

# Knowledge Transfer between Automated Planners

*Susana Fernández, Ricardo Aler, and Daniel Borrajo*

■ *In this article, we discuss the problem of transferring search heuristics from one planner to another. More specifically, we demonstrate how to transfer the domain-dependent heuristics acquired by one planner into a second planner. Our motivation is to improve the efficiency and the efficacy of the second planner by allowing it to use the transferred heuristics to capture domain regularities that it would not otherwise recognize. Our experimental results show that the transferred knowledge does improve the second planner's performance on novel tasks over a set of seven benchmark planning domains.*

In the context of solving difficult tasks, humans naturally transfer problem-solving experience from previous tasks into the new task. Recently, the artificial intelligence community has attempted to model this transfer in an effort to improve learning on new tasks by using knowledge from related tasks. For example, classification and inference algorithms have been extended to support transfer of conceptual knowledge (for a survey see Torrey and Shavlik [2009]). Likewise, reinforcement learning has also been extended to support transfer (for a survey see Taylor and Stone [2009]; Torrey and Shavlik [2009]).

This article presents an attempt to transfer structured knowledge in the framework of automated planning. Automated planning is the branch of artificial intelligence that studies the computational synthesis of ordered sets of actions that perform a given task (Ghallab, Nau, and Traverso 2004). A planner receives as input a collection of actions (that indicate how to modify the current state), a collection of goals to achieve, and a state. It then outputs a sequence of actions that achieve the goals from the initial state. Given that each action transforms the current state, planners may be viewed as searching for paths through the state space defined by the given actions. However, the search spaces can quickly become intractably large, such that the general problem of automated planning is PSpace-complete (Bylander 1994).

The most common approach to coping with planning complexity involves defining heuristics that let the planner traverse the search space more efficiently. Current state-of-the-art planners use powerful domain-independent heuristics (Ghallab, Nau, and Traverso 2004; Nau 2007). These are not always sufficient, however, so an important research direction consists of defining manually or learning automatically domain-dependent heuristics (called control knowledge). In the latter case,

domain-independent machine-learning techniques acquire the domain-dependent knowledge used to guide the planner. The goal of transfer then is to apply the experience gained in solving simple source tasks to improve the performance of the same planner in more complex target tasks (Borrajo and Veloso 1997).

Our work takes the idea one step further: we study the use of heuristics learned on one planner for improving the performance of a different planner. Importantly, the two planners use different biases and strategies to search through the problem space. We consider the domain-independent heuristics part of the planner biases, while the learned heuristics provide extra information that let the planner overcome its biases and improve the search. Although different planners use different strategies to search for solutions, some planners share commonalities that make transfer possible. In general, domain-dependent heuristics can be learned for a planner by observing how it traverses the search space in a set of problems and detecting opportunities for improvement. For instance, a system could learn shortcuts that let the planner find solutions faster, or it could obtain knowledge to avoid dead ends that restrain the planner from finding the solution. However, detecting opportunities for improvement depends very much on the way the planner explores its search space. We hypothesize that heuristics learned from one planner can be transferred to another planner, thereby improving the latter's performance compared to learning heuristics from scratch. Thus, the aim of this article is to explore the transfer of heuristics between two planners that share similarities in the metaproblem spaces while using different biases to explore them.

In particular, we focus on transfer of control knowledge between two planners: from TGP (Smith and Weld 1999) to IPSS (Rodríguez-Moreno et al. 2006). TGP is based on Graphplan (Blum and Furst 1995) and so searches a reachability graph with a bias toward parallel plans. Conversely, IPSS derives from PRODIGY (Veloso et al. 1995) and represents a class of state-based backward-chaining planners that produce sequential plans. It also has the advantage of supporting heuristics represented in a declarative language. The planners overlap in their use of backward-chaining search, but differ in their search strategies (reachability graph versus state-based search) and in the types of plans that they produce (parallel versus sequential).

Our goal is to transfer the heuristics learned by TGP (such as favoring parallel plans) into IPSS in hopes of producing a bias that neither system could have created independently. Zimmerman and Kambhampati (2003) discuss many ways to represent heuristics in planning, such as control rules, cases, or policies. Our approach uses control

rules that guide the planner's decisions by recommending steps to take based on recognized conditions. More specifically, our system generates these control rules automatically by applying machine-learning techniques in TGP, and then translating them into the IPSS control-knowledge description language. We selected to transfer from TGP to IPSS, because both are backward-chaining planners (so that planning concepts and search spaces are similar) but they use different search biases, allowing to test our hypothesis that transferring heuristics from one planner to another one using different search biases can help the second planner.

This article is organized as follows. The next section provides background on the planning techniques involved such as state- and graph-based planning. After that, we describe the major elements of transfer, such as knowledge acquisition, mapping between planning systems, and knowledge refinement. In the remaining sections we present experimental results, related work, and conclusions and future work.

## Planning Models and Techniques

In general, two elements typically define a planning task: a *domain definition* composed of a set of states and a set of actions; and a *problem description* composed of the initial state and a set of goals that the planner must achieve. Here, a state description includes a collection of grounded predicates and functions, while an action includes the parameters or typed elements involved in its execution, a set of preconditions (a list of predicates describing the facts that must hold for the action to apply), and a set of effects (a list of changes to the state that follow from the action). The planning task consists of obtaining a partially ordered sequence of actions that achieve all of the goals when executed from the initial state. Each action modifies the current state by adding and deleting the predicates represented in the domain-action effects. The following example defines a planning problem in the Zenotravel domain.

### Example 1.

The Zenotravel domain involves transporting people among cities in planes, using different modes of flight: fast and slow. The domain includes five actions: (1) (board ?person ?plane ?city) boards a person on a plane in a specified city, (2) (debark ?person ?plane ?city) a person disembarks from a plane in a specified city, (3) (fly ?plane ?city0 ?city1 ?fuel0 ?fuel1) moves the plane from one city to another at slow speed and consumes one level of fuel, (4) (zoom ?plane ?city0 ?city1 ?fuel0 ?fuel1 ?fuel2) moves the plane from one city to another at fast speed and consumes two levels of fuel, and (5) (refuel ?plane ?city ?fuel0 ?fuel1) increases the plane's fuel level.

In example 1, the question marks indicate the pred-

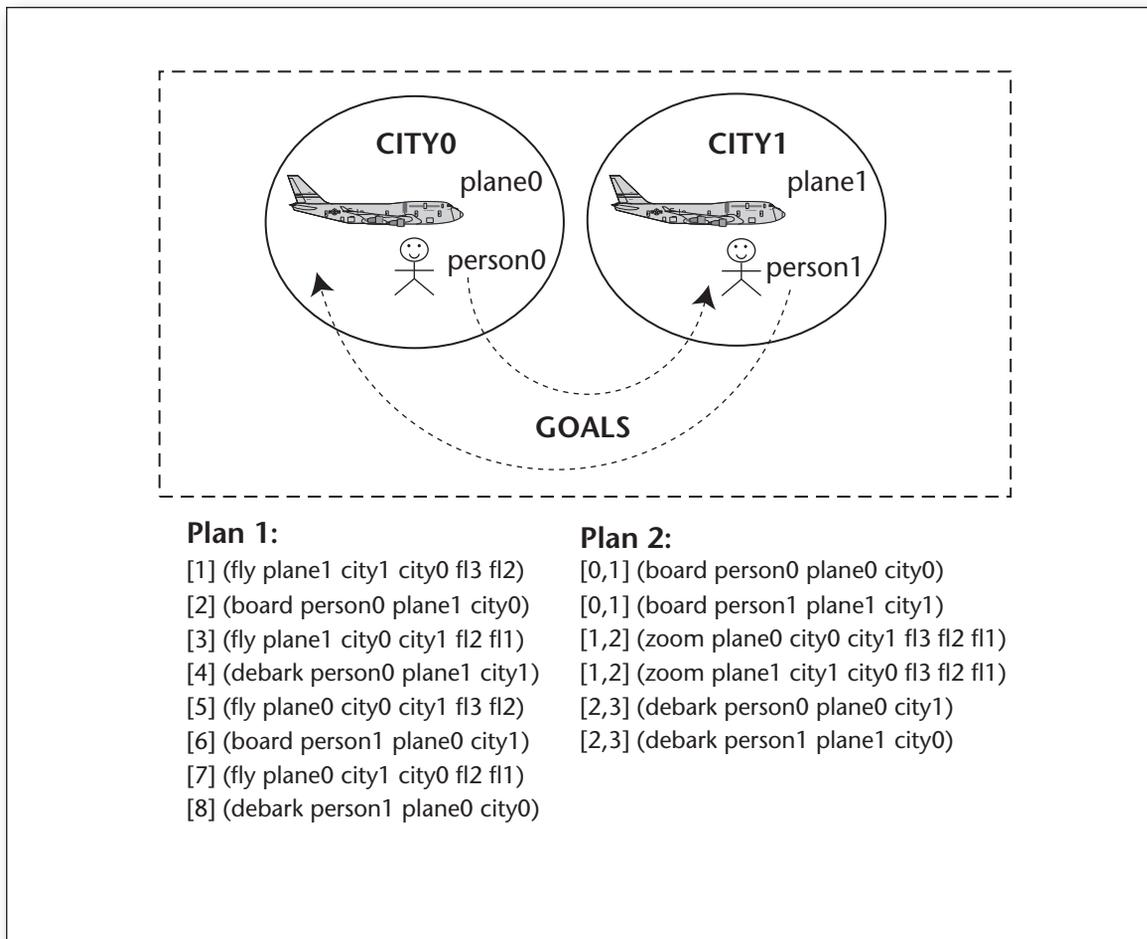


Figure 1. Problem Example in the Zenotravel Domain.

icate variables, which the planner must bind to domain constants to create specific action instances. The example problem consists of transporting two persons: person0 from city0 to city1, and person1 from city1 to city0. There are seven fuel levels (fl<sub>i</sub>) ranging from 0 to 6 and there are two planes initially at city1 and city0 with a fuel level of fl<sub>3</sub>. Figure 1 shows the problem and two possible solution plans. For example, the first action in Plan 1 represents that plane1 flies from city1 to city0 and the fuel level varies from fl<sub>3</sub> to fl<sub>2</sub>.

