refit tasks are simple because they are

[1][Nebel and Köhler 1995] expose that PRIAR does not identify the *maximal* reusable part of the stored solution, hence the reuse is only *quasi-conservative*.

[2]An example of all the annotations and their handling in the Blocksworld domain can be found in the original paper [Kambhampati and Hendler 1992].

expressed on quite a high level (e.g. *achieve[fact], remove[validation]*). The result of this step is a consistent validation structure. However, the validation structure cannot be applied yet because the implementation of the refit tasks is missing. In other words, the partially reduced validation structure needs to be fully reduced. For that purpose PRIAR uses NONLIN, where the consistent, partially reduced validation structure is treated as an intermediate step of generative planning and is solved as a sub-problem by the hierarchical planner.

In our example, the system suggests to address the extra goal by adding a refit-task `achieve`[`on crate2 crate1`] into the plan retrieved from the case. The generative planner is then called to substitute each such refit-task by a sequence of actions, producing a complete plan.

If needed, NONLIN can backtrack over the whole validation structure and find an unrelated solution plan which ensures that the "satisficing" performance is not affected, i.e., NONLIN with the addition of PRIAR can solve the same set of problems as NONLIN itself. While the completeness of the planner remains untouched, the authors claim that in the worst case using PRIAR is as efficient as using pure NONLIN. They however discuss the difficulty of stating an average improvement in the "general case". Under several assumptions they claim even exponential reduction of the solution search space though.

The main result concerning complexity states that the time needed to obtain a consistent validation structure by means of refitting tasks is $O(n^4)$, where n is the length of the plan. This is due to the fact that the number of validations is linear to the size of the case (which contains the validation structure here), but the repair of a failing validation can require up to $O(n^3)$ because it may involve detection of interactions of new validations that appeared during the repair of other missing validations [Chapman 1987].

Note that even though the authors do not state it explicitly, such a result implies the refitting to be polynomial also in the length of the problem description because the case base (and consequently every solution stored within) is assumed to be of polynomial size with respect to the problem's description. However, the resulting validation structure is only partially reduced and needs to be reduced fully in order to be applied to the problem. Reducing such a structure should be less expensive than generating a plan (and its explanation) from scratch according to the authors. Admittedly, in some scenarios the generative planner may produce a solution that is better than the one obtained by modification. However, the use of the solution produced by PRIAR may reduce the amount of used resources with respect to NONLIN, especially when replanning.

2.4. Discussion

Besides being one of the first case-based planners, the main contribution of PRIAR is the retrieval phase. It offers a heuristic strategy that, besides matching features of the cases and the current problem, estimates the cost of the adaptation and uses it as a selection criterion to make the retrieval more informed. Hence we can say that the retrieval is not based entirely on the problem similarity, but it is guided by the estimated reuse effort as well, making the system less dependent on the regularity of a domain it is applied in. Note that the hierarchical approach (the full reduction of the partially reduced validation structure), involves the use of more domain-dependent knowledge than other case-based planners, even though the algorithms are domain independent.

3. SPA

After PRIAR, Hanks and Weld introduced the *S*ystematic *P*lan *A*daptor, SPA [Hanks and Weld 1995]. Their main aim was to provide a simple framework to study plan modification and therefore their work does not address the retrieval- and storage- related problems in any great depth.

SPA uses the SNLP planner [McAllester and Rosenblitt 1991], which addresses the planning task by a search in a space of partial plans. A search for a solution to a given problem can be pictured as a tree whose nodes correspond to partial plans; every child node represents a plan that solves one flaw of its parent. The root contains an empty plan for the instance; that is, the description of the initial state and goals set. And the leaves correspond either to a solution (all the flaws are solved) or to a failure (all possible refinements are exhausted). If a sufficiently similar case is found in the case base, SNLP starts searching from a node corresponding to the retrieved plan applied to the current problem instead of searching from the node representing an empty plan.

3.1. Case representation

A case in SPA consists of the problem description and a complete consistent solution plan. As SPA builds the solution from a partial plan, the stored information contains also causal links and constraints explaining the ordering of the actions. Hence the plan consists of a set of steps, constraints and links. The *steps* are unique appearances of actions and are denoted by S_i. The causal *links* record the interaction between steps. Every link has a form $S_i \longrightarrow^Q S_j$ and denotes that step S_i achieves the expression Q, which is a precondition of step S_j. The *constraints* restrict either the order of steps ($S_i < S_j$) or a binding of variables ($(= v_i\, v_j), (\neq v_i\, v_j)$). Furthermore, each constraint records the reason of introducing the restriction: addition of a step to the plan; addition of a link among the plan's steps; or resolution of a threat to an ordering constraint by the negative effects of another step.

3.2. Retrieval

The retrieval of a case for SPA is a two-steps process — first, a suitable case is found in the case base, which is then slightly modified to correspond better to the current problem. At first the system partially unifies the goal descriptions of the current problem and the stored cases. The main selection criterion is the number of matching goals. There may be several cases (or one case under several mappings) matching the same number of goals. Then, the similarity of the initial states is taken into account. The similarity of the initial states corresponds to the estimated planning work needed to convert the current initial state to the case's initial state and is measured by the number of *open preconditions* (see the explanation below) which appear in the stored solution after replacing the initial state with the current initial state. Naturally, the case with the smallest number of open preconditions is selected. Any remaining ties are broken arbitrarily.

When unifying the goal descriptions, SPA tries to rename constants of the problems under consideration. The authors however note that finding the best matching takes time exponential in the number of the objects in the description of the case and therefore may cause the whole planning procedure to be very inefficient. For the exprimental purposes they use a linear time domain-dependent heuristic [Hanks and Weld 1995].

The selected case is then adjusted to match the current problem as closely as possible. First, the previously found mapping is applied to the case and the case's goals are replaced by the goals of the current problem. The "extra goals" introduce flaws in the plan (called *open preconditions*) while the causal links connecting "removed goals" are deleted. Then the description of the case's initial state is replaced by the current initial state and more open preconditions are introduced for the facts removed from the case's initial state whose effects were needed for some causal links of the solution. The result is a *fitted* plan, which is consistent, but probably incomplete. The adjustment step prunes very little, because the relevance of the constraints and links is not clear until the adaptation is attempted. However, the conservative pruning strategy (i.e., removal of as many constraints as possible) is crucial for ensuring completeness of the algorithm.

When SPA retrieves the case corresponding to the problem described in Figure 3a to solve the problem described on Figure 3b, it marks the goal (`on crate2 crate1`) as an extra goal, which introduces an open precondition to the plan presented on Figure 4. Then, the description of the case's initial state (Figure 3a) is replaced by the current initial state (Figure 3b) and the system identifies the facts that were removed, such as (`clear crate1`). Here, the removed fact introduces an open precondition, as the fact (`clear crate1`) is needed for (`lift hoist0 crate1 crate0 depot0`), the first action of the retrieved plan. The retrieval results in a fitted plan, which starts from the current initial state, achieves the case's goals and is aware of the incompleteness in the case's plan.

3.3. Reuse

The adaptation method is based on a constraint-posting technique of Chapman [1987] and its later refinement [McAllester and Rosenblitt 1991]. The fitted plan to be adapted constitutes a node in the search space and the search starts from there instead of traversing the space of partial-order plans in breadth-first manner starting from the root, as SNLP does.

The plan is modified by two high-level procedures as in standard partial-order planners — the flaws are repaired by a *refine* procedure and backtracking is realised by a *retract* procedure. The refine procedure selects a flaw in a plan and corrects it; depending on the kind of the flaw, different repair actions are taken. The *open precondition* flaw is caused by an action's precondition which has no causal support and is addressed by the *AddStep* repair procedure, adding an action whose effect is the precondition and an appropriate causal link is established. The *threatened link* is a flaw caused by ordering actions so that an action and its desired effect (needed for another action) may be interleaved with another action, a threat, which cancels the desired effect. Such flaw is repaired by introducing an ordering preventing the threat to interleave the action and its effect (by promoting or demoting the threat) or by forcing the objects of the threat not to unify with the objects of the desired effect. The information about which flaw was repaired, how and at which point of the search is stored in special data structures to allow the search to be systematic; that is, to consider every possible partial plan exactly once.

The fitted plan in the example produced by the retrieval step contains flaws that need to be repaired. The problems are related to the extra goal (`on crate2 crate1`) and the removed initial fact (`clear crate1`). They form open preconditions and are addressed by a refine procedure. For example, the flaw related to the extra goal is corrected by adding a step to the fitted plan so that the newly introduced action introduces as its effect the open precondition. This can be achieved by adding either (`drop hoist0 crate2 crate1 depot0`), (`drop hoist1 crate2 crate1 depot0`), (`drop hoist0 crate2 crate1 distributor0`) or (`drop hoist1 crate2 crate1 distributor0`) to the plan. The first option leads to a goal loop and is not further explored. The next two actions achieve (`on crate2 crate1`), but cannot be used, because `hoist1` is not at `depot0` and `hoist0` is not at `distributor0`, hence the actions' preconditions will never be satisfied. So, the action to drop the crate at `distributor0` by `hoist1` is added, which then forces also the use of the `unload` action to ensure that the hoist has the crate to drop, of the `load` action that places the crate on the correct truck and so on, until `crate2` is lifted from `crate1` in `depot0`, which, as a side-effect also resolves the remaining open precondition by making (`clear crate1`) true.

Based on the properties of SNLP, the adaptation algorithm is proved to be complete, which means that it will eventually find a solution if one exists. This property is guaranteed by the capability of the planner to backtrack over any link or constraint in the explanation of the fitted plan, consequently making it possible to reach the root (empty plan) and explore any other part of the search tree. Therefore any decision in the solution needs to be retractable. SPA achieves that by introducing refine and retract procedures modifying the plan to be inverse to each other. Note however that the retraction procedure guarantees completeness only if the plan is fitted conservatively (i.e., there are no decisions over which the planner could backtrack) or if the solution was obtained by a previous run of SPA and hence all the decisions are introduced by the refine procedure and therefore retractable.

3.4. Retention

The authors assume that a case base is provided to the planner beforehand and is not modified over time. The only related remarks state that, due to the need of specific supporting structures (as the explanations), it is easy to store solutions generated by SPA but very hard to include solutions obtained by other means.

A small but important remark states that theoretically it is worth to adapt plans that contain at most 40% of inappropriate actions; otherwise, the generative approach is more efficient [Hanks and

Weld 1995]. This estimation is based on the size of the search space of the transformational approach compared to the one of the generative approach. CBP reuse can be either *transformational*, where the actual solution plan is reused, or *derivational*, where the system reuses the reasoning about the solution.

3.5. Discussion

Both, PRIAR and SPA, are very similar in that the retrieval and the consecutive matching of the problem instances is guided by the same principle, and both present sound and complete adaptation algorithms. But, the information stored in an element of the case base of PRIAR is significantly more complex and so is the procedure to remove a flaw. SPA uses plan-space search, whereas PRIAR uses the hierarchical approach, moving through several levels of plan reduction. Nevertheless PRIAR seems to outperform SPA on more complex tasks, when only the reuse phase is considered, which is probably due to including more information on its domain models. Moreover, SPA searches the plan-space in breadth-first manner while the authors of PRIAR did not offer any study of the systematicity of the search and the selection of flaws to repair is random. However, the impact of systematicity on the planner's performance is doubtful anyway — on one hand it guarantees that the planner does not reconsider the same plans several times, but it also prevents the system from quick convergence to the right part of the search space.

4. MRL

MRL (Modification and Reuse in Logic) [Köhler 1996] is a case-based extension of a deductive planning system PHI [Bauer et al. 1993]. PHI generates plans as constructive proofs of plan specifications in the temporal logic LLP. Consequently, a plan is a logical formula in LLP. When searching for a solution, PHI uses proof tactics that help it prune the search space very efficiently and guide the planning process in a strictly goal-oriented way. MRL is based on a logical formalisation of the plan reuse process.

4.1. Case representation

LLP is an interval-based modal temporal logic, featuring the modal operators *next, sometimes, always* and a binary modal operator *chop* to sequentially compose formulae. Several control structures are available, too — iterations, conditionals, as well as local variables. As mentioned before, a plan is an LLP formula, denoted $Plan$, and a plan specification is a special form of a formula, $[preconditions \wedge Plan] \rightarrow goals$ meaning that if $Plan$ is carried out in a state where $preconditions$ hold, $goals$ will be achieved. PHI generates a plan by building a proof of the plan specification during which the $Plan$ variable is replaced with a formula satisfying the specification. To build a case, the plan is enriched by an index used to determine the case's position in the case base. The case base is indexed and organised hierarchically.

In the simple example we use here, the formula $Plan$ is a sequence of implications ($pre \Rightarrow post$) that represents the actions listed on Figure 4. The formula $preconditions$ is a conjunction of initial facts relevant for the plan, which, in our example, are all the propositions of the initial state on Figure 3a. The $goals$ is a formula $(\mathtt{on}\,\mathtt{crate0}\,\mathtt{pallet1}) \wedge (\mathtt{on}\,\mathtt{crate1}\,\mathtt{crate0})$. Note however that MRL is significantly more expressive than this — the presence of modal operators brings a possibility to define also intermediate goals, or a sequence of goals to happen in a specified order.

4.2. Retrieval

As the case base is indexed, the retrieval process can search it very efficiently. The index to be used for searching the case base is obtained by mapping the LLP formula of the current problem to concept descriptions in a terminological logic.

In their work [Nebel and Köhler 1995], the authors devoted quite a significant effort to study theoretical issues underlying object matching, which is an integral part of retrieval. Here, we just briefly summarise that the selection of the most suitable case is guided by a similarity function $simil$ that counts the number of matching goals between the case and the current problem and the problem of selecting the optimal matching reduces to the sub-graph isomorphism problem, which is known to be NP-hard.

4.3. Reuse

The plan modification is a two-steps process: plan interpretation followed by refitting. Plan interpretation is done deductively by proving relations between initial and goal states of the case and the current problem, resulting either in a proof that the plan stored in the case can be reused without further modifications, or a failed proof requiring refitting of the plan. Refitting starts with constructing a *plan skeleton* from the case. Then, it is extended to a correct plan by building a proof of the plan specification formula $Plan$ which is instantiated by the skeleton formula.

During the matching step, the system tries to prove that the case is applicable in the current initial state ($pre_{current} \rightarrow pre_{case}$) and that the case achieves at least all of the current goals ($goal_{case} \rightarrow goal_{current}$). If such match is proven, the necessary substitution information is extracted from the proof and applied to the case, which then solves the current problem. Then the applicability of the plan follows directly from the proof produced during matching and no further verification is needed.

If the proof of applicability fails, the plan stored in the case is refitted, starting with the extraction of the information from the failed proof. Based on the extracted information, MRL flexibly removes and adds actions to the plan, but it performs no reordering operations to avoid expensive computations of all possible permutations of a given actions' sequence. If an action occurs in a wrong position, it is deleted from the plan and subsequently re-introduced during the refitting process. This refitting strategy is referred to as *plan instantiation* and leads to remarkable efficiency gains in comparison to plan generation, especially if the plan to refit is very complex.

In the running example, the proof of applicability fails, because $goal_{case} \nrightarrow goal_{current}$ as the case described on Figure 3a does not imply the goals related to `crate2` present in the current problem (Figure 3b). Therefore MRL proceeds to refit the plan, in this situation simply by adding the actions related to loading and transporting `crate2` to the desired destination. However, if in the current problem the `crate0` and `crate1` were swapped in the initial state, the planner would remove all the actions related to loading `crate1` and insert them again after the actions loading `crate0`, instead of just reordering the initial part of the stored plan (Figure 4).

Differently from previous case-based planners, which address total- or partial-order plans, MRL can reuse also conditional plans (generated by case analysis) and iterative plans (generated by inductive proofs). MRL ensures that modified plans are correct by performing plan refitting deductively as an interleaved process of plan verification and plan generation. Plan verification is grounded on a replay of the proof process. However, the refitting of the plan's control structure may become very expensive, e.g., if an operator is inserted into a plan inside a while-loop, the inductive proof that has led to this while-loop must be replayed in order to verify the correctness of the refitted plan.

The modification of sequential plans in MRL is usually more efficient than the generation from scratch. Only minimal effort has to be spent on the verification of sequential control structures and the savings from the reuse of basic actions compensate the effort for plan matching and skeleton plan construction. This result also holds for the whole reuse process including the retrieval effort. The modification of iterative plans is always more expensive than planning from scratch because plan refitting is nearly as expensive as planning from scratch, and furthermore plan matching and skeleton plan construction require costly inference processes dealing with universally quantified goals and iterative control structures. Besides restricting the refitting strategy to plan instantiation (as described in the previous paragraph), there is also another possibility in their work to ensure efficiency gains for the modification of complex plans. Also the admissible refitting operations can be restricted so that only some parts of the proofs have to be replayed in order to guarantee the correctness of the modified plan.

4.4. Retention

Once the current problem is solved by modifying a case, the system creates a new case out of it. First, the system extracts the resulting plan and the information from the proof tree which extends the proof skeleton. Then, the index is computed. Combining the information together, a new case is created and introduced into the right position in the case base.

4.5. Discussion

Nebel and Köhler performed an experimental comparison of MRL with SPA and PRIAR in the Blocksworld domain. It shows that MRL performs better than the other two planners, but worse than its generative component PHI alone. This result is due to the combination of the search tactics of PHI (and MRL, too) and the tested domains. PHI uses very efficient search tactics and hence it is easy to outperform the other planners for both PHI and MRL. The fact that MRL is being out-performed by its generative planner in the Blocksworld domain, however, does not imply that the case-based approach in the proof deductive system is not worth the extra effort; it is rather a property of the domain. For example, in the Mail domain MRL outperforms PHI. We find such result natural, especially considering that Blocksworld problems are proved to be in PTIME, while the matching problem that needs to be solved by the case-based approach is NP-hard. Clearly, a well implemented generative planner should outperform the case-based planner in such a domain, unless a heuristic approach to the matching problem is implemented. On the other hand in domains with higher complexity of plan generation, the case-based approach may be beneficial.

Besides outperforming PRIAR and SPA, MRL also provides a formal framework to study plan reuse and theoretical complexity of related tasks. The use of modal temporal logic introduces the possibility of specifying temporary goals (valid *sometimes* during the execution of the plan instead of at the end) as well as it allows to order the goals in a temporal sense ($goal_1$ needs to be achieved before $goal_2$). Lastly, MRL does not always need to verify the plan it produces, which may bring significant time savings when dealing with long or complex plans.

5. PRODIGY/ANALOGY

The PRODIGY/ANALOGY system [Veloso 1994] extends the planner PRODIGY[3] [Veloso et al. 1995] and it is the first case-based planner that executes the whole case-based cycle. The cycle consists of the following steps: generation/annotation of a case, its storage, retrieval and replay on a new problem. The first two steps define a case and a structure of a case base and constitute the retention step, while the later two steps show how to re-use a previously stored solution, describing the retrieval and adaptation phases. PRODIGY/ANALOGY was used, for instance, in route planning [Haigh et al. 1997].

5.1. Case representation

Differently from the previous planners, PRODIGY/ANALOGY does not adapt a previous solution plan. It rather tries to repeat the decisions that lead the generative search engine (PRODIGY) to a successful solution while it searches for a solution to the current problem. This is reflected in the definition of the case. Besides a *solution*, a case also contains a *derivational trace*, which provides the justifications of the solution and explains the choices made during the search. There are three kinds of justifications: *goal dependencies* capturing (partial) ordering among goals, *failed alternatives* recording search directions that did not lead to a solution and *external guidance* pointing to external knowledge which determined a search direction.[4] The solution is easy to obtain as it is the solution found by PRODIGY and the derivational trace can be generated by PRODIGY after minor modifications as well, as it is a record of the internal decisions the search engine makes.

For instance, Figure 5 shows the case structure stored by PRODIGY/ANALOGY for the problem shown in Figure 2. PRODIGY/ANALOGY is a backward search planner, so it starts from the goals, finding actions that achieve the goals and checking whether the preconditions of those actions are true in the current state. By default, it follows a linear planning strategy, so the first solution would

[3]In the first versions the underlying generative planner was NOLIMIT, a nonlinear planner similar to PRODIGY.

[4]PRODIGY is able to learn domain specific knowledge and such knowledge can then be used to perform a more informed search of the solution space.

move first `crate0` to `distributor0` and then `truck0` would come back to pick up `crate1` and bring it to `distributor0`. In order to find the optimal solution (that of using `truck0` for both crates at the same time, as shown in Figure 4), we would have to make it search for more than one solution. In order to compare the representation of cases with other techniques with respect to the same plan, we will show next the case representation of the optimal plan.

s00	(on crate0 pallet1)	*precond-of user*	s21	(load hoist0 crate1 truck0 depot0)	*relevant-to s19*
s02	(drop hoist1 crate0 pallet1 distributor0)	*relevant-to s00*	s22	(LOAD HOIST0 CRATE1 TRUCK0 DEPOT0)	
s03	(lifting hoist1 crate0)	*precond-of s02*	s23	(LIFT HOIST0 CRATE0 PALLET0 DEPOT0)	
s05	(unload hoist1 crate0 truck0 distributor0)	*relevant-to s03*	s24	(LOAD HOIST0 CRATE0 TRUCK0 DEPOT0)	
s06	(at truck0 distributor0)	*precond-of s05*	s25	(DRIVE TRUCK0 DEPOT0 DISTRIBUTOR0)	
s08	(drive truck0 depot0 distributor0)	*relevant-to s06*	s26	(UNLOAD HOIST1 CRATE0 TRUCK0 DISTRIBUTOR0)	
s09	(in crate0 truck0)	*precond-of s05*	s27	(DROP HOIST1 CRATE0 PALLET1 DISTRIBUTOR0)	
s11	(load hoist0 crate0 truck0 depot0)	*relevant-to s05*	s28	(on crate1 crate0)	*precond-of user*
s12	(lifting hoist0 crate0)	*precond-of s11*	s30	(drop hoist1 crate1 crate0 distributor0)	*relevant-to s28*
s14	(lift hoist0 crate0 pallet0 depot0)	*relevant-to s12*	s31	(lifting hoist1 crate1)	*precond-of s30*
s15	(clear crate0)	*precond-of s14*	s33	(unload hoist1 crate1 truck0 distributor0)	*relevant-to s31*
s17	(lift hoist0 crate1 crate0 depot0)	*relevant-to s15*	s34	(UNLOAD HOIST1 CRATE1 TRUCK0 DISTRIBUTOR0)	
s18	(LIFT HOIST0 CRATE1 CRATE0 DEPOT0)		s35	(DROP HOIST1 CRATE1 CRATE0 DISTRIBUTOR0)	
s19	(available hoist0)	*precond-of s14*			

Fig. 5. Case structure generated by PRODIGY/ANALOGY for problem specified in Figure 2. It shows three out of four types of nodes in PRODIGY/ANALOGY: goal, bindings and applied. We omitted operator nodes for compactness.