To transfer heuristics from one planner to another, we must define the planning task in terms of a unified model. This model must account for the important aspects of planning from a perspective of learning and using control knowledge, that is, how planners traverse the search space so that the heuristics can improve the search. The unified model requires two definitions: a domain problem space and a metaproblem space, which we now provide.

**Definition 1.**

A domain problem space  $P$  is defined as a tuple  $\langle S, s_0, G, A, T \rangle$ , where:

$S$  is a finite collection of states.

$s_0 \in S$  is an initial state.

$G$  is a collection of goals.

$A$  is a collection of actions.

$T: S \times A \rightarrow S$  is a transition function that maps states and actions into a state.

A solution plan for this problem is an ordered set of actions,  $(a_0, a_1 \dots a_n)$ , that transforms the initial state,  $s_0$ , into a terminal state,  $s_{n+1}$ , in such a way that  $s_{i+1} = T(s_i, a_i)$  ( $i \in [0, n]$ ) and  $G \subseteq s_{n+1}$  (goals are true in the end state).  $T$  is defined through the effects of the actions in  $A$  that are defined in the planning domain.

**Example 2.**

Continuing with the Zenotravel domain, a state in  $S$  is described by a conjunction of literals, which are simply grounded predicates. For Zenotravel, the predicates include: (at ?p ?c) a person is in a city, (at ?a ?c) a plane is in a city, (in ?p ?a) a person is inside a plane, (fuel-level ?a ?l) a plane has a level of fuel, (next ?l1 ?l2) two consecutive levels of fuel.

The initial state is  $s_0 = \{(at\ person0\ city0), (at\ person1\ city1), (at\ plane1\ city1), (at\ plane0\ city0), (fuel-level\ plane1\ fl3), (fuel-level\ plane0\ fl3), (next\ fl0\ fl1), (next\ fl1\ fl2) \dots (next\ fl5\ fl6)\}$ . The goals are

$G = \{(at\ person1\ city0), (at\ person0\ city1)\}$ . Figure 1 shows two possible plans for satisfying the example goal given the start state. The first plan represents a total ordering over the solution and corresponds to the type of output that planners such as IPSS might generate. The second plan represents a partial ordering over the solution and corresponds to output of planners like TGP. Notice that the solutions are not equivalent, as Plan 1 requires additional steps for swapping the initial locations of the two planes.

Although planners could search directly in problem spaces such as the one used in example 2, in practice they tend to derive metaproblem spaces. While a problem space models the domain, a metaproblem space models the search process performed by a particular planner. This is similar to the notion of branching described by Geffner (2001), which is a scheme used for generating search nodes during plan construction.

**Definition 2.**

A metaproblem space  $M_p$  is defined as a tuple  $\langle M, m_0, M_T, O, F \rangle$ , where:

$M$  is a finite collection of metastates.

$m_0 \in M$  is an initial metastate.

$M_T \subseteq M$  is a collection of terminal metastates.

$O$  is a set of search operators, such as “apply an action  $a \in A$  to the current state  $s \in S$ ” or “select an action  $a \in A$  for achieving a given goal  $g \in G$ ,” where  $S, A$ , and  $G$  are elements of the problem space  $P$ .

$F : M \times O \rightarrow M$  is a function mapping metastates  $m$  and search operators  $o$  into a metastate.

Metastates are planner dependent and include all the knowledge that the planner needs for making decisions during its search process. Usually each metastate  $m$  contains the current state  $s$  ( $s \in S$  in the problem-space definition), but it can contain other elements needed for the search such as the pending (unachieved) goals. The following two subsections provide examples of metaproblem spaces used by TGP and IPSS for the Zenotravel task discussed in examples 1 and 2.

### TGP Planner

TGP (Smith and Weld 1999) enhances the Graphplan algorithm temporally to support actions with different durations. Although in this work, we use only TGP’s planning component so that the resulting system is equivalent to Graphplan, we prefer TGP because it supports future extensions to tasks with a temporal component. TGP alternates between two phases during planning. The first phase, graph expansion, extends the planning graph until it has achieved necessary, though possibly insufficient, conditions for plan existence (that is, when all of the problem goals appear in the graph for the first time and none are pairwise mutually exclusive). The second phase, solution extraction, performs a backward-chaining search

on the planning graph for a solution. If the search cannot identify a solution, then the system begins a new round of graph expansion and solution extraction.

A *planning graph* is a directed, leveled graph that encodes the planning problem. The graph levels correspond to time steps in the domain such that each time step gets associated with two levels in the graph. The first level for a time step contains a node for each fact true in that time step, while the second contains a node for each action whose precondition is satisfied in the time step. For example, fact level 0 corresponds to facts (grounded predicates) true in the initial state, while action level 0 corresponds to the applicable actions in  $s_0$ . Subsequent levels are similar, except that the nodes indicate which facts may be true given the execution of the actions in the preceding level, and which actions may be applicable given the projected facts. Specifically, the edges point either from a fact to the actions for which it is a precondition, or from an action to the facts whose truth values it changes. TGP also computes lists of mutually exclusive facts and mutually exclusive actions based on the domain description.

Using the planning graph and the lists of mutually exclusive facts and actions, TGP now conducts a backward search through the metaproblem space. The planner begins with the goal state, which it identifies as the lowest level that contains all of the problem goals with no pair of goals on the mutual exclusion list. In that search, a set of (sub)goals  $g$  at level  $i$  yields a new set of subgoals  $g'$  at level  $i - 1$ , by first selecting one action for each goal in  $g$  and then adding to  $g'$  the actions’ preconditions. The search terminates successfully when the initial level is reached, or unsuccessfully when all options have been explored. In the latter case TGP extends the graph another level, and then continues the search. When the search from a subgoal set  $g$  at layer  $i$  fails,  $g$  is memorized or remembered as unsolvable at  $i$ .

Figure 2 shows a simplified planning graph of the problem in example 1. We display action nodes only in the level where the action is compatible (not mutually exclusive) with the other actions in the solution plan. For example, the action instance (zoom plane0 city0 city1 fl3 fl2 fl1) is displayed in level 1 instead of in level 0 because it is mutually exclusive with the action instance (board person0 plane0 city0) (they cannot be executed at the same time). A proposition in gray color represents a condition of an action that appears in a lower graph level. Nodes that are not part of the solution plan are not displayed.

In terms of the unified model, the backward-chaining search on the planning graph that TGP performs during the solution-extraction phase can be described as follows.

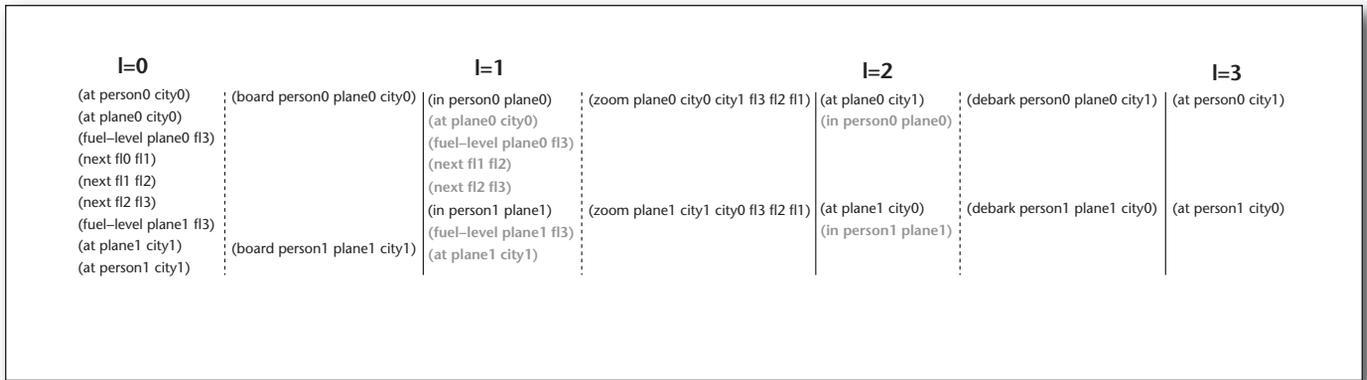


Figure 2. Simplified Planning Graph of the Problem in Example 1.

Each metastate  $m$  is a tuple  $\{PG, G_p, L, l\}$  where  $PG$  is the plan graph,  $G_p$  is the set of pending (unsolved) goals,  $L$  is a set of links  $(a, g)$  representing that the action  $a$  was used to achieve the goal  $g$  (current partial plan), and  $l$  is the current plan-graph level of the search.

The initial metastate,  $m_0 = \{PG_n, G, \emptyset, l_n\}$  where  $PG_n$  is the plan graph built in the first phase up to level  $n$  (this second phase can be called many times),  $G$  is the set of top-level goals, and  $l_n$  is the last level generated of the plan graph (backward-chaining search).

A terminal metastate will be of the form  $M_T = \{PG_n, \emptyset, L, 0\}$ , that is, the search reaches the initial state (level 0) and all goals are solved (the set of pending goals is empty). The solution plan can be obtained from the actions contained in  $L$ .

The set of search operators  $O$  is composed of only one operator (given the current metastate  $\{PG, G_p, L, l\}$ ): “for each goal  $g \in G_p$  select (assign) an action  $a \in (A \cup no-op)$  for achieving it.” If  $a$  is the empty operator, also named  $no-op$ , the goal persists, that is, the goal will still be in the  $G_p$  of the successor metastate. Otherwise, the preconditions of each  $a$  are added to  $G_p$ , and each  $g$  is removed from  $G_p$ . Also, the links  $(a, g)$  are added to  $L$ .

Figure 3 shows the metastates that TGP expands for solving the problem in example 1. The initial metastate  $m_0$  contains the plan graph until level 3,  $PG_3$ , and the problem goals, included in  $G_p$ . The first search operator  $o0$  assigns two instantiations of the debark action to achieve the goals in  $G_p$ .  $F$  maps  $(m_0; o0)$  into  $m_1$  by: (1) replacing in  $G_p$  the assigned goals for the preconditions of the two debark instantiations (debark has two preconditions,  $(at ?a ?c)$   $(in ?p ?a)$ ), (2) updating  $L$  with the two assignments, and (3) decreasing the graph level  $l = 2$ . Metastate  $m_2$  is generated in a similar way but with the action zoom whose preconditions are  $(at ?a ?c)$   $(fuel-level ?a ?l1)$   $(next ?l2 ?l1)$   $(next ?l3 ?l2)$ . All of them belong to the initial state (for both instantiations), so no new pending goal is added to  $G_p$ . Finally,  $m_3$  is a terminal state ( $G_p = \emptyset$  and  $l = 0$ ),

and the solution plan (plan 2 displayed in figure 1) is obtained from  $L$ .

TGP generates optimal parallel plans. In optimal parallel planning the task is to compute a plan that involves the minimum number of time steps where at each time step many actions can be executed in parallel (since they are independent). Consider, for instance, a problem where two persons P1 and P2 have to be transported from a location A to another one B with a plane C. A valid sequential plan would be (board P1 C A), (board P2 C A), (fly C A B), (debark P1 C B), (debark P2 C B). However, since there is no causal relation between the first two actions, nor between the two last ones, a parallel plan could be represented as parallel((board P1 C A), (board P2 C A)), (fly C A B), parallel((debark P1 C B), (debark P2 C B)). When ignoring action durations, the length of the parallel plan measures the total plan-execution time, and it is called make-span. In the previous example, the sequential plan would have a make-span of five, while the parallel plan would have a make-span of three. Actions that can be executed in parallel increase the length of the parallel plan in only one unit.

### IPSS Planner

IPSS is an integrated tool for planning and scheduling (Rodríguez-Moreno et al. 2006). The planning component is a nonlinear planning system that follows a means-ends analysis (see Veloso et al. [1995] for details). A backward-chaining procedure selects actions relevant to the goal (reducing the difference between the state and the goal) and arranges them into a plan. Then, a forward chainer simulates the execution of these actions and gradually constructs a total-order sequence of actions. The planner keeps track of the simulated world state that would result from executing this sequence. It utilizes the simulated state in selecting actions.

In terms of the unified model, IPSS can be described as follows.

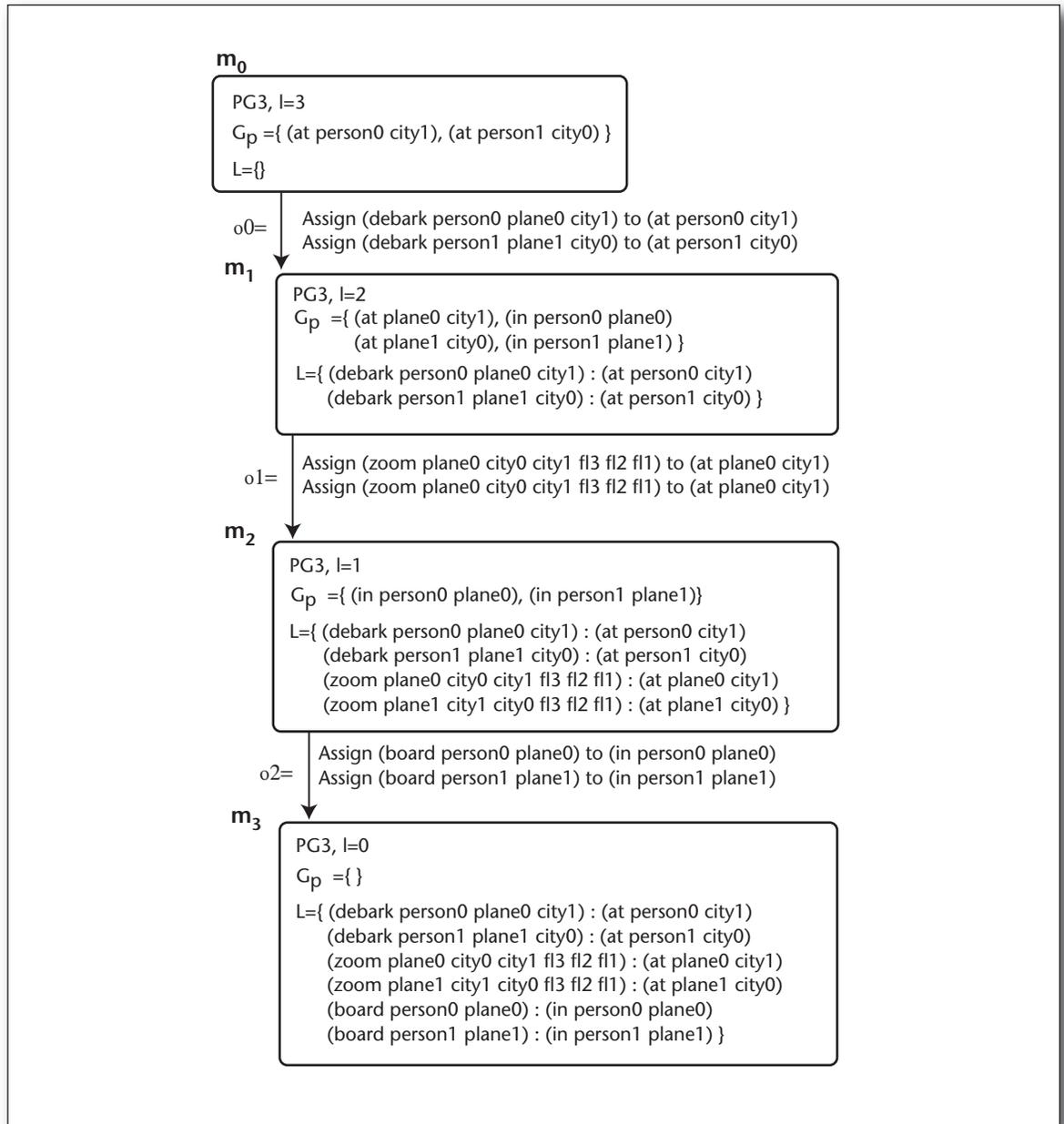


Figure 3. TGP Metastates of the Problem in Example 1.

Each metastate  $m$  is a tuple  $\{s, G_p, L, g, \sigma, a, P\}$  where  $s$  is the current state,  $G_p$  is the set of pending goals,  $L$  is a set of links  $(a, g)$  representing that the action  $a$  was used to achieve the goal  $g$ ,  $g$  is the goal on which the planner is working,  $\sigma$  is the name of an action that the planner has selected for achieving  $g$ ,  $a$  is an action that the planner has selected for achieving  $g$ ,<sup>1</sup> and  $P$  is the current partial plan for solving the problem.

The initial metastate,  $m_0 = \{s_0, G, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\}$ .  $s_0$  is the initial state of the problem and  $G$  represents the goals the planner has to achieve.

A terminal metastate will be of the form  $M_T = \{s_\nu, \emptyset, L_\nu, \emptyset, \emptyset, \emptyset, P\}$  such that  $G \subseteq s_\nu L_t$  will be the

links  $(a, g)$  between actions and goals, that is,  $(a, g) \in L_t$  if the planner chooses  $a$  to achieve  $g$ , and  $P$  is the solution plan.

The set of search operators  $O$  is composed of (given the current metastate  $\{s, G_p, L, g, \sigma, a, P\}$ ): “select a goal  $g \in G_p$ ”, “select an action name  $\sigma$  for achieving the current goal  $g$ ”, “select a grounding for the  $\sigma$  parameters” (the grounded action is kept in  $a$ ), and “apply the current action  $a$  to the current state  $s$ .”