In Figure 5 we only show the main label of each node (search step) in the case for compactness. There are four kinds of nodes in PRODIGY: *goal nodes*, where PRODIGY decides which node to work on next (as nodes s00, s03, s06, ...); *operator (action) nodes*, where it decides which operator to use for a given goal (as nodes s01, s04, ...); *bindings nodes*, where it decides which objects to assign to the operator parameters (as nodes s02, s05, ...); and *applied operator nodes*, where it executes forward an instantiated operator (as nodes s18, s22, s23, ...).[5] As mentioned before, each of these steps is a more complex structure that includes information on: goal dependencies (as s03 goal depending on s00), failed alternatives (as (`drop hoist1, crate0, pallet1, depot0`) for step 02, since `pallet1` is initially at `distributor0`), and external guidance used (if we would have used control rules in this execution).

5.2. Retrieval

The case base of PRODIGY/ANALOGY is indexed by two levels of indices. The first level of indexing uses the goal statement — the goals are partially ordered so that they form independent components of interacting goals and are connected according to the partial order defined by the solution. On this level, a hash table is used. The second level is indexed by a set of features of the initial state that are relevant to the solution (footprint) and is stored as a discriminating network. Therefore, the system can efficiently distinguish cases that share significant parts of the goal description, but differ in some relevant parts of the initial state.

For example, PRODIGY/ANALOGY would index the case on Figure 5 by its interacting goals ((`on crate0 pallet1`), (`on crate1 crate0`)), and then by the footprint of the initial state (all literals from the initial state except those that were not used for finding the solution, which is (`at truck1 distributor0`) here). If it is the only element in the case base, it would create a single node of the discriminating network for the initial state of that case pointing to it from the

[5]We did not show operator nodes in the figure for space reasons.

corresponding position in the goals hash table (each entry of the hash table would have a pointer to the root of a discriminating network). As more problems are solved, and their solutions are stored as cases, they are first indexed in the corresponding position of the hash table. Then, the initial state of each new problem is compared against the initial state of the root of the corresponding discriminating network. The intersection of both initial states will remain in the root node, and the parts that are not common, will go to different branches of the discriminating network.

The similarity metric is not only based on the absolute number of matching goals and initial facts. It also considers: the interaction among the goals, favouring the cases with the goal interaction more similar to the current one; and the footprint of the initial state of the case. As an example, if we take problem in Figure 2 as a case in the library and a new problem specified in Figure 3b), PRODIGY/ANALOGY would compute the distance between the goals of both problems, and the distance between the footprint of the case and the initial state of the new problem. And it computes it for each potential substitution. Thus, if we consider the obvious substitution: $\sigma_0 = \{$(crate0/crate0) (crate1/crate1) ... (truck0/truck0)$\}$, the matching value would be:

$$\delta^{\sigma_0} = \delta_G^{\sigma_0} + \delta_S^{\sigma_0}$$

where $\delta_G^{\sigma_0} = 2$, since both problems have two equal goals (under substitution σ_0), and $\delta_S^{\sigma_0} = 13$, since the footprint of the case (complete initial state except for information on `truck1` that was not used) and the new initial state is of 13 literals. Thus, the similarity between both problems would be $\delta^{\sigma_0} = 2 + 13 = 15$. And, it would be greater than the similarity using $\sigma_1 = \{$(crate0/crate1) (crate1/crate0) ... (truck0/truck0)$\}$ for instance.

During the retrieval, the system selects a set of cases that address a subset of the current goal statement and so that the current initial state partially matches the features needed for achieving the goals. This set is not further filtered by means of an evaluation phase. It was the first system to retrieve and reuse multiple cases [Veloso 1997].

5.3. Reuse

PRODIGY/ANALOGY also addresses the planning task by searching. Analogous to PRODIGY, the planner combines a backward-chaining for goal-directed reasoning with simulated actions execution. The search choices are guided by the corresponding decisions stored in the retrieved cases. The justification of each decision is interpreted in the current context and if it still holds, the decision is repeated. When the justification does not hold, the system plans for the new goal using PRODIGY. It may be that at a decision point, the system has more than one case addressing the decision. One may adopt different strategies to decide which case to follow or how to merge the guiding information of several cases at the same time. In PRODIGY/ANALOGY, four strategies were considered: *serial*, that reused cases in a given order; *round-robin*, that reused cases in an alternation order; *eager*, that at each node selected to reuse the case that recommends to follow some of the children nodes; and *exploratory*, that selected cases to follow randomly at each node.

As an example, suppose the best scenario for the serial strategy. We have two simple cases in the case base in the Depots domain. In one case, a crate is clear and on a pallet in a given depot and the goal is to have it inside a given truck. The case would include (among other derivational information) the plan: (`lift hoist crate pallet depot`), (`load hoist crate truck`). In the second case, a crate is inside a truck, the truck is in a given depot and the goal is to have the crate at that depot on top of a given clear pallet. The case would include the plan: (`unload hoist crate truck depot`), (`drop hoist crate pallet depot`). Suppose we have now a new problem where there is a crate initially clear and on top of a pallet in `depot1`, a truck is in `depot1`, and we have to move that crate to `depot2`. If PRODIGY/ANALOGY retrieved these two cases as the most similar ones to the new problem, the serial strategy would follow first the derivation of the first case and load the crate in the truck. Then, it would try to follow the second case, but the truck is not at `depot2`. So, it has to interleave some steps of planning from scratch to include the subgoaling on (`at truck depot2`), the selection of operator `drive` and of its bindings, (`drive truck depot1 depot2`), and the execution of that action. Finally, it would follow the second case. If it would have used a different strategy for reusing cases, as the round-robin, it would have created a different search tree. So, as expected, the strategy can influence the results.

5.4. Retention

When a new problem is solved, the case is created by extracting the derivational trace and after computing the corresponding indices, the case is inserted into the case base. So, retention needs first to follow the solution path, recording all success decisions made, and their failed alternatives. Then, since PRODIGY/ANALOGY indexes cases, it has to compute the corresponding indexes (as explained in the retrieval subsection). First, for the individual goals of the new problem, and next for their combination. Finally, it has to follow the tree structure that stores the corresponding initial states to decide where to include the new initial state.

5.5. Discussion

The PRODIGY/ANALOGY system contributed greatly to the field of CBP, by implementing the whole case-based cycle. While doing so it clearly demonstrates the tight link among the phases of the CBP cycle, showing how to modify a generative planner to annotate its solutions in order to be stored and reused later.

The retrieval phase is novel in the use of an indexed case library. Before, the planners only established a similarity metric and considered every case in the library in order to identify the most similar one. By the use of a two level index structure, PRODIGY/ANALOGY can quickly narrow down the search, eliminating the cases that are likely not very useful. It is also the first time a possibility to reuse more than just one case is successfully implemented.

Lastly, PRODIGY/ANALOGY approaches the whole problem of plan reuse in a different way than transformational planners such as PRIAR or SPA. It starts solving the new problem by searching the solution space and when confronted with a choice, it tries to repeat a choice that has proven to be successful in a similar situation previously. Such method is called *reconstructive* and is defined by the transfer and reuse of the line of reasoning rather than adapting the final solution.

6. FAR-OFF

FAR-OFF is the first case-based planner adopting a heuristic approach to the retrieval and adaptation phases. In the retrieval phase, it introduces a novel similarity metric called ADG (Action Distance-Guided), and the stored solution plans are modified by a heuristic generative system based on FF[6] [Hoffmann and Nebel 2001]. Another improvement on the previous planners is the implementation of the case base update with a maintenance policy based on the competence of the case base, where the *competence* is interpreted in the sense introduced in [Smyth and McKenna 2001] for CBR.

6.1. Case representation

FAR-OFF [Tonidandel and Rillo 2002] models STRIPS-like states, actions and plans in Transaction Logic [Bonner and Kifer 1995], which is an extension of first-order logic that includes the serial conjunction operator. A case in such setting is a rule $\eta \leftarrow body$, where the $body$ is a serial conjunction of the case's preconditions (i.e., the initial state), a plan stored in the case, and the case's postconditions (i.e., the goals). The head of the rule η represents the case. The plans stored in cases are always complete and the actions and predicates are grounded, corresponding to the initial state and goals.

To improve the performance of the retrieval and maintenance phases, some of the stored cases are marked as *footprint* elements. The footprint cases have the same competence as the original case base, but they form a smaller case base. Each footprint case relates to a subset of similar cases, named Related Set [Smyth and McKenna 1999]. The Related Set is defined by means of a predicate $solves$ — the Related Set of a footprint element is a set of problems such that their solution can be

[6]The plan reuse engine is a Delphi implementation of FF.

obtained by adaptation of the solution of the footprint element; that is, the footprint element *solves* the cases from its related set. The union of both sets, the footprint cases and their Related Sets, forms the original case base. Such division allows quick screening of the case base during the retrieval and focusing the detailed search into the most promising area of the case base, namely the Related Set of the most similar footprint element.

6.2. Retrieval

The use of footprint cases effectively creates two levels in the case base. Hence the retrieval process proceeds in two steps. First, the footprint elements are considered and the most similar one is selected. Then, the Related Set of the selected footprint element is considered and the most similar case (or a set of ordered k most similar cases) is chosen to be reused. The selection process does not implement any matching function. Consequently, a case can be retrieved only if it shares the naming convention with the current problem. This decreases the usability of the system, even though it leads to a significant speed-up.

The definition of Related Set closely depends on the definition of the predicate *solves*. In FAR-OFF, a case c is said to solve another case, c', if and only if the ADG metric value of the case c for solving a problem defined as case c' is less than an upper limit ρ. The ADG metric estimates the adaptation effort needed in order to turn the plan stored in a case into a solution of the current problem (this metric can be seen as an inverse measure of similarity). First, an *initial similarity value* (δ_I) is computed, which is an estimation of the actions needed to turn the current initial state into a state satisfying the initial conditions of the case. Then a *goal similarity value* (δ_G) is computed, which estimates the distance between the state reached by the application of the plan stored in the case and a goal state of the current problem. The FF heuristic (the size of the relaxed plan) is used to estimate the distance between two states. A relaxed plan is the solution of the planning problem where no deletes of actions are considered. Computing a relaxed plan can be done in polynomial time, while computing the optimal one is NP-hard. The length of the relaxed solution is an estimate of the length of the optimal (non-relaxed) solution. The ADG similarity metric is simply a sum of the initial and goal similarity values.

Obviously, the limit ρ of the similarity value influences directly the number of footprint cases of a case-base as well as the number of cases in the Related Set of each case. A high value of ρ results in a case base with only few footprint cases, where each footprint case can solve almost all cases. However, the adaptation costs are likely to be very high. On the other hand, if the value of ρ is small, one case could solve only a very limited set of other cases and the case-base would have many footprint cases with insignificant Related Sets. Hence a reasonable value of ρ needs to be chosen if one wants to achieve any significant improvement in the retrieval time.

Considering our running example, we assume for simplicity that the problem `case1` is a footprint case; in this situation when `problem1` is presented to FAR-OFF, the system computes the ADG similarity metric which estimates the number of actions that are needed to transform `case1` to a solution of `problem1`. The first component of ADG is the *initial similarity value* (δ_I). It estimates the similarity between the initial state of `problem1` and the initial relevant facts of `case1`. Here, the only relevant fact which is unsupported in `case1` w.r.t. the initial state of `problem1` is (`clear crate1`). The system then computes a relaxed plan that estimates the adaptation cost; the only action that belongs to the relaxed plan is (`lift hoist0 crate2 crate1 depot0`).

The second value that needs to be computed is called *goal similarity value* (δ_G) and it is a distance estimation between the goal state of `problem1` and the final state features of `case1`. The unsupported goal is (`on crate2 crate1`) and the system builds a relaxed plan with actions (`drop hoist1 crate2 crate1 distributor0`) and (`unload hoist1 crate2 truck0 distributor0`). Then the total adaptation cost is estimated to be three actions.