Figure 4 shows the beginning of the search tree that IPSS expands for solving the problem in example 1. IPSS planning-reasoning cycle involves several decision points: (1) select a goal from the set of

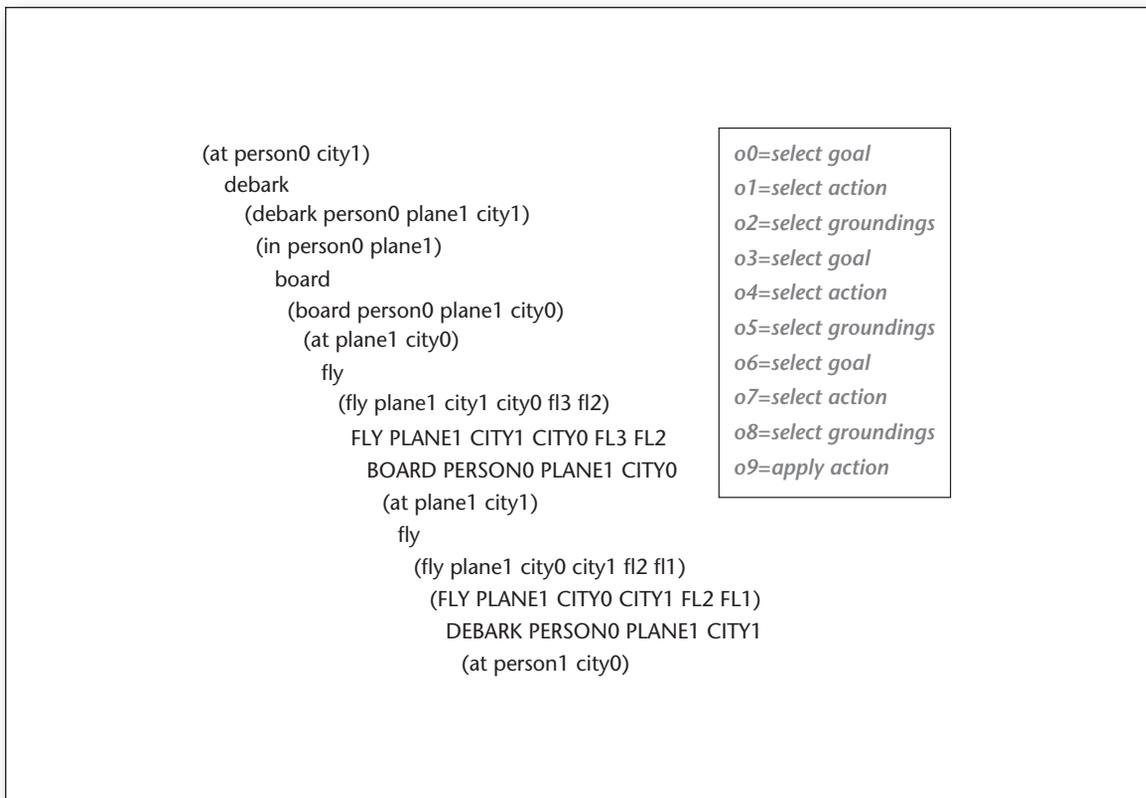


Figure 4: Partial Search Tree That IPSS Expands to Solve the Problem in Example 1.

pending goals; (2) select an action to achieve a particular goal; (3) select groundings to instantiate the chosen action; (4) apply an instantiated action whose preconditions are satisfied or continue sub-goaling on another unsolved goal. Uppercase text represents actions applied and added to the solution plan. We show the search operator applied at the right side of each node that corresponds to the decision the planner took.

#### Example 3.

IPSS finds Plan 1 displayed in figure 1 to solve the problem in example 1. The first metastates and search operators that IPSS applies to find this plan are:

1.  $m_0 = \{s_0, (\text{at person0 city1})(\text{at person1 city0}), \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\}$   
 $o_0 = \text{"select goal (at person0 city1)"}$
2.  $m_1 = \{s_0, (\text{at person1 city0}), \emptyset, (\text{at person0 city1}), \emptyset, \emptyset, \emptyset\}$   
 $o_1 = \text{"select action debark"}$
3.  $m_2 = \{s_0, (\text{at person1 city0}), \emptyset, (\text{at person0 city1}), \text{debark}, \emptyset, \emptyset, \emptyset\}$   
 $o_2 = \text{"select groundings (debark person0 plane1 city1)"}$

When IPSS selects an action, it adds the action preconditions to  $G_p$ . For example, (in person0 plane1) is a precondition of the (debark person0 plane1 city1), so  $o_3$  can select it (see figure 4).

When IPSS applies an action, it changes the state  $s$  by adding and deleting the action effects.

## Transfer Learning Task

The goal of this work with respect to transfer is to exploit the differences between the two planning systems to create one that outperforms either TGP or IPSS individually. In this case, TGP generates optimal parallel plans (minimum make-span), while IPSS generates sequential plans. In problems with many equal resources, IPSS search bias (*or* search algorithm) does not take into account the make-span of solutions, so it does not try to minimize it. When it selects an object to ground an action, it does not reason on make-span, but on trying to select an object that can achieve the goal. Plan 1 displayed in figure 1 provides an example. IPSS finds Plan 1 in contrast to the optimal parallel plan, Plan 2, found by TGP. On the other hand, the Graphplan algorithm has to generate all possible instantiations of all actions before searching for a solution, which often causes a combinatorial explosion in large-scale problems. By combining the biases of these two approaches, we aim to produce a planning system that performs well on both types of problems.

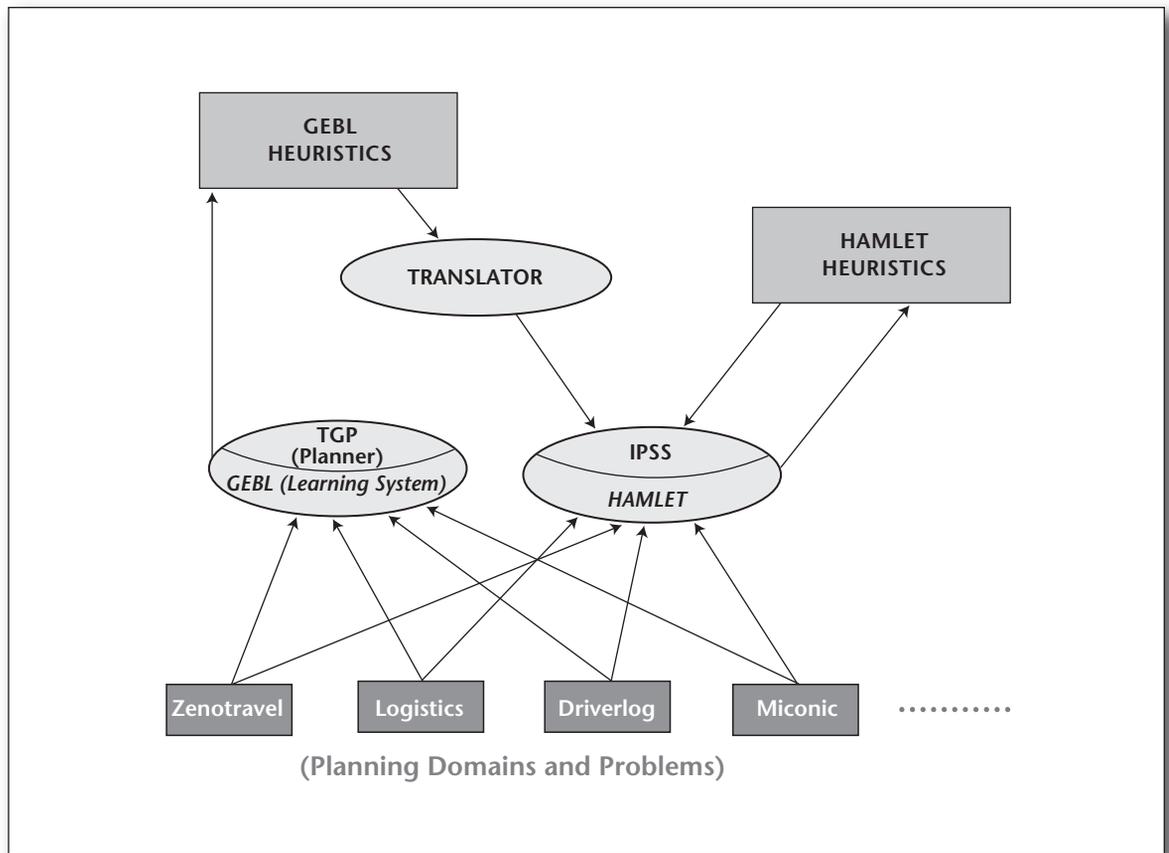


Figure 5. Transfer of Knowledge from TGP to IPSS.

Toward this end, we implemented a deductive learning method to acquire control knowledge by generating bounded explanations of the problem-solving episodes on TGP, named GEBL (Graphplan explanation-based learning, or GraphplanEBL). The learning approach builds on HAMLET (Borrajo and Veloso 1997), an inductive-deductive system that learns control knowledge in the form of PRODIGY control rules. We have also implemented a translator from the rules learned in TGP to the representation language for heuristics in IPSS. Figure 5 shows the transfer of knowledge between planner TGP and IPSS. GEBL learns heuristics from the experience of TGP solving problems from different planning domains. IPSS uses these heuristics, after a translating process, to improve the efficiency on solving new planning problems. The figure also shows the heuristics that HAMLET learns from IPSS (used to compare our approach).