6.3. Reuse

FAR-OFF does not attempt to modify the plan stored in the retrieved case. The authors have noted that such a process may be extremely time-consuming if all the possible permutations of the operators need to be considered. Instead, the stored plan is treated as a part of the solution, which only needs to be completed. Intuitively, the system generates a partial plan that connects the current initial state with the initial state of the stored plan and another partial plan to connect the case's goal state with the current one. The solution plan is then a concatenation of the partial plan from the current initial state, the stored plan and the partial plan to reach the current goals. To generate the partial plans, an FF-based generative technique is used. Basically, the reuse phase performs very similar steps to the ones of the retrieval phase. However, the retrieval phase only *estimates* the length of the plans to be concatenated with the stored one, as it considers the length of relaxed plans computed by an FF heuristic, while during the reuse phase the same heuristic is used to find the solution plan.

The reuse may fail if the partial plan cannot be generated. In such situation, if the retrieval phase provided a set of k cases to be adapted, the next element is attempted to be reused.

Suppose that `case1` has been retrieved for reuse; then the system builds a partial plan that connects the current initial state of `problem1` with the initial state of `case1`, which contains the actions (`lift hoist0 crate2 crate1 depot0`) and (`load hoist0 crate2 truck0 depot0`). Similarly, a plan is concatenated at the end of the solution plan of `case1` (see Figure 4) in order to solve the goals of `problem1`; then the system introduces the actions (`unload hoist1 crate2 truck0 distributor0`) and (`drop hoist1 crate2 crate1 distributor0`) in order to produce a complete solution plan.

6.4. Retention

When a new case is introduced into the case base of FAR-OFF, then the competence of elements, the footprint element set and the Related Sets need to be recomputed using the ADG similarity metric. To avoid the case base occupying a large amount of space, not all solutions are stored. FAR-OFF employs a case-deletion policy based on the *minimal-injury* method. When the number of cases stored in a case base (or the total amount of space occupied by the case base) reaches a given limit, the deletion policy is called. The policy chooses in a greedy manner the case that causes the minimal injury to the competence of the case base and removes it. It repeats this process until the case base is of a reasonable size again.

Authors also mention the problems related to the complexity of the case base update. The re-computation of the footprint cases may take up to $O(N^2)$, where N is the number of stored cases. Nevertheless, this does not cause severe problems, as the maintenance and update of the case base is not performed with each run of the planner and can be performed offline.

6.5. Discussion

In the experiments, FAR-OFF was forced to reuse a previous solution, even though the system is capable of heuristically choosing between reusing a case or generating a solution from scratch if no suitable case is found in the case base. FAR-OFF is reported to perform well in the Blocksworld domain — it solves more problems than the FF-based generative planner and sometimes FAR-OFF even outperforms it in the CPU time. Other experiments were performed over the Logistics domain, in which the FF's heuristic works very well. Even though FAR-OFF does not outperform the generative planner there, its planning time is usually very close. The retrieval phase during the experiments is reported to require less time than the reuse, which makes the case-based approach feasible. If the problems recur, the overall savings in the CPU time are highly probable.

From the theoretical point of view, FAR-OFF differs from the previous systems in several aspects. It is the first system that heuristically prunes some parts of the case base when looking for a suitable reuse candidate, because it rules out all the cases relevant to the footprint elements that are not similar to the current problem. This lowers the time needed by the retrieval, but it may also lower the quality of the reuse candidate as the best one may be missed. The retrieval phase of FAR-OFF is also shortened by the absence of an object matching procedure, which is proven to be the real bottleneck of the retrieval (and arguably of the whole CBP process). However, it significantly decreases the chances of successful reuse.

$$\begin{array}{l}
\mathcal{Q}_{\text{truck0}}=\{(\{at_1\},\emptyset),\\
\qquad(\{at_1\},\text{no-op}),\\
\qquad(\{at_1, in_2\},load_3),\\
\qquad(\{at_1, in_2\},\text{no-op}),\\
\qquad(\{at_1, in_2, in_2\},load_3),\\
\qquad(\{at_1, in_2, in_2\},drive_1),\\
\qquad(\{at_1, in_2\},unload_3),\\
\qquad(\{at_1, in_2\},\text{no-op}),\\
\qquad(\{at_1\},unload_3),\\
\qquad(\{at_1\},\text{no-op})\}
\end{array}
\qquad
\begin{array}{l}
\mathcal{Q}_{\text{crate0}}=\{(\{at_1, on_1, on_2\},\emptyset),\\
\qquad(\{at_1, on_1, clear_1\},lift_3),\\
\qquad(\{at_1, on_1, clear_1\},\text{no-op}),\\
\qquad(\{lifting_2\},lift_2),\\
\qquad(\{in_1\},load_2),\\
\qquad(\{in_1\},\text{no-op}),\\
\qquad(\{lifting_2\},unload_2),\\
\qquad(\{at_1, on_1, clear_1\},drop_2),\\
\qquad(\{at_1, on_1, clear_1\},\text{no-op}),\\
\qquad(\{at_1, on_1, on_2\},drop_3)\}
\end{array}$$

Fig. 6. Cases stored by CABALA for `truck0` and `crate0` for problem in Figure 2.

If the retrieval is successful and a reuse candidate is found, the stored solution is reused *fully*. The previous systems tend to reuse only those parts of the stored solution that match the current situation. In a sense, the reuse implemented in FAR-OFF can be viewed as conservative, but without the inherit hardness related to the identification of the maximal reusable portion of the retrieved plan, which is always reused whole. The stored plan is used and connected to the current initial state and goals by newly generated plans. Such approach on one hand simplifies the reuse process, because there is no need to consider all the possible modifications of the plan, on the other hand it decreases the quality of the solution as the resulting plans tend to be longer than the optimal ones.

Another novelty of the approach is the implementation of the retention step. The authors provide an algorithm to maintain the case base, which is based on the competence and hence suitable to be used over the whole lifetime of the case base without decreasing its performance.

7. CABALA

CABALA [de la Rosa et al. 2013] was designed to be a recommendation system for heuristic planners, like FF. It incorporates a full CBP cycle, from retrieval to retaining new cases. It has been tested in off-line as well as in on-line settings. One of the main differences with the rest of approaches is that the case representation is based on a plan trace, but from the point of view of a specific object. Therefore, CABALA generates several cases from each plan; one for each object in the plan. Another difference with some of the previous approaches is that CABALA reuses plans as search guidance.

7.1. Case representation

Cases are represented using a structure called *typed sequence*. We will use the plan in Figure 4 as the running example, that is the solution found by a planner to the problem depicted in Figure 2. Given the plan in Figure 4, CABALA creates one case for each object involved in the plan. As an example, the typed sequences for `truck0`, $\mathcal{Q}_{\text{truck0}}$, and `crate0`, $\mathcal{Q}_{\text{crate0}}$, are shown on Figure 6. Each case represents the plan from the perspective of a specific object. A case is a list of pairs, one for each action in the plan. So, it is composed of the state after applying the action, and the corresponding action. Since CABALA is only interested on what happens to the particular object at each step in the plan, CABALA summarizes the state and the action with the subset of information relevant to the object. Thus, the relevant state (also called typed sub-state) is a set of literals where the object is an argument (CABALA only stores the position of the object in the literal), and the action is labeled with the position of the object in the action arguments. For instance, the third pair in the `truck0` case, $(\{\mathtt{at}_1, \mathtt{in}_2\}, \mathtt{load}_3)$, represents the fact that `truck0` is the first argument of an `at` literal (represents the location of the truck), and is the second argument of an `in` literal (an object has been loaded in the truck). The action $\mathtt{load}_3$ represents that `truck0` is the third argument of the `load` action (the plan loaded an object in that truck in a given location).

All cases related to trucks would be stored in the `truck` case base, while all cases related to crates would be stored on the `crate` case base. In order to use less memory, CABALA does not store cases for objects that were not involved in the plan (no action in the plan uses them). For instance, in the running example, `truck1` does not appear in the plan, so CABALA does not store a case for it.

7.2. Retrieval

As each planning state contains information on several objects, CABALA retrieves one case for each object in the new problem P. So, for each object o in P, it computes the similarity of P with all stored cases P' according to o. The features used by the similarity function $s(P, P', o)$ are: the object type of o (to retrieve all cases of that type); the typed sub-state generated from the initial state I of the new problem, $\varphi_{I,o}$, as well as the initial state I' of the previous case, $\varphi_{I',o}$; the typed sub-state generated from the set of goals G of the new problem, $\varphi_{G,o}$ as well as the goals G' of the previous case, $\varphi_{G',o}$; and some extra information related to the relaxed plan footprint computed for the initial state.

As an example, suppose CABALA solves the problem in Figure 2 and stores all cases (one per object that appears in the plan) in the case base. Now, it is given a new problem to be solved as specified in Figure 3b. For each object in the new problem, `depot0`, `distributor0`, `truck0`, ..., `hoist1`, it would retrieve the cases corresponding to their respective types (`depot, distributor, truck, ..., hoist`) and match initial and goal typed sub-states. For instance, given `truck0`, there is only one case stored for trucks from the problem in Figure 2. In that case, the retrieval would return that case. However, there are two cases for each of the three crates in the new problem (`crate0`, `crate1` and `crate2`): one for the previous `crate0` and another one for `crate1`. Thus, it would compute the similarity of initial states and goals of the two stored cases against the initial state and goals of the new problem (from the perspective of each new crate). It is clear from Figure 7 that the match between `crate0` of the new problem with `crate0`$'$ of the case is perfect, while the match between `crate0` and `crate2`$'$ of the new problem is worse.

Initial typed sub-state of	`crate0`$'$: at_1, on_1, on_2	`crate2`$'$: $at_1, on_1, clear_1$
	`crate0` : at_1, on_1, on_2	`crate0` : at_1, on_1, on_2
Goals typed sub-state of	`crate0`$'$: on_1, on_2	`crate2`$'$: on_1
	`crate0` : on_1, on_2	`crate0` : on_1, on_2

Fig. 7. Comparison of the initial states and goals for `crate0` of the case with `crate0`$'$ and `crate2`$'$ of the new problem.

Then, the most similar case for each object is selected, resulting in the retrieval of a set of cases, one per object in the new problem. In the example problem, there are 11 objects, so 11 cases would be retrieved. For `depot0` and `distributor0`, there is only one case for each. CABALA only saved one case for `truck0` and `truck1`, given that `truck1` did not intervene in the previous plan. So, each new truck would be associated to the same case. As for the new pallets, each would be assigned the most similar case for each. Thus, `pallet0` would be better matched with the case of `pallet0` in the case base than the one corresponding to `pallet1`. And the same would happen to `pallet1`. For hoists, there would be two cases in the library (one for each previous hoist). The initial and goal states would be equal. The cases would differ though, since `hoist0` was used in the beginning of the plan and `hoist1` was used at the end. If it would only match the initial states and goals, there would be no difference. Thus, CABALA also matches the footprint of both hoists. The footprint is computed from the relaxed plan, focusing on each specific object. Details can be found in the paper [de la Rosa et al. 2013]. This serves to differentiate the best case for each hoist. And, it would retrieve the previous `hoist0` case for the new `hoist0` and the previous `hoist1` case for the new `hoist1`. And the same reasoning would be done for the crates.

7.3. Reuse

Similarly to PRODIGY/ANALOGY, the retrieved cases are kept in a replay table which keeps track, for each retrieved typed sequence, of the pointer to the last element (action in the type sequence) used in the current search. The advise given by typed sequences is usually independent of the search algorithm. Accordingly, CABALA uses the concept of a "recommended node", to recognize a promising successor suggested by the CBP component. Once a node is recommended, the search technique can use two basic strategies to consider the advise, generating several versions of CABALA, depending on the search technique and the recommendation strategy used:

— Pruning strategy. A recommended node (if exists) can be directly selected, discarding the rest of successors. This is 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.