From a perspective of heuristic acquisition (without considering the transfer task yet), the key issue consists of learning how to select search operators of the metaproblem space given each metastate. The learned heuristics may be viewed as functions,  $H : M \rightarrow O$ , that map a metastate  $m$  into a search operator:  $H(m) \rightarrow o$ . This is due to the fact that the decisions made by the planner (branch-

ing) that we would like to guide are precisely what search operator to apply at each metastate. We define these  $H(m)$  by a set of control rules, if conditions( $m$ ) then select  $o$ , in which  $m$  is a metastate and  $o$  is a search operator of the metaproblem space (metaoperator). If some conditions on the current metastate hold, then the planner should apply the corresponding search operator. In the transfer task, we learn the control rules in one planner, but they have to guide the search of a different planner.

### Problem Setting

The transfer setting consists of two planning techniques,  $PI^1$  and  $PI^2$ , a set of domain problem spaces,  $\Phi$ , and two sets of metaproblem spaces,  $\Xi^1$  and  $\Xi^2$ . The transfer task is then to learn heuristics in  $PI^1$  (a collection of  $H^1(m^1)$  functions) from  $PI^1$  search episodes, and then map them into heuristics for  $PI^2$  (a collection of  $H^2(m^2)$  functions) that can guide future  $PI^2$  search episodes. Formally:

$$\begin{aligned}
 P &= \langle S, s_0, G, A, T \rangle \in \Phi \\
 Mp^1 &= \langle M^1, m^1_0, M^1_T, O^1, F^1 \rangle \in \Xi^1 \\
 Mp^2 &= \langle M^2, m^2_0, M^2_T, O^2, F^2 \rangle \in \Xi^2 \\
 H^1(m^1) &: \{h_i \mid h_i = (\text{if conditions}(m^1) \text{ then select } O^1)\} \\
 H^2(m^2) &: \{h_i \mid h_i = (\text{if conditions}(m^2) \text{ then select } O^2)\}
 \end{aligned}$$

The source tasks consist of several domain problem

spaces in  $\Phi$  in which  $s_0$  and  $G$  differ among them and the other elements are equal. Thus, the source includes a collection of training problems, each defined by a given initial state and goals, for a single domain. The target tasks consist of a second collection of problem spaces such that  $s_0^t$  and  $G^t$  differ among them and from  $s_0$  and  $G$ .  $PI^1$  searches the solution plans in  $M_p^1 \in \Xi^1$  and  $PI^2$  in  $M_p^2 \in \Xi^2$ . Again, all  $M_p^1 \in \Xi^1$  have equal  $M^1$ ,  $O^1$ , and  $F^1$  sets; and all  $M_p^2 \in \Xi^2$  have equal  $M^2$ ,  $O^2$  and  $F^2$  sets (for each planning problem, the planners search for the solution in a particular metaproblem space, but each planner always uses the same strategy to solve the problems). Therefore, the transfer method defines explicit mappings between: (1) source and target tasks, (2) metastates,  $M^1 \rightarrow M^2$ , and (3) metaoperators,  $O^1 \rightarrow O^2$ .

For example, the metastate  $m_0$  and metaoperator  $o0$  displayed in figure 3 map into the metastate  $m_1$  and the metaoperators  $o1$  and  $o2$  displayed in example 3. TGP chooses action (debark person0 plane0 city1), but IPSS chooses other grounding for its parameters, (debark person0 plane1 city1). So, the TGP metaoperator  $o0$  should be mapped into two IPSS metaoperators: IPSS  $o1$  and a new  $o2$  with the TGP grounding for the parameters (*plane0* instead of *plane1*).

Source and target tasks were automatically generated using random problem generators. The knowledge acquisition method learns the heuristics from all the source problems and all the heuristics are used for solving all target tasks. There is no explicit task selection mechanism to select the source tasks that are more similar to the target ones to guard against negative transfer.<sup>2</sup> This differs from methods such as MASTER, which can measure task similarity through model prediction error, used in transfer learning for reinforcement learning (Taylor, Jong, and Stone 2008).

## Proposed Solution

A possible solution to the posed problem is to define languages for representing heuristics that consider metastate characteristics and metaoperators, and then to find similarities between  $M^1$  and  $M^2$  characteristics and  $O^1$  and  $O^2$  metaoperators. Metastate characteristics, called *metapredicates*, are queries to the current metastate of the search, such as whether some literal is true in the current state (Veloso et al. 1995). In this way, metastates are described by general characteristics and each planner defines its own specific implementation. This makes mappings for transfer easier to define. In general, there might be several ways of defining the mapping between both metastates, but as we will shortly explain, we have chosen to implement the mapping for which we have obtained best results.

Our approach to transfer control knowledge

between two planners  $PI^1$  and  $PI^2$  has five parts: (1) a language for representing heuristics for  $PI^1$  ( $H^1$  functions); (2) a knowledge-acquisition method to learn  $H^1$  functions from  $PI^1$  search episodes; (3) a mapping between the metaproblem space of  $PI^1$  and the metaproblem space of  $PI^2$ , which is equivalent to provide a translation mechanism between  $H^1$  and  $H^2$  functions; (4) a language for representing heuristics for  $PI^2$  ( $H^2$  functions); and (5) an interpreter (matcher) of  $H^2$  functions in  $PI^2$ . Given that IPSS already has a declarative language to define heuristics (part 4), and an interpreter of that language (part 5), we only needed to define the first three parts. The heuristic language for  $PI^1$  (step 1) should ideally be as similar as possible to the language used by  $PI^2$ , IPSS, in order to simplify the mapping problem. With this in mind, we borrowed as many metapredicates as possible from the IPSS declarative language. In the next section, we discuss the heuristic learning methods, and then focus on the mapping problem in the Knowledge Reuse section.

## Knowledge Acquisition

Our learning systems rely on explanation-based learning (EBL) to acquire the knowledge that gets transferred between the two planners. EBL is a machine-learning technique that takes advantage of a complete domain theory to generalize from training examples. In the context of planning, the domain description provides the needed theory while the search trace produced by the planner provides the training examples. In the present case, GEBL is used to generate control knowledge from TGP. As explained earlier, both planners search a metaproblem space. GEBL generates explanations about why particular local decisions made by the planner during the search process led to a success state. GEBL generalizes these explanations into TGP control rules or heuristics ( $H^1$ ). In particular, the GEBL process includes two stages (Borrajo and Veloso 1997):

*Stage One: Labeling the search tree.* (1) Generation of a trace: TGP solves a planning problem and obtains a trace of the search tree. (2) Identification of successful decisions: GEBL identifies the metastates located in the tree leaves and uses them to determine which decision nodes led to the solution.

*Stage Two: Generating control rules.* (1) Generation of specialized rules: The system creates new rules from two consecutive successful decision points by selecting the relevant preconditions. They are specialized in the sense that only constants appear in their condition and action parts. (2) Generalization of control rules: GEBL generalizes constants in the specialized control rules to variables, and applies goal regression to reduce the number of rule conditions. GEBL follows the preceding

process for every training planning problem. This creates a set of control rules from every problem in the training set. In the following subsections, we provide more detail for each step in the process.

### Labeling the Search Tree

First, TGP generates a search tree by solving a planning problem. The search tree contains the decisions made by the planner in the metaproblem space. Each decision point (or node) is a metastate with a set of successors, and every successor is an applicable search operator that leads to a successor metastate. The search tree includes three types of nodes.

*Success nodes* belong to a solution path. *Failure nodes* arise in two ways: when consistent actions that achieve all goals in the node cannot be found, or when all of the node's children fail. *Memo-failure nodes* were never expanded. This happens if the goals were previously memorized as nogoods (failure condition of Graphplan-based planning).

All nodes in the search tree are initially labeled as unexpanded (memo-failure). If a node fails during the planning process, its label changes to *failure*. When the planner finds a solution, all the nodes that belong to the solution path are labeled as *success*.

For example, all the metastates displayed in figure 3 can be labeled as *success*. As the solution was reached without backtracking, there are no *failure* nodes in this search trace. And, when TGP reaches the terminal metastate *m3* all nodes between the root node (*m0*) and the solution node (*m3*) can be labeled as *success*.

### Generating Control Rules

After labeling the search tree nodes, GEBL generates specialized control rules from pairs of consecutive success nodes separated by one or more failure nodes. The rationale is that if there is no possibility of failure between two successful metastates, then default decisions are sufficient. This mode of learning is typically called lazy learning, and from this point of view, memo-failure nodes belong to the same category as success nodes since the planner does not explore them. Note that we call the resulting rules specialized because they contain only constant objects, with no variables.

Learning control rules from correct default decisions can be useful in some cases. For instance, after the rules have been generalized, they could apply to future situations different from the ones they have been created from. This second mode of learning is named eager learning (Borrajó and Veloso 1997). GEBL can use both learning modes.

Lazy learning is usually more appropriate when the control knowledge is obtained and applied to the same planner to correct only the wrong default decisions. However, eager learning may be useful

for the purpose of transferring knowledge between different planners, as is the case of this article, because the metaproblem space will be searched differently and default decisions will also differ. Figure 6 shows the rules generated from a fictitious search tree. The search starts in level 7 (*l7*) where problem goals (*G*) appear and *L* is still empty. Each metastate (also named decision point) has a different set of pending goals (such a *G1*, *G2*, and *G8*), set of assignments (such a *L1*, *L2*, and *L8*) and graph-plan level (such *l7*, *l6*, and *l0*). When a node fails, the search process backtracks and the affected nodes are labeled as *failure* (see decision points (*G2*, *L2*), (*G9*, *L9*) and *DPx1*). When the search reaches the terminal node *DPy4* all nodes between the root node and *DPy4* are labeled as *success*. Afterward, the rule-generation process starts generating the rules displayed in figure 6.