7.4. Retention

Once a new problem is solved, CABALA generates the corresponding cases and stores them in the case base. It also keeps track of measures as percentage of each case correctly used by the search algorithm to assess the utility of each case. This information is used for providing more informative recommendations; they are not simply recommend a given node (action) or not, but recommend it with a given strength.

7.5. Discussion

Results on eight planning benchmarks[7] showed the benefits of using typed sequences during the search process. Improvements obtained by different CBP algorithms were due to the reduction in the number of evaluated nodes. Since computing the relaxed plan heuristic for all nodes in the search tree 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.

In comparison with previous approaches (as the ones surveyed in this paper), CABALA case structure is based on objects types. Therefore, object matching is fast, focused on the type sequences stored for the types of objects that appear in the new problem. However, this case base structure loses information on the specific relations among objects. Thus, the reuse is a soft bias instead of a strong recommendation. CABALA reuses several cases as PRODIGY/ANALOGY does (one of the main differences with other approaches such as SPA or PRIAR). CABALA uses a heuristic forward planner, as FAR-OFF does, in contrast to previous systems that used non-heuristic backwards search planners, such as SPA, PRIAR or PRODIGY/ANALOGY.

8. OAKPLAN

OAKPLAN [Serina 2010] is a case-based planner that uses heuristic approaches to address the retrieval process — it achieves an efficient retrieval by using graph-theoretical techniques and kernel functions [Scholkopf and Smola 2001]. The adaptation is implemented in an incremental generative manner using the LPG-adapt procedure [Fox et al. 2006], which performs a local search in the space of plans.

8.1. Case representation

Each case in OAKPLAN consists of a description of a planning problem, that is, the initial state and goals, the graph representation of the problem called *Planning Encoding Graph* used to compute the kernel functions, the degree sequences of this graph [Ruskey et al. 1994] used to avoid considering dissimilar cases, and the solution plan represented as a sequence of actions.

The underlying idea of the Planning Encoding Graph is to provide a description of the "topology" of a planning problem without making assumptions regarding the importance of specific problem features for the encoding. The Planning Encoding Graph of a planning problem Π is the union of the directed labelled graphs encoding the goals and the propositional initial facts. The *Initial Fact*

[7]The number of tested domains greatly increased with respect to other previous approaches.

Encoding Graph of a propositional initial fact $\rho = (p \;\; c_1 \ldots c_n)$, $\rho \in \mathcal{F}, p \in \mathcal{P}, c_1, \ldots, c_n \in \mathcal{O}$ is a directed graph where the first node is a propositional node I_p, which is connected with c_1 by an edge with label $\{I_p^{0,1}\}$. The object node c_i is labelled with the type of the object c_i and it is connected with all the remaining nodes c_j by an edge with label $\{I_p^{i,j}\}$. For example, the initial fact (`on crate1 crate0`) of our running example of Figure 3a determines an Initial Fact Encoding Graph that connects the I_{on} propositional node to the `crate1` node (the label of this edge is $\{I_{on}^{0,1}\}$) and it is connected with the node `crate0` by an edge with label $\{I_{on}^{1,2}\}$. The object nodes `crate0` and `crate1` are labelled by the corresponding object type (`crate`).

Similarly, it is possible to define the *Goal Fact Encoding Graph* of a propositional goal ρ using the token G_p instead of token I_p. Furthermore, in [Gerevini et al. 2012] the authors propose an extension of the Planning Encoding Graph in order to work with numeric domains involving resources and numeric preconditions and goals.

8.2. Retrieval

The retrieval is split into three steps. The first step is a filtering algorithm, which prunes as many cases as possible, leaving only a small group of candidates for the reuse. The primary screening uses *degree sequences* [Ruskey et al. 1994] of the Planning Encoding Graphs in order to avoid to consider unpromising planning problems. It only compares the precomputed degree sequences of the case base elements with the degree sequences of the current problem and hence it can be performed in $O(m \, log \, m)$, where m is the number of nodes of the considered graphs.

OAKPLAN computes the upper bounds on the similarity among the Planning Encoding Graphs of the case base elements and the Planning Encoding Graph of the current problem. The cases that are promisingly similar are passed to the next step, where their objects are matched to those of the current problem. As was mentioned before, object matching is an NP-hard problem and therefore OAKPLAN proceeds heuristically, using kernel functions to approximate the optimal resolution of the matching problem. Specifically, the task of the kernel functions is to define a mapping function from the case's objects to the objects of the current problem, so that their Planning Encoding Graphs are as similar as possible after the objects are matched. The main advantage of using kernel functions at this stage is their efficiency — their computation requires only polynomial time, namely $O(m^5)$ [Serina 2010], where m is the number of nodes of the smaller Planning Encoding Graph.

The retrieval phase is then able to identify a set of cases with Planning Encoding Graphs that better fit the Planning Encoding Graph of the current planning problem and use the corresponding matching functions to "translate" the solution plans of the corresponding case base elements in candidate solutions of the current planning problem. These solution plans are then carefully evaluated in order to determine their capability to solve the current planning problem.

This evaluation simulates the execution of the stored plans in order to identify the unsupported goals and the unsupported preconditions of the actions in the plans. The different importance of the inconsistencies related to unsupported facts is estimated by computing relaxed plans starting from the states obtained simulating the execution of the actions in the candidate solution plan preceding the different inconsistencies. The number of actions in each relaxed plan corresponds to the complexity of making the selected facts supported; similarly the sum of the number of actions in the different relaxed plans is used in order to determine the competence of the stored plan π to solve the current planning problem. The estimated adaptation cost of the best stored plan is compared with the estimated cost of generating the new plan from scratch in identical manner. The reuse is applied only if it is estimated to be worth.

Consider the solution plan of Figure 4, obtained by a matching process that associates all the objects of the case depicted on Figure 3a to the objects with the same name of the current planning problem (Figure 3b). The first action the system tries to apply during the evaluation phase is (`lift hoist0 crate1 crate0 depot0`) which has (`clear crate1`) as unsatisfied precondition. Starting form the initial state of `case1`, the system builds a relaxed plan in order to satisfy this precondition without invalidating the other preconditions that are supported. The actions selected for in this relaxed plan are (`lift hoist0 crate2 crate1 depot0`) that makes (`clear crate1`) supported and (`load hoist0 crate2 truck0 depot0`) that makes (`available hoist0`) supported. Similarly, for the unsupported goal (`on crate2 crate1`), the system builds a relaxed plan with actions (`unload hoist1 crate2 truck0 distributor0`) and (`drop hoist1 crate2 crate1 distributor0`). Hence the estimated adaptation costs sum up to four.

8.3. Reuse

OAKPLAN uses LPG-adapt [Fox et al. 2006] to modify the retrieved plan so that it solves the current problem. LPG-adapt addresses planning as a local search in the space of plans. It starts its search from the plan that was retrieved and incrementally modifies it until it reaches a flawless plan. The search process may continue even after a solution is found and produce a sequence of valid plans. Each plan has a better quality than the previous ones or it has a better stability with respect to the retrieved plan. LPG-adapt also features a mechanism for leaving local optima by simply introducing few random inconsistencies into a valid plan that was found, but that did not improve over the best solution so far. Nevertheless, the reuse step of OAKPLAN can be performed by any other adaptation planner.

8.4. Retention

Similarly to other systems like PRODIGY/ANALOGY or CABALA, there is an extra work that needs to be done in order to create a new case and introduce it into the case base. First, only initial state *relevant* facts, that is, only facts needed for the execution of the solution plan, are identified. Those would be equivalent to the footprint computed by PRODIGY/ANALOGY or CABALA. Then, the Planning Encoding Graph is constructed, using the initial state relevant facts instead of the original initial state. The Planning Encoding Graph together with its degree sequences and the solution plan then constitute the new case, which is inserted into the case base. As a last step, the additional data structures supporting the retrieval phase are updated.

The problem of case base maintenance was studied in [Gerevini et al. 2013a; 2013b] and a second problem related to retention is formulated. Apart from deciding which case to add (and how to implement such addition, as was discussed in the previous systems), OAKPLAN also addresses the problem of the excessive growth of the case-base by proposing policies to reduce a case base. The system implements several kinds of *offline* maintenance policies of different computational complexity, yielding case-base reductions of various qualities. The maintenance step is invoked by the user, who also specifies which strategy is used for the reduction. The strategies range from a completely uninformed, but very fast *random* policy to a fully informed *weighted coverage-guided* policy, which is however NP-complete and needs to be approximated.

8.5. Discussion

The OAKPLAN system aims at improving the retrieval phase, which is the bottleneck of the case-based planning approach. Similarly to PRODIGY/ANALOGY and FAR-OFF, it first filters out the cases that have a low similarity. However, the screening procedure of OAKPLAN considers briefly all elements of the case base, while FAR-OFF considers only a subset of the case base (the footprint elements) but spends more time evaluating those. After the filtering, the retrieval is based on the estimated cost of repairs needed in order to apply the stored plan to the current problem, which resembles the approach of PRIAR.

The adaptation phase does not bring any significant (theoretical) improvements. It uses an incremental generative planner to generate a new solution plan based on the stored solution. OAKPLAN uses LPG-adapt for this purpose, but it can be easily replaced by any other reuse planner as the rest of the system is independent on the choice. Note that, for instance, FAR-OFF uses information relevant to the generative planner used in the adaptation phase also in its retrieval phase. This is not the situation of OAKPLAN, where the strategy for modifying the retrieved solution does not influence any other part of the case-based process.

Many of the previous systems (e.g. PRIAR, SPA) ensured completeness of the planner by the ability of the adaptation procedure to change any of the decisions made in the stored plan and consequently to produce a new solution plan that has nothing in common with the stored one. OAKPLAN can also produce a completely different solution, but in two different ways: if a sufficiently similar problem is found in the case base, but for some reason the solution is not applicable, LPG-adapt finds one completely unrelated. If the retrieval step fails and does not provide any reuse candidate, OAKPLAN runs LPG-adapt from an empty plan, which equals to generation from scratch. Moreover, LPG-adapt is very robust; the local search it performs can very quickly change significant parts of the plan if needed, providing solution plans even if the retrieved case is quite dissimilar to the solution plan to be found.

9. SIN

SIN (*S*HOP *i*ntegrated with *N*aCoDAE) [Muñoz-Avila et al. 2001] is a case-based planning algorithm that was implemented in the HICUP planning system [Muñoz Avila et al. 1999]. HICUP worked on military planning operations in a Noncombat Evacuation Operations Domain, and the specific use introduced several requirements on the underlying planning algorithm. Due to the military use, the plans are hierarchical, the domain theory is incomplete, as the military doctrine cannot cover all possible courses of action, and the planner has only limited information about the current state. To address these requirements, SIN combines a generative, hierarchical planner SHOP [Nau et al. 1999] with a retrieval system based on interaction, NACODAE [Breslow and Aha 1998]. It uses cases to cover for the missing domain knowledge and state information.

The retrieval and reuse in SIN interleave much more than in other CBP systems. While the other planners surveyed here perform first the retrieval step and then try to reuse the retrieved cases, SIN chooses whether to decompose the task at hand by means of the stored cases (retrieve) or whether it uses the generative engine, mimicking the reuse step when no suitable case has been retrieved. The decision whether to generate a new task decomposition or to retrieve one from the case base depends on which of the supporting systems is capable of doing so, and which system is currently active. Note that if SIN retrieves a case to decompose a task, the consecutive reuse step is not needed, as the decomposition is a valid one. Similarly, if there is no case matching the task to be decomposed, the “reuse” step will try to generate a brand new decomposition. Hence, one could understand the two phases as mutually exclusive.