Learned control rules have the same syntax as in PRODIGY, which can select, reject, or prefer alternatives. The conditions of control rules refer to queries, called *metapredicates*, to the current metastate of the search. PRODIGY provides several metapredicates for determining whether some literal *l* is true in the current state,  $l \in s$ , whether some literal *l* is the current goal,  $l = g$ , whether some literal *l* is a pending goal,  $l \in Gp$ , or some object is of a given type. However, the language for representing heuristics also admits coding user-specific functions.

GEBL learns two kinds of control rules: goal selection and operator selection. The inputs for rule learning are the two consecutive success metastates with their goal and assignment sets. The select goals rules learn the goal that persists in the decision point (when just one goal persists) and the select operator rules learn to select the actions that fulfill the goals in the decision point. One rule is created for each achieved goal.

As an example, GEBL would not create any rules from the tree in figure 3 when in lazy learning mode, because there are no failure nodes and no backtracking. On the other hand, in eager mode, it would create two rules from every pair of decision nodes. For instance, from the first two decision points *m0* and *m1*, one of the rules would choose the action (debark person0 plane1 city1) to achieve the goal (at person0 city1) (the second rule is equivalent with respect to the second goal).

GEBL generalizes the initial control rules by converting constants to variables (denoted by angle brackets). In order to generalize them further, the number of true-in-state metapredicates is reduced by means of a goal regression (as in most EBL techniques [DeJong and Mooney 1986]). Only those literals in the state strictly required, directly or indirectly, by the preconditions of the action involved in the rule (the action that achieves goal) are included.

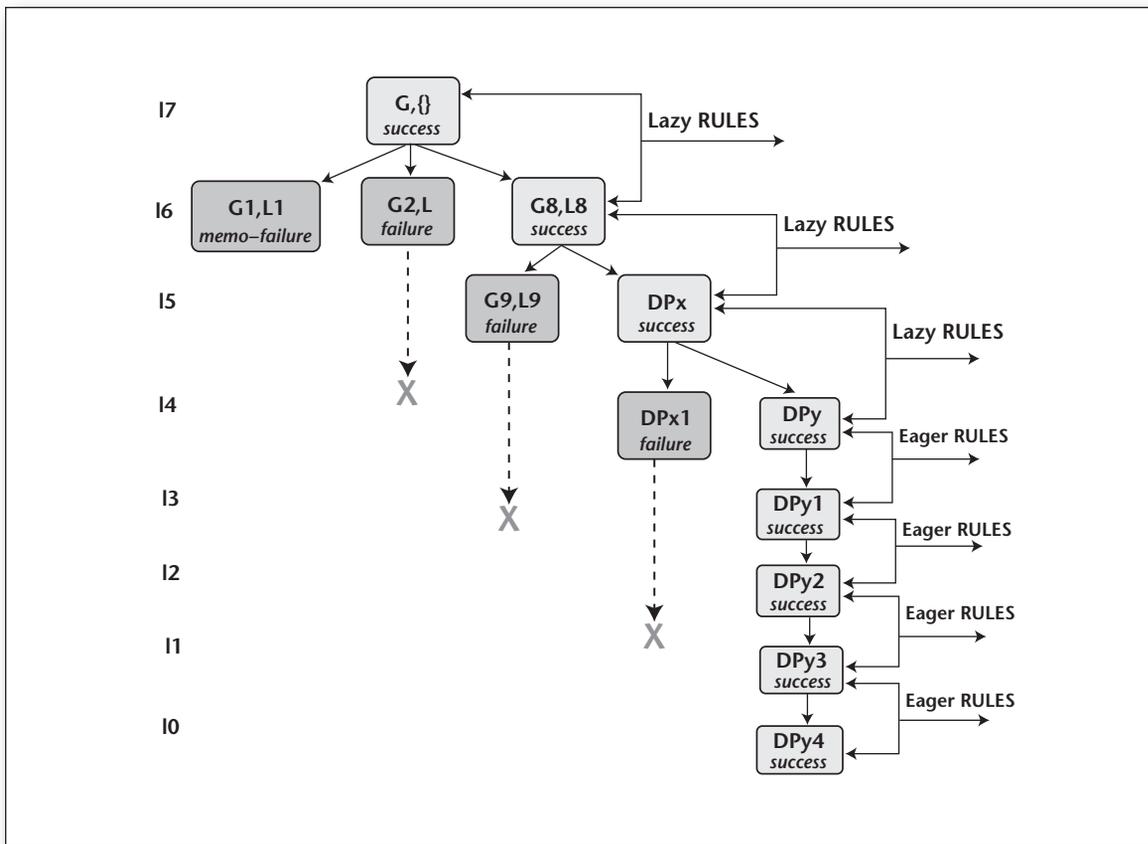


Figure 6. GEBL Rule-Generation Process.

One of the select operator rules generated from the  $m_0$  and  $m_1$  decision points of figure 3 is displayed in figure 7. This rule selects the action debark for achieving the goal of having  $\langle \text{person0} \rangle$  in  $\langle \text{city1} \rangle$ . The rule selects the most convenient plane ( $\langle \text{plane1} \rangle$ ) for eventually debarking: the plane with enough fuel to fly that is in the same city as the person that will debark.

Figures 8 displays an example of a select goal rule. It chooses between two goals that involve moving either  $\langle \text{person0} \rangle$  to  $\langle \text{city1} \rangle$  or  $\langle \text{person1} \rangle$  to  $\langle \text{city0} \rangle$  (the arguments of the metapredicates target-goal and some-candidate-goals). The rule selects moving  $\langle \text{person1} \rangle$  because she or he is in a city where there is a plane  $\langle \text{plane1} \rangle$  with enough fuel to fly.

### Knowledge Reuse

The final step in transferring control rules generated by TGP into IPSS is to translate the GEBL rules into the IPSS control language. The translation must include both the syntax (small changes given that we have built the control-knowledge language for TGP based on the one defined in IPSS) and the semantics (translation of types of condi-

tions in the left-hand side of rules, and types of decisions — search operators of the metaproblem space). So, the mapping must support the translation of the left-hand side of rules (conditions referring to metastates) and the right-hand side of rules (selection of a search operator of the metaproblem space).

With respect to the right-hand side of the control rules, the translator takes the following steps.

*Step One.* Decide which goal to work on first. IPSS has a specific search operator for choosing a goal from the pending goals; on the contrary, the TGP search operator always assigns an action (including the *no-op*) for each goal in the set of pending goals. However, when TGP selects the *no-op* to achieve a goal (in the backwards process), this means the planner decides that it is better to achieve the goal in levels closer to the initial state. Therefore, in case only one goal persists, it can be transferred to the way in which IPSS would select that goal.

*Step Two.* Each TGP action for achieving a particular goal must be split into two IPSS metaspace operators: one for selecting the action name (debark) and another for instantiating it ((debark p1 p11 c0)). This means that for each select-operator control rule learned by GEBL, two control rules have to be defined for IPSS. From the IPSS side,

```
(control-rule rule-ZENO-TRAVEL-ZENO1-e1
  (if (and (current-goal (at <person0> <city1>))
    (true-in-state (at <person0> <city0>))
    (true-in-state (at <plane1> <city0>))
    (true-in-state (fuel-level <plane1> <fl1>))
    (true-in-state (aircraft <plane1>))
    (true-in-state (city <city0>))
    (true-in-state (city <city1>))
    (true-in-state (flevel <fl0>))
    (true-in-state (flevel <fl1>))
    (true-in-state (next <fl0> <fl1>))
    (true-in-state (person <person0>))))
  (then select operators (debarb <person0> <plane1> <city1>)))
```

Figure 7. Example of Select Operator Rule in the Zenotravel Domain.

```
(control-rule regla-ZENO-TRAVEL-PZENO-s1
  (if (and (target-goal (at <person1> <city0>))
    (true-in-state (at <person1> <city1>))
    (true-in-state (at <person0> <city0>))
    (true-in-state (at <plane1> <city1>))
    (true-in-state (fuel-level <plane1> <fl2>))
    (true-in-state (aircraft <plane1>))
    (true-in-state (city <city0>))
    (true-in-state (city <city1>))
    (true-in-state (flevel <fl1>))
    (true-in-state (flevel <fl2>))
    (true-in-state (next <fl1> <fl2>))
    (true-in-state (person <person0>))
    (true-in-state (person <person1>))
    (some-candidate-goals ((at <person0> <city1>))))))
  (then select goals (at <person1> <city0>)))
```

Figure 8. Example of Select Goals Rule in the Zenotravel Domain.

three types of control rules can be learned in TGP to guide the IPSS search process: select goals, select operator (select an action), and select bindings (select the grounding of the action parameters) rules. Select goals rules are the rules generated by TGP when one goal persists (as explained in point 1), and the two other types of rules come from the TGP select-operator rules (as explained in point 2).

The equivalence between metastates is not straightforward (for translating the conditions of control rules) due to the differences between the search algorithms. IPSS performs a kind of bidirec-

tional depth-first search, subgoaling from the goals, and executing actions from the initial state. TGP performs a backward search on the planning graph that remains unchanged during each search episode. The difficulty arises in defining the state  $s$  that will create the true-in-state conditions of the control rules. While IPSS changes the state while planning, given that it applies actions while searching for the solution, TGP does not apply any action. It just propagates all potential literals of the state that could be true after  $N$  levels of action applications. Thus, TGP does not have an explicit state at each level, but a set of potential states.