9.1. Case Representation

SIN is able to combine the generative knowledge coming from an HTN-based (Hierarchical Task Network [Erol et al. 1994]) planner with the experiences stored in the cases. The cases are designed to aid such interaction. In HTN, the domain knowledge consists of *methods*, which specify how and when a task can be decomposed to subtasks, and *operators*, which allow transforming a state to another state. In SIN, a case is an instance of a method enriched by a set of preferences related to matching of a case to the current state. The preferences are ordered pairs of question and answer that, when it lacks complete information about the state, allow it collecting extra information from a user and consequently can rank the cases according to their match to the current state.

9.2. Retrieval

SIN uses NACODAE [Breslow and Aha 1998] to retrieve cases from the case base. Differently to retrieval algorithms of other case-based systems, NACODAE does not require a complete world information. On the contrary, one of its tasks is to elicit the exact world state by gathering the information about the state from the user as well as assessing the user’s preferences. This is achieved by *conversations* with the user, providing NACODAE with a rank of the conversational case retriever.

When NACODAE is presented with a compound task that needs to be decomposed, it queries the case base and selects all the cases that are not contradictory to the current (partial) state and task to be decomposed. Then, the system tries to specify the current world state as closely as possibly by interacting with a user. Such interaction takes the form of a sequence of questions and answers, which help determine how applicable are different cases in the current situation. Based on the answers of the user, the retrieved cases are re-ranked and at the end of the conversation, the most suitable case is retrieved. Such a case is then used to decompose the task and SIN continues to decompose the rest of the compound tasks in the plan.

NACODAE stops either when all the compound tasks have been decomposed, or when there is no suitable case to be reused (or when the user refuses the proposed cases). In the latter, SIN hands the control over to SHOP to generate a new task decomposition.

9.3. Reuse

Given the current (partial) state, an incomplete domain theory and a set of cases, SIN tries to refine a set of tasks into a plan. To do so, it may use the cases retrieved by NACODAE as well as find a new decomposition by SHOP.

A case in SIN is an instance of a method, therefore an application of a case can be viewed as a single step in a derivational trace. When no suitable case is available for decomposing the task at hand, a new decomposition may be found by SHOP [Nau et al. 1999]. SHOP models the world using complete knowledge about its states and performs a total-order forward-search on the states in combination with HTN-style problem reduction. The resulting search strategy is quite similar to Prolog's and turned out to be very successful in the real world applications such as manufacturing planning.

SHOP terminates after it had successfully decomposed all the compound tasks (and hence produced a solution), or when it needs to backtrack on a compound task that has no more applicable methods to be decomposed (a failure). In the former, SIN terminates as well, offering the same solution as SHOP had found. In the latter (SHOP failed), the control is handed over to NACODAE, which tries to decompose the remaining tasks by replaying cases from the case base.

9.4. Retention

SIN does not implement any retention algorithm. This is due to the high importance of human expert's opinion and approval given by the army domain for which SIN was developed. By design, SIN is initialized with an incomplete domain theory and a case base. The domain theory is derived from the military doctrine and standard operating procedures and provides the system with a knowledge how to partially decompose high-level tasks. However, the full decomposition to primitive tasks often requires the use of previous experiences, which are stored in the case base. The case base is therefore crucial for finding a solution and guaranteeing its qualities and the contents of the case base can be changed only by the responsible human experts.

9.5. Discussion

Similarly to PRIAR, the planning algorithm is based on hierarchical task decomposition and hence requires a richer domain model than previous planners. However, SHOP considers the tasks in the order they will be executed in the plan, which avoids some issues related to tasks interactions and goals interactions and results in a much simpler algorithm than NONLIN, used in PRIAR.

Differently from the previous systems, SIN can also handle problems with incomplete domain information, as the missing information is retrieved from the experience stored in the case base or during the conversations with the user. By doing so, SIN opens an interesting direction for case-based systems by showing the potential they have in domains and situations where not all information is guaranteed to be known during the implementation of the planner.

10. ML-CBP

The development of Model-Lite Case-Based Planner (ML-CBP [Zhuo et al. 2012]) has a slightly different motivation than most of the planners presented here. The authors are concerned with finding a plan in situations where the domain theory is incomplete. Hence, a planning task for ML-CBP is described by an initial state $\mathcal{I}$, a goal specification $\mathcal{G}$ and a set of *incomplete action models* $\widetilde{\mathcal{A}}$. An incomplete action model contains actions where some predicates from the $pre(a)$, $add(a)$ or $del(a)$ are missing. The library of plans contains plans that are valid for such a domain, and hence extends the domain knowledge of the planner. The search for the solution combines the use of the incomplete domain knowledge to create a *skeletal* plan, whose correctness is then increased by using fragments of the stored plans.

10.1. Case Representation

The reuse information in ML-CBP is a *plan example*; that is, the actions present in the plan comply with the action model $\widetilde{\mathcal{A}}$, but no propositions are missing in their description. Every plan example is composed of an initial state, a sequence of actions and a goal state obtained by application of the actions sequence from the initial state. When ML-CBP receives a new problem to be solved, it decomposes the plan examples into *fragments* relevant for solving the new problem, the set of such fragments then plays the role of the case base.

10.2. Retrieval

When a new problem is presented to the system, a suitable mapping is found and applied to all the stored plan examples. The mapping can cause some actions of the plan examples not to be valid/applicable any more. Such actions are removed from the plan example and the resulting sequence of actions is called *plan fragment*.

The quality of the mapping is determined by the number of propositions of the initial state and goal specification shared by the current problem and the plan example after the mapping was applied to it. The search for a suitable mapping is guided by the features/properties of the objects involved — e.g., only mappings between crates that are "on" in the Depots domain are considered. Such a constraint is claimed to reduce the amount of mappings that need to be considered so that a suitable mapping can be found at reasonable computation costs.

Even with the best available matching, the probability of finding a plan fragment corresponding to the current problem is quite low. Therefore, a utility of each plan fragment is estimated and the best one is selected. The utility of the plan fragment is based on the frequency of its occurrence in different plan fragments. Hence, the problem of retrieval can be translated into a frequent sequential pattern-mining task and solved as such. The frequent sequential pattern-mining task consists of finding the complete set of sequential patterns whose support is larger than the threshold, where the support of a pattern is given by the number of sequences in the database that contain the pattern. The input to the problem is a sequence database which is effectively the case base here, and a threshold specifying the support, which may depend on the domain or other settings. To address the mining problem, the authors use the SPADE algorithm [Zaki 2001] and from the resulting patterns they choose only the ones of maximal length. The set of such fragments is then passed to the adaptation phase.

10.3. Reuse

The input of the adaptation phase is a skeletal plan based on $\langle \mathcal{I}, \mathcal{G}, \widetilde{\mathcal{A}} \rangle$, represented as a set of causal pairs; a set of plan fragments based on plan examples and causal pairs; and a set of frequent fragments with a specific threshold obtained by retrieval. A causal pair is a pair of actions such that the first one provides one or more conditions for the second.

The frequent fragments are integrated together as indicated by the skeletal plan to form the final solution as follows: a fragment that contains one or both actions of a causal pair is appended at the beginning or the end of the solution plan and all the causal pairs that got satisfied are removed from the skeletal plan. The causal pair is satisfied if the (partial) solution contains both actions of the pair. This is repeated until there are no causal pairs left (the skeletal plan is fully resolved by the frequent fragments) or there is no fragment to be used and the solution cannot be found. Note that ML-CBP is the only system presented here which uses a custom made algorithm in the reuse step instead of employing another adaptive planner.

Table I. Extended version of Spalazzi's table — case and case base.

system	plan representation	memory organisation	stored solution
PRIAR	transformational	flat	validation structure
SPA	transformational	flat	plan, causal links and constraints among the actions
MRL	transformational	indexed	LLP-formula
PRODIGY/ANALOGY	derivational	indexed	derivation trace
FAR-OFF	transformational	indexed	transaction logic rule
CABALA	derivational	semi-indexed by types	typed sequences
OAKPLAN	transformational	indexed	plan and supporting structures
SIN	hybrid	flat	task decomposition
ML-CBP	transformational	flat	plan example

10.4. Retention

The system has a set of plan examples at its disposal that are maintained fixed over the lifetime of the planner. However, this set of examples is turned into a set of plan fragments specific for the problem the system is trying to solve, where the set of plan fragments functions as a case base. The plan fragments are obtained from the plan examples by mapping the objects using the mapping defined by the retrieval and filtering out the actions that no longer comply with the domain theory due to the mapping.

10.5. Discussion

Zhuo *et al.* [2012] experimentally observed that the system solves a comparable number of problems as OAKPLAN if aided with a complete domain knowledge. The average plan length tends to be worse than those generated by generative planners, such as FF, which is understandable considering the way the plans are constructed. On the other hand, in some domains the solutions found by ML-CBP are reported to have better quality than those found by FF, which is probably due to the high quality of the provided plan fragments.

The mapping is similar to the one used by OAKPLAN, but, differently from OAKPLAN, it considers all the propositions, not only the relevant ones. Furthermore, the mapping is heuristic guided by the number of matching predicates the objects appear in in the two instances, but the choice of such heuristic is not clarified. Note that at the date of this survey, ML-CBP (as well as OAKPLAN) is still being developed and the implementation can be improved in later work.

11. CONCLUSION

In this work, we have surveyed several case-based planners to demonstrate the evolution of the field as well as the variety of approaches that can lead to a successful implementation. The described systems range from very simple to more elaborate ones that, under some conditions, can compete with generative planners. We close the paper with an overall comparison of the surveyed planners, as well as a survey of very related work on CBP that covers either domain-dependent CBP or CBP in execution environments.

11.1. Evolution of CBP

It is nearly impossible to compare the performance of the surveyed techniques, as the systems were developed over quite a long period of time having different computational resources and different goals in mind. Also, the understanding and competence in the whole area of automated planning evolved since the introduction of the first of these systems until today. Therefore, we propose here just a snapshot of different features of the case-based planners that may be of interest. Tables I and II feature the categorisation of presented approaches according to the taxonomy proposed by Spalazzi [2001], extended to the planners not listed there (below the line). Note that the planners are sorted in the same order as they were described here, not chronologically.

Table II. Extended version of Spalazzi's table — case-based cycle.

system	retrieval	reuse	retention
PRIAR	associative	heuristic-based, generative	accumulation
SPA	associative	heuristic-based, generative	accumulation
MRL	hierarchical	heuristic-based	classification
PRODIGY/ANALOGY	hierarchical	generative, merging, derivational replay	accumulation
FAR-OFF	hierarchical	heuristic-based, generative, merging	accumulation
CABALA	associative	derivational replay, search recommendations	accumulation
OAKPLAN	hierarchical	heuristic-based, generative, merging	selective
SIN	conversational	generative, derivational replay	manual
ML-CBP	associative	heuristic-based, replay	manual

We started the survey by describing PRIAR, one of the first domain independent systems implementing the case-based approach. The domain independence is a bit disputable for PRIAR, as the validation structure used by the planner embeds some domain knowledge in a way similar to Hierarchical Task Networks. However, the algorithm as such works in any domain and hence we place PRIAR among domain-independent systems.

A similar approach (to transform the stored solution) was taken later by SPA, but the form of the stored solution is different (a plan with causal links and ordering constraints in place of the validation structure) and the system does not store any domain-dependent knowledge in any of its internal structures.