When TGP learns rules, we must consider two possibilities. The simplest one is that the state  $s$  to be used in control rules is just the problem initial state  $s_0$ . The second possibility is that TGP persists a goal that was already achieved by IPSS (as explained earlier). The state  $s$  is therefore the one reached after executing the actions needed to achieve the persisted goals in the TGP metastate. To compute this state, we look in the solution plan, and progress  $s_0$  according to each action's effects in such a partial plan.

Equivalent metastates are computed during rule generation, and a translator makes several transformations after the learning process finishes. The first splits the select-operator control rules into two: one to select the action and another one to select its instantiation. The second translates true-in-state metapredicates referring to variable types into type-of-object metapredicates.<sup>3</sup> Finally, the translator deletes those rules that are more specific than a more general rule in the set (they are subsumed by another rule) given that GEBL does not perform an inductive step.

## Experimental Results

We conducted several experiments to examine the utility of our approach, under the hypothesis that we can improve IPSS performance by transferring knowledge generated by GEBL. We compare our work with HAMLET (Borrajo and Veloso 1997), a system that learns control rules from IPSS problem-solving episodes (see figure 5).

Our experiments employed a subset of benchmark domains taken from the repository of planning competitions<sup>4</sup> (the STRIPS versions, since TGP can only handle the plain STRIPS version). Within each of these domains, we trained HAMLET and GEBL against one set of randomly generated source problems and tested them against a different random set of target tasks. HAMLET learns a set of domain-dependent heuristics from these source problems by performing an EBL step on IPSS traces followed by induction to acquire control rules. GEBL learns a different set of domain-dependent heuristics by solving source problems with

TGP. In the target task, we use IPSS to solve the test problems three times: without heuristics, with heuristics acquired by HAMLET, and with heuristics acquired by GEBL.

Note that we trained GEBL and HAMLET on slightly different sets of source problems in order to treat both algorithms fairly. The reason is that GEBL suffers from the utility problem (of generating many overspecific control rules) common to EBL systems, which can be controlled by limiting the number of training problems and goals. In contrast, HAMLET incorporates an inductive module that diminishes the utility problem, but also requires more training data as it incrementally learns from experience generated by acquired control rules. Our solution was to train GEBL on a subset of the source problems we employed to train HAMLET.

Table 1 illustrates the number and complexity of the source and target problems we presented to both systems. Among the source tasks, column R identifies the number of generated rules, column TR displays the number of training problems, and column G measures problem complexity by the range in the number of supplied goals. The table also displays the complexity of the target problems.

Table 2 shows the percentage of target problems IPSS solved alone, and by using control heuristics acquired through HAMLET and GEBL. The table also shows the percentage of problems solved by the TGP planner. In general, we tested against 100 random target problems in each domain. The exceptions were Miconic, and Gripper, where we employed the 140 and 20 target problems used respectively in the domains for the planning competition. We applied a 60 second time limit across all target problems, and varied the number of goals per problem as shown in table 1. The difference between source and target task complexities is relevant: source tasks have at most 5 goals, while target tasks have up to 58, as in the Robocare domain (see table 1).

The results show that IPSS solved a greater percentage of target problems using control rules acquired by GEBL vs. HAMLET. This result holds in all but one domain (Miconic). Transferred knowledge also improved IPSS performance in all but one domain (Robocare), even to the point that performance with transfer surpassed TGP levels in three domains; Zenotravel, Miconic, and Gripper.

However, control rules acquired through HAMLET reduced IPSS performance in Driverlog and Robocare. We attribute this behaviour to the intrinsic problems of the HAMLET learning technique: first, the utility problem common to EBL methods, and second, the fact that incremental relational induction techniques can fail to converge to a set of control rules in the available time.

Domain	Source Task						Target Task
	HAMLET			GEBL			
	R	TR	G	R	TR	G	
Logistics	16	400	1-3	406	200	2	1-5
Driverlog	9	150	2-4	71	23	2	2-6
Zenotravel	8	200	1-2	14	200	1-2	2-13
Miconic	5	10	1-2	13	3	1	3-30
Rovers	-	90	1-3	59	7	2	2-13
Robocare	5	21	1-4	36	4	2-3	5-58
Gripper	3	40	2-5	15	10	2-5	4-37

Table 1. Number of Generated Rules, Number of Source Problems, and Complexity (Range of Number of Goals) of Source and Target Problems.

Domain	TGP	IPSS	
		HAMLET Heuristics	GEBL Heuristics
Logistics	97%	12%	25%
Driverlog	100%	26%	4%
Zenotravel	21%	38%	40%
Miconic	20%	4%	100%
Rovers	72%	37%	-
Robocare	95%	99%	44%
Gripper	10%	100%	100%

Table 2. Percentage of Solved Random Problems With and Without Heuristics.

Here, HAMLET failed to find any control rules in the Rover domain.

The transferred knowledge also improves the efficiency of the problem solver's search and the quality of its solutions. We measure efficiency by the number of nodes the planner generates en route to a solution (this relates to both memory consumption and search time). We measure plan quality by the maximum length of the parallel plan structure in the target problem's solution, called its make-span. Figure 9 shows the percentage decrease in the number of nodes and the make-span due to transferred knowledge across all domains. In particular, we display the percentage decrease in the number of nodes (or make-span) measured for the problems solved by IPSS with GEBL rules, relative to those quantities measured for IPSS alone. The data for the Logistics domain uses the make-span from problems solved by HAMLET instead, as IPSS found plans with one or two actions that skewed the resulting percentage. The data shows that the transferred knowledge

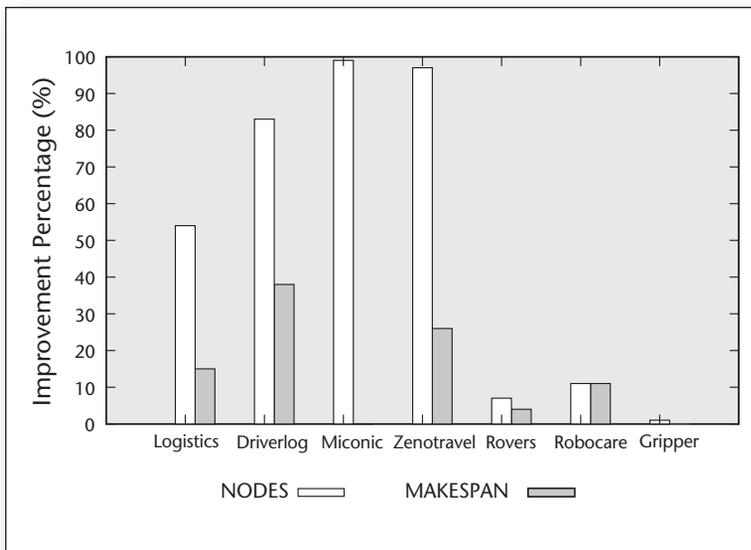


Figure 9. Percentage of Improvement in the Number of Generated Nodes and Make-Span Due to the Transferred Knowledge.

always decreased the number of nodes. In addition, it decreased the make-span in every domain except Zenotravel and Gripper, where it remained equal. These results demonstrate that we were able to transfer the search bias of TGP, which obtains optimal parallel plans, to IPSS, which does not explicitly use that bias in its search.

## Related Work

As far as we know, our approach is the first one that is able to transfer learned knowledge between two planning techniques. However, some recent work in planning success fully utilizes certain types of knowledge acquired from prior experience to improve performance within the same planner. For instance, Marvin (Coles and Smith 2007) and Macro-FF (Botea et al. 2005) learn macrooperators, while the systems reported in de la Rosa et al. (de la Rosa, Jiménez, and Borrajo 2008) and Yoon et al. (Yoon, Fern, and Givan 2008) learn control knowledge for forward search planning. Other work has seeded a planner with control knowledge acquired by a separate learner. This is the case, for instance, of EVOCK (Aler, Borrajo, and Isasi 2002) that used Genetic Programming to evolve the rules learned by HAMLET, overcoming the search bias of HAMLET deductive and inductive components. Also, Fernández, Aler, and Borrajo (2004) presented ways of obtaining prior control knowledge from a human, another planner, and a different machine-learning technique. This work applied previous experience to improve the performance of the control rule learner, as opposed to the performance of the planner, as in the current article. However,

none of these systems described transferred structured knowledge between problem solvers.

Our approach is related to work on transfer in reinforcement learning (RL) settings. Taylor and Stone (2009) describe methods for communicating control knowledge between source and target tasks in several forms, for example, as preference rules, state-action mappings, or value functions, although the work is typically conducted in contexts where the source and target problem solvers both employ RL. These methods could be extended to communicate preference rules obtained from other learners, through mechanisms of the kind we have explored in this article. Besides, RL and planning are also related under the viewpoint of problem solving as a means of selecting the next action to apply given the current state. In RL problems, agents learn policies that map states to actions with the goal of maximizing the reward over time (Sutton and Barto 1998). The main difference between planning and RL is that in RL goals are usually implicit (given by the reward function), while in planning those goals are explicitly given in the problem formulation. Learning control knowledge in planning aims to define these states to actions mappings.