We surveyed MRL to highlight the variety of different formalisations in CBP. In MRL, planning problems are represented as formulae in temporal logic LLP and it uses a deductive planning system PHI to construct a proof that the goals can be reached from the initial state. Such a proof is then the solution plan.

PRODIGY/ANALOGY reuses the reasoning steps that previously led to finding a solution rather than the actual plans, representing the *derivational replay* as a technique complementary to the *transformational* approach implemented by the previously described systems. In addition, PRODIGY/ANALOGY was the first system able to retrieve several cases at once and to combine them during reuse.

With the success of heuristic approaches for generative planning, the CBP systems performing transformational plan reuse started to reappear. Similarly to PRODIGY/ANALOGY and CABALA, also FAR-OFF can retrieve several cases to reuse, but it does so only if the most suitable candidate failed to be modified to solve the current problem. FAR-OFF tries to find a plan to connect the initial state of the current problem with the case's initial state and concatenates the plan stored in the case. Then it only needs to find a plan from the final state of the case to a state that satisfies the goals of the current problem. FAR-OFF saves significant effort on identifying the reusable parts of the stored solution to improve the performance, even though it lowers the quality of the new solution plans.

CABALA is a recent system that stores cases to guide the search for a new solution plan. Hence it falls among derivational planners together with PRODIGY/ANALOGY. Differently from PRODIGY/ANALOGY, CABALA stores plan traces for every object of the planning problems. Consequently, it retrieves several cases to address the current problem.

OAKPLAN is a contemporary transformational system, which addresses the retrieval problem by graph-based heuristics and the plan reuse by a local search in the space of plans started from the retrieved plan.

Starting from FAR-OFF, the reuse phase is based on forward search heuristic planners, similar to FF. All surveyed systems before the end of the nineties used backward search planners (PRODIGY/ANALOGY, SPA) or hierarchical planners (PRIAR). Thus, the plan adaptation steps were not guided by heuristics, and planning time was huge even in *simple* problems. Using forward search heuristic planners dramatically changed the size of planning problems that could be solved, and thus planners coverage. So, CBP approaches such as FAR-OFF, CABALA, or OAKPLAN benefit from this boost in planners coverage.

Table III. Other features - retrieval and matching.

system	mapping	similarity evaluation
PRIAR	partial unification of $\mathcal{I}, \mathcal{G}$	number and type of inconsistencies in the validation structure
SPA	partial unification of $\mathcal{I}, \mathcal{G}$	number of open preconditions
MRL	relations between initial and goal states	number of matching goals
PRODIGY/ANALOGY	unification	number of matching goals and their interactions
FAR-OFF	none	ADG similarity metric
CABALA	based on types of objects	matching degree among the initial state and goals sub-states; relaxed plans
OAKPLAN	kernel functions	matching degree among the initial state, goals; relaxed plans
SIN	similarity of the state and case's preconditions	case's preference pairs
ML-CBP	partial unification of $\mathcal{I}, \mathcal{G}$	frequent sequential pattern mining

Table IV. Other features - retrieval and reuse.

system	filtering	reuse engine
PRIAR	estimated adaptation effort	NONLIN
SPA	number of matching goals	SNLP
MRL	mapping the plan specification to the concept descrptions	PHI
PRODIGY/ANALOGY	subset of goals, partial match of initial state features	PRODIGY
FAR-OFF	footprint elements	FF
CABALA	one case per type	Sayphi
OAKPLAN	degree sequences	LPG-ADAPT
SIN	applicability of a case	SHOP
ML-CBP	utility of plan fragment	none

An interesting direction in CBP is represented by the last two described systems which suggest to use the knowledge stored in the case base not only as *control knowledge* (to guide the search for a new solution), but also as *domain knowledge*, where the cases substitute for missing or incomplete domain descriptions.

SIN is an approach based on Hierarchical Task Networks (similarly to PRIAR). It is also more interactive than the previous systems as it uses *conversations* with the user in order to obtain more details about the current state. The idea of using the cases for task decomposition was further examined in [Xu and Muñoz Avila 2005].

Lastly, we surveyed ML-CBP, which is specifically designed to plan with an incomplete domain theory. Despite the different settings, the system still addresses the phases of the case-based methodology; the retrieval is guided by the utility of the stored plans and allows retrieving multiple cases, which are then combined into a solution plan.

Tables III and IV compare several implementation choices made in the studied planners — we focus on the presence of a procedure for matching the objects and the mapping it uses, the similarity function used by the retrieval step (Table III), the implementation of the filtering step during retrieval and the generative engine underlying the reuse of the retrieved solution (Table IV).

11.2. Applications of CBP

As we have discussed in the introduction, we believe CBP has a huge potential in applications of planning technology. Here, we review some examples of recent applications of CBP. Among them, two domains have lately received some attention: workflow management and Real-Time Strategy (RTS) games. Games present several new challenges compared to traditional planning domains: the state space and decision space are huge; the world is only partially observable and non-deterministic (the player can sense only part of the world and the game may include unpredictable opponents); and they are real-time, so the time for planning or replanning is very limited.

Simpler models of planning have been used together with CBR [Aha et al. 2005] to provide tactics for RTS games, as in WARGUS, a clone of WARCRAFT II. In the context of the same game, several issues related to CBP have been investigated in [Dannenhauer and Muñoz-Avila 2013; Finestrali and Muñoz-Avila 2013; Jaidee et al. 2013]. In particular, DARMOK [Ontañón et al. 2010] is a CBP system that focuses on real-time. So, it needs to address several issues most of the planners we discussed in this paper did not consider explicitly; e.g., the dynamics of the world during the execution or the high complexity of the RTS domains. The authors propose an extension of the CBR cycle to allow for delayed adaptation to postpone the reuse steps until just before their execution, so that the state of the world is as close to reality as possible. The authors of DARMOK have also created a middleware platform, MAKEMEPLAYME, to help on the use of AI techniques (as learning and planning) for game playing [Gómez-Martín et al. 2010]. Their approach is game-independent, though not clearly fully domain-independent, since it only seems to work for specific kinds of games. Since it learns the actions, it might generate incomplete domains, leading to unsound plans. Also, the engineering of their domain models seems to be slightly more complex than in PDDL, since they augment the types of conditions on actions. This kind of CBP approach is well suited for planning in dynamic environments such as games. It would be interesting to compare the knowledge engineering effort and the validity of the resulting planning systems using a standard PDDL-based approach and MAKEMEPLAYME.

A recent interesting concept related to planning in real-time applications (such as games) is Goal Driven Autonomy (GDA) [Molineaux et al. 2010]. In a planning and execution scenario where plans are not executed until the end due to unexpected events, GDA proposes to dynamically change goals driven by execution failures and explanations of those failures. There have been some works that apply CBP to games using a GDA approach [Muñoz-Avila et al. 2010]. A related approach performs a meta-reasoning step on the planning and execution cycle as in META-DARMOK [Mehta et al. 2009].

In the last decade, different approaches successfully took advantage of the close relationship between AI planning and workflow technologies used for modeling and managing the execution of tasks; in fact, the requirements of formal planning models apply also well to workflow. In particular, CBR and CBP have been effectively applied to workflows generation. The CODAW system [Madhusudan et al. 2004] allows the user to incrementally model and compose workflows using an HTN planner. The system stores two types of cases: *prototypical* cases that encode workflow schemas and *instance-level* cases that correspond to instantiations of prototypical cases. PROCHIP [Macedo and Cardoso 2004] is another example of the combination of CBR and HTN.

The CABMA system [Xu and Muñoz-Avila 2004] uses case-based HTN planning techniques in order to assist the users with their project plans. A case is defined as a generalized version of a *Work Breakdown Structure* which consists of a task, a set of subtasks decomposing it, an ordering relation between the subtasks and a set of resources necessary for applying the case. Since there exists a one-to-one correspondence between WBS and HTN structures [Xu and Muñoz-Avila 2004], it is possible to use HTN planning techniques in order to fix the inconsistences relative to the reused cases.

Other explored domains are manufacturing and e-learning. CAPLAN/CBC [Muñoz-Avila and Weberskirch 1996] uses derivational replay with a SNLP-like [McAllester and Rosenblitt 1991] generative planner to solve the task of manufacturing workpieces. It was designed to behave optimally in this specific domain. In [Garrido et al. 2012], the authors describe an approach to convert e-learning contents and student preferences and goals into planning domains and problems expressed in PDDL. Even though these problems can be solved by any planner, the authors encourage using CBP techniques as in the e-learning domain it is desired to produce plans closely resembling those already used, because the previously approved solutions comply with additional set of requirements coming from e.g. pedagogy or other restrictions on the course organisation.

11.3. Future Work

It seems that the efficiency of the CBP systems depends on the implemented heuristics in retrieval and reuse rather than on the theoretically fundamental choice of derivational or transformational plan reuse, provided that the later one is not implemented in a conservative manner. The importance of good heuristics for plan reuse is also supported by theoretical evidence, which defines one of the future directions CBP community should try to investigate. Other interesting open problems related to the implementation include integration with current generative planners, integration with portfolios, reuse systems designed for the run on multi-core machines and, in general, development of better planning by reuse systems. Another interesting direction for CBP consists on focusing the research to settings in which experience plays an essential role and therefore calls for some form of experience reuse. An example of such settings can be problems with incomplete domain knowledge or problems where an approval of human experts is crucial. In the last section, we have encountered a number of domains fulfilling these criteria, where the use of CBP lead to satisfactory performance.

To conclude this paper we would like to claim that the field of CBP still evolves and, even though it is not a mainstream approach, there are open questions and research challenges worth addressing. We believe that the continuous research in this area may result in a good alternative to generative planners and in some domains may even lead to better results than use of first principle solvers. Additional attention should be paid to theoretical issues as well as to application of generative planning techniques and heuristics to CBP, as many of the described systems show that such an influence can be beneficial.

References

A. Aamodt and E. Plaza. 1994. Case-based reasoning: foundational issues, methodological variations, and system approaches. *AI Communications* 7, 1, 39–59.

D.W. Aha, M. Molineaux, and M.J.V. Ponsen. 2005. Learning to Win: Case-Based Plan Selection in a Real-Time Strategy Game. In *Proc. of ICCBR (LNCS)*, Vol. 3620. Springer, 5–20.

R. Alterman. 1990. An Adaptive Planner. In *Readings in Planning*, J. Allen, J. Hendler, and A. Tate (Eds.). Kaufmann, 660–664.

T. Au, H. Muñoz-Avila, and D. S. Nau. 2002. On the Complexity of Plan Adaptation by Derivational Analogy in a Universal Classical Planning Framework. In *Proc. of ECCBR*. Springer, 13–27.

C. Bäckström, Y. Chen, P Jonsson, S. Ordyniak, and S. Szeider. 2012. The Complexity of Planning Revisited - A Parameterized Analysis. In *Proc. of AAAI*. AAAI Press, 1735–1741.

C. Bäckström and B. Nebel. 1995. Complexity Results for SAS+ Planning. *Computational Intelligence* 11, 4, 625–655.

B. Bauer, S. Biundo, D. Dengler, J. Koehler, and G. Paul. 1993. PHI - A Logic-Based Tool for Intelligent Help Systems. In *Proc. of IJCAI*. 460–466.

B. Bonet and H. Geffner. 2001. Planning as Heuristic Search. *Artificial Intelligence* 129, 1-2, 5–33.

A. J. Bonner and M. Kifer. 1995. *Transaction Logic Programming (or a Logic of Declarative and Procedural Knowledge)*. Technical Report CSRI-323. University of Toronto.

D. Borrajo and M. M. Veloso. 2012. Probabilistically Reusing Plans in Deterministic Planning. In *Preprints of Workshop on Heuristics and Search for Domain-Independent Planning (ICAPS)*. 17–25.

L. Breslow and D. Aha. 1998. *NaCoDAE: Navy Conversational Decision Aids Environment*. Technical Report AIC-97-018. Research Laboratory, Navy Center for Applied Research in Articial Intelligence.

T. Bylander. 1994. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence* 69, 1-2, 165–204.

D. Chapman. 1987. Planning for conjunctive goals. *Artificial Intelligence* 32, 3, 333–377.

M. T. Cox, H. Muñoz Avila, and R. Bergmann. 2005. Case-Based Planning. *The Knowledge Engineering Review* 20, 03, 283–287.

D. Dannenhauer and H. Muñoz-Avila. 2013. Case-Based Goal Selection Inspired by IBM's Watson. In *Proc. of ICCBR (LNCS)*, Vol. 7969. Springer, 29–43.

R. de Haan, A. Roubíčková, and S. Szeider. 2013. Parameterized Complexity Results for Plan Reuse. In *Proc. of AAAI*. AAAI Press, 224–231.

T. de la Rosa, A. García-Olaya, and D. Borrajo. 2013. A Case-Based Approach to Heuristic Planning. *Applied Intelligence* 39, 1, 184–201.

R. L. de Mántaras. 2001. Case-Based Reasoning. In *Machine Learning and Its Applications*. Springer, 127–145.

K. Erol, J. Hendler, and D. S. Nau. 1994. HTN Planning: Complexity and Expressivity. In *Proc. of AAAI*, Vol. 94. AAAI Press, 1123–1128.

R. E. Fikes and N. J. Nilsson. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 2, 3/4, 189–208.

G. Finestrali and H. Muñoz-Avila. 2013. Case-Based Learning of Applicability Conditions for Stochastic Explanations. In *Proc. of ICCBR (LNCS)*, Vol. 7969. Springer, 89–103.

M. Fox, A. E. Gerevini, D. Long, and I. Serina. 2006. Plan Stability: Replanning versus Plan Repair. In *Proc. of ICAPS*. AAAI Press, 212–221.

A. Garrido, L. Morales, and I. Serina. 2012. Using AI Planning to Enhance E-Learning Processes. In *Proc. of ICAPS*. AAAI Press, 47–55.

A. E. Gerevini, P. Haslum, D. Long, A. Saetti, and Y. Dimopoulos. 2009. Deterministic Planning in the Fifth Int. Planning Competition: PDDL3 and Experimental Evaluation of the Planners. *Artificial Intelligence* 173, 5, 619–668.

A. E. Gerevini, A. Roubíčková, A. Saetti, and I. Serina. 2013a. Offline and Online Plan Library Maintenance in Case-based Planning. In *Proc. of AI*IA*. Springer, 239–250.

A. E. Gerevini, A. Roubíčková, A. Saetti, and I. Serina. 2013b. On the Plan-library Maintenance Problem in a Case-based Planner. In *Proc. of ICCBR (LNCS)*, Vol. 7969. Springer, 119–133.

A. E. Gerevini, A. Saetti, and I. Serina. 2012. Case-based Planning for Problems with Real-valued Fluents: Kernel Functions for Effective Plan Retrieval. In *ECAI*. IOS Press, 348–353.

M. Ghallab, D. Nau, and P. Traverso. 2004. *Automated Planning. Theory & Practice*. Morgan Kaufmann.

P. P. Gómez-Martín, D. Llansó, M. A. Gómez-Martín, S. Ontañón, and A. Ram. 2010. MMPM: A Generic Platform for Case-Based Planning Research. In *Workshop on Case-Based Reasoning for Computer Games (ICCBR)*. 45–54.

K. Z. Haigh, J. Shewchuk, and M. M. Veloso. 1997. Exploring geometry in analogical route planning. *Journal of Experimental and Theoretical Artificial Intelligence* 9, 509–541.

K. J. Hammond. 1990. Case-Based Planning: A Framework for Planning from Experience. *Cognitive Science* 14, 3, 385–443.

S. Hanks and D.S. Weld. 1995. A Domain-Independent Algorithm for Plan Adaptation. *Journal of Artificial Intelligence Research* 2, 319–360.

J. Hoffmann and B. Nebel. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14, 253–302.

U. Jaidee, H. Muñoz-Avila, and D. W. Aha. 2013. Case-Based Goal-Driven Coordination of Multiple Learning Agents. In *Proc. of ICCBR (LNCS)*, Vol. 7969. Springer, 164–178.

S. Jiménez, T. de la Rosa, S. Fernández, F. Fernández, and D. Borrajo. 2012. A Review of Machine Learning for Automated Planning. *Knowledge Engineering Review* 27, 4, 433–467.

S. Kambhampati and J. A. Hendler. 1992. A Validation-Structure-Based Theory of Plan Modification and Reuse. *Artificial Intelligence* 55, 193–258.

S. Kambhampati and B. Srivastava. 1996. *Unifying Classical Planning Approaches*. Technical Report. Arizona State University ASU CSE TR. 96–106 pages.

J. Köhler. 1996. Planning from Second Principles. *Artificial Intelligence* 87, 1-2, 145–186.

V. Kuchibatla and H. Muñoz-Avila. 2006. An Analysis on Transformational Analogy: General Framework and Complexity.. In *Proc. of ECCBR (LNCS)*, Vol. 4106. Springer, 458–473.

D. B. Leake. 1996. *Case-Based Reasoning*. The MIT Press.

D. B. Leake and D. C. Wilson. 1998. Categorizing Case-Base Maintenance: Dimensions and Directions. In *Proc. of 4th European Workshop on Case-Based Reasoning*. Springer, 196–207.

P. Liberatore. 2005. On the complexity of case-based planning. *Journal of Experimental & Theoretical Artificial Intelligence* 17, 3, 283–295.

M. Likhachev and S. Koenig. 2005. A Generalized Framework for Lifelong Planning A* Search. In *ICAPS*. AAAI Press, 99–108.

D. Long and M. Fox. 2003. The 3rd International Planning Competition: Results and Analysis. *Journal of Artificial Intelligence Research* 10, 1–59.

L. Macedo and A. Cardoso. 2004. Case-Based, Decision-Theoretic, HTN Planning. In *Proc. of ECCBR*. LNCS, Vol. 3155. Springer, 257–271.

T. Madhusudan, J. L. Zhao, and B. Marshall. 2004. A Case-Based Reasoning Framework for Workflow Model Management. *Data & Knowledge Engineering* 50, 1, 87–115.

D. McAllester and D. Rosenblitt. 1991. Systematic Nonlinear Planning. In *Proc. of AAAI*. AAAI Press, 634–639.

M. Mehta, S. Ontañón, and A. Ram. 2009. Using Meta-reasoning to Improve the Performance of Case-Based Planning. In *Proc. of ICCBR (LNCS)*, Vol. 5650. Springer, 210–224.

M. Molineaux, M. Klenk, and D.W. Aha. 2010. Goal-Driven Autonomy in a Navy Strategy Simulation. In *Proc. of AAAI*. AAAI Press, 1548–1554.

H. Muñoz Avila, D. W. Aha, L. Breslow, and D. Nau. 1999. HICAP: An Interactive Case-Based Planning Architecture and its Application to Noncombatant Evacuation Operations. In *Proc. of IAAI*. AAAI Press, 870–875.

H. Muñoz-Avila. 2001. Case-Base Maintenance by Integrating Case-Index Revision and Case-Retention Policies in a Derivational Replay Framework. *Computational Intelligence* 17, 2, 280–294.

H. Muñoz-Avila, D.W. Aha, D.S. Nau, L.A. Breslow, R. Weber, and R. Yamal. 2001. SiN: Integrating Case-Based Reasoning with Task Decomposition. In *Proc. of IJCAI*. 999–1004.

H. Muñoz-Avila and M. T. Cox. 2008. Case-Based Plan Adaptation: An Analysis and Review. *IEEE Intelligent Systems* 23, 4, 75–81.

H. Muñoz-Avila, U. Jaidee, D. W. Aha, and E. Carter. 2010. Goal-Driven Autonomy with Case-Based Reasoning. In *Proc. of ICCBR (LNCS)*, Vol. 6176. Springer, 228–241.

H. Muñoz-Avila and F. Weberskirch. 1996. Planning for Manufacturing Workpieces by Storing, Indexing and Replaying Planning Decisions. In *Proc. of AIPS*. AAAI Press, 150–157.

D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. 1999. SHOP: Simple hierarchical ordered planner. In *Proc. of IJCAI*. 968–973.

B. Nebel and J. Köhler. 1995. Plan Reuse Versus Plan Generation: A Complexity-Theoretic Perspective. *Artificial Intelligence* 76, 427–454.

S. Ontañón, K. Mishra, N. Sugandh, and A. Ram. 2010. On-Line Case-Based Planning. *Computational Intelligence* 26, 1, 84–119.

F. Ruskey, R. Cohen, P. Eades, and A. Scott. 1994. Alley CATs in Search of Good Homes. *Twenty-fifth Southeastern Conference on Combinatorics, Graph Theory and Computing* 102, 97–110.

B. Scholkopf and A. J. Smola. 2001. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA.

I. Serina. 2010. Kernel Functions for Case-Based Planning. *Artificial Intelligence* 174, 16-17, 1369–1406.

B. Smyth and E. McKenna. 1999. Footprint-Based Retrieval. In *Proc. of ICCBR*. AAAI Press, 343–357.

B. Smyth and E. McKenna. 2001. Competence Models and the Maintenance Problem. *Computational Intelligence* 17, 2, 235–249.

L. Spalazzi. 2001. A Survey on Case-Based Planning. *Artificial Intelligence Review* 16, 1, 3–36.

A. Tate. 1977. Generating project networks. In *Proc. of IJCAI*. 888–889.

F. Tonidandel and M. Rillo. 2002. The FAR-OFF System: A Heuristic Search Case-Based Planning. In *Proc. of AIPS*. AAAI Press, 302–311.

R. van der Krogt and M. de Weerdt. 2005. Plan Repair as an Extension of Planning.. In *Proc. of ICAPS*. AAAI Press, 161–170.

M. Veloso, J. Carbonell, A. Pérez, D. Borrajo, E. Fink, and J. Blythe. 1995. Integrating Planning and Learning: The PRODIGY Architecture. *Journal of Experimental and Theoretical AI* 7, 81–120.

M. M. Veloso. 1994. *Planning and Learning by Analogical Reasoning*. LNCS, Vol. 886. Springer.

M. M. Veloso. 1997. Merge strategies for multiple case plan replay. In *Proc. of ICCBR*. Springer, 413–424.

M. M. Veloso and J. G. Carbonell. 1993. Derivational Analogy in PRODIGY: Automating Case Acquisition, Storage, and Utilization. *Machine Learning* 10, 249–278.

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

K. Xu and H. Muñoz Avila. 2005. A Domain-Independent System for Case-Based Task Decomposition without Domain Theories. In *Proc. of AAAI*. AAAI Press, 234–239.

K. Xu and H. Muñoz-Avila. 2004. CaBMA: Case-Based Project Management Assistant. In *Proc. of AAAI*. AAAI Press / The MIT Press, 931–936.

M. Zaki. 2001. SPADE: An Efficient Algorithm for Mining Frequent Sequences. *Machine Learning* 42, 1-2, 31–60.

H. H. Zhuo, S. Kambhampati, and T. A. Nguyen. 2012. Model-Lite Case-Based Planning. *CoRR* abs/1207.6713.

T. Zimmerman and S. Kambhampati. 2003. Learning-Assisted Automated Planning: Looking Back, Taking Stock, Going Forward. *AI Magazine* 24, 2, 73–96.

Received February 2014; revised May 2014; accepted September 2014