Learning control knowledge in planning can be divided in two groups: those that learn a complete policy, and those that learn a partial policy. In the first case, once a policy has been learned we could potentially not use the underlying planner to solve problems but to directly use the policy (García-Durán, Fernández, and Borrajo 2008; Yoon, Fern, and Givan 2007; Khardon 1999; Martín and Geffner 2000). In the second case, the learning systems attempt to obtain heuristics that guide the planner search (Yoon, Fern, and Givan 2008; de la Rosa, Jiménez, and Borrajo 2008). They are incomplete policies in the sense that they cannot replace completely the underlying planner. As in our work, they are usually employed to complement (guide) the search instead of replacing the search. These systems could be modeled in terms of our unified model by defining their metaproblem spaces. We could then study the viability of transferring knowledge among them through the learned heuristics.

It is instructive to frame our work using the vocabulary employed by Taylor and Stone (2009) to categorize transfer learning methods. Four of their dimensions are relevant here: (1) the assumed differences between source and target tasks, (2) the method of selecting source tasks, (3) the form of the mapping between source and target tasks/solutions, and (4) the type of knowledge transferred. Regarding dimension 1, our source and target tasks can differ in the initial and goal state; in fact, the number and semantics of goals can also vary. Work by Fernández and Veloso (2006) is similar in that

tasks can differ in both the initial and goal state, while the goals are unique (that is, it considers one goal per problem, as in most RL tasks). Regarding dimension 2, our transfer learning algorithm acquires heuristics from multiple source tasks and employs them all for transfer without more explicit task selection assumptions. There are several works in the RL literature that fall into this category. Regarding dimension 3, our transfer learning algorithm employs an intertask mapping between the metaproblem space of the source tasks and the metaproblem space of the target tasks, where the metastates and meta-actions both vary. A human supplies the mapping like in the RL work reported by Taylor, Jong, and Stone (2008). Regarding dimension 4, our object of transfer is the situation to (meta) action mapping, analogous to the RL work reported in Sherstov and Stone (2005).

## Conclusions

This article presents an approach to transferring control knowledge (heuristics) learned from one planner to another planner that employs a different planning technique and bias. In the process, we address several of the most difficult problems in transfer learning: the source and target tasks are different, target tasks are more complex than source tasks, the learning algorithm acquires knowledge from multiple tasks, we do not explicitly preselect source tasks that are similar to the target tasks within a given domain, and the state and action spaces differ between the knowledge acquisition and knowledge reuse tasks.

Our work began from the observation that the search biases of a planner could prevent it from solving problems that a different planner could. We hypothesized that transferred knowledge could compensate for this bias, and developed a method for communicating heuristics in the form of control rules between planners. We tested our approach by applying it to acquire control rules from experience in the TGP planner, and using it to improve performance in the IPSS planner.

In more detail, we compared the behavior of IPSS in a variety of benchmark planning domains using heuristics learned in TGP and heuristics learned by HAMLET, an inductive-deductive learning system based on PRODIGY that shares the search biases of IPSS. In particular, TGP generates optimal parallel plans, while IPSS generates sequential plans. In tasks with many equal resources, IPSS has a bias towards plans in which the same resource is heavily used. These solutions cannot be parallelized, and have high scores relative to the make-span metric. So, it is reasonable to expect that the transferred control knowledge, learned in TGP, can help IPSS minimize the make-span. The experiments confirm our hypothesis; the

rules learned in TGP cause IPSS to reduce the make-span of the new solution plans in five of seven domains, where it remained equal in the other two. The transferred heuristics also let IPSS increase the percentage of problems it solved while reducing memory consumption (measured by the number of generated nodes). The learned rules worsened the percentage of problems solved in one domain, although there was no evidence of negative transfer with respect to memory use or plan quality (make-span).

At the same time, the nature of the performed experiments makes it hard to know if the improvement is due to superior learning by GEBL versus HAMLET, or because of the additional bias that we have introduced by transferring knowledge across planners, or even because the learning biases used to generate the heuristics. GEBL and HAMLET employ similar knowledge representation languages and rule-learning techniques (EBL), so it seems plausible that the improvement is only due to the extra information that the search bias of TGP provides to IPSS. We plan to clarify the source of power in future work.

More broadly, our work applies to the problem of producing general-purpose planners. While there is no universal optimal planner (a universally superior strategy for planning in all planning problems and domains), it may be possible to collect and employ a wide body of good domain-dependent heuristics by employing machine-learning methods to integrate multiple planning techniques. That is, we propose to learn control knowledge for each domain from the planning paradigms that behave well in that domain. This could lead to the creation of a domain-dependent control-knowledge repository that could be integrated with domain descriptions and used by any planner. Toward this end, we intend to show that our approach can be applied to other combinations of planners and learning systems. More specifically, we hope to explore learning from SAT problem-solving episodes (using SATplan [Kautz, Selman, and Hoffmann 2006]) while applying the acquired knowledge within a forward-chaining planner. In principle, this will demonstrate that knowledge transfer of the form explored in this article can improve performance of the most competitive current planning engines.

## Acknowledgements

This work has been partially supported by the Spanish MICINN under projects TIN2008-06701-C03-03 and TIN200508945-C06-05.

## Notes

1. IPSS selects an action for achieving a goal in two steps: first, it selects the name of the action and second it selects values for its parameters (grounding).
2. In our case, negative transfer occurs when the learned

heuristics mislead  $P^2$  search episodes.

3. TGP does not handle variable type expressions explicitly; it represents them as initial state literals. However, IPSS domains require variable type definitions as in typed PDDL.

4. See [www.icaps-conference.org](http://www.icaps-conference.org).

## References

- Aler, R.; Borrajo, D.; and Isasi, P. 2002. Using Genetic Programming to Learn and Improve Control Knowledge. *Artificial Intelligence* 141(1–2): 29–56.
- Blum, A., and Furst, M. 1995. Fast Planning through Planning Graph Analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*. San Francisco: Morgan-Kaufmann Publishers.
- Borrajo, D., and Veloso, M. 1997. Lazy Incremental Learning of Control Knowledge for Efficiently Obtaining Quality Plans. *Artificial Intelligence Review* 11(1–5): 371–405.
- Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *Journal of Artificial Intelligence Research* 24: 581–621.
- Bylander, T. 1994. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence* 69(1–2): 165–204.
- Coles, A., and Smith, A. 2007. Marvin: A Heuristic Search Planner with Online Macro-Action Learning. *Journal of Artificial Intelligence Research* 28: 119–156.
- de la Rosa, T.; Jiménez, S.; and Borrajo, D. 2008. Learning Relational Decision Trees for Guiding Heuristic Planning. In *Proceeding of the International Conference on Automated Planning and Scheduling (ICAPS 08)*. Menlo Park, CA: AAAI Press.
- DeJong, G., and Mooney, R. 1986. Explanation-Based Learning: An Alternative View. *Machine Learning* 1(1): 47–80.
- Fernández, F., and Veloso, M. 2006. Probabilistic Policy Reuse in a Reinforcement Learning Agent. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*. New York: Association for Computing Machinery.
- Fernández, S.; Aler, R.; and Borrajo, D. 2004. Using Previous Experience for Learning Planning Control Knowledge. In *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference*. Menlo Park, CA: AAAI Press.
- García-Durán, R.; Fernández, F.; and Borrajo, D. 2008. Learning and Transferring Relational Instance-Based Policies. In *Transfer Learning for Complex Tasks: Papers from the AAAI Workshop*. Technical Report WS-08-13. Menlo Park, CA: AAAI Press.
- Geffner, H. 2001. Planning as Branch and Bound and Its Relation to Constraint-Based Approaches. Technical Report, Universidad Simón Bolívar, Caracas, Venezuela.
- Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning — Theory and Practice*. San Francisco: Morgan Kaufmann Publishers.
- Kautz, H.; Selman, B.; and Hoffmann, J. 2006. SatPlan: Planning as Satisfiability. Unpublished booklet distributed at the 2006 International Planning Competition, the English Lake District, Cumbria, UK 6–10 June.
- Khardon, R. 1999. Learning Action Strategies for Planning Domains. *Artificial Intelligence* 113(1–2): 125–148.
- Martín, M., and Geffner, H. 2000. Learning Generalized Policies in Planning Using Concept Languages. In *Proceedings of the 7th International Conference on Knowledge Representation and Reasoning (KR 2000)*. San Francisco: Morgan Kaufmann Publishers.
- Nau, D. S. 2007. Current Trends in Automated Planning. *AI Magazine* 28(4): 43–58.
- Rodríguez-Moreno, M. D.; Oddi, A.; Borrajo, D.; and Cesta, A. 2006. IPSS: A Hybrid Approach to Planning and Scheduling Integration. *IEEE Transactions on Knowledge and Data Engineering* 18(12): 1681–1695.
- Sherstov, A. A., and Stone, P. 2005. Improving Action Selection in MDP's Via Knowledge Transfer. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*. Menlo Park, CA: AAAI Press.
- Smith, D., and Weld, D. 1999. Temporal Planning with Mutual Exclusion Reasoning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, 326–337. San Francisco: Morgan Kaufmann Publishers.
- Sutton, R. S., and Barto, A. G. 1998. *Introduction to Reinforcement Learning*. Cambridge, MA: The MIT Press.
- Taylor, M. E., and Stone, P. 2009. Transfer Learning for Reinforcement Learning Domains: A Survey. *Journal of Machine Learning Research* 10(1): 1633–1685.
- Taylor, M. E.; Jong, N. K.; and Stone, P. 2008. Transferring Instances for Model-Based Reinforcement Learning. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases — Part II*, 488–505. Berlin: Springer-Verlag.
- Torrey, L., and Shavlik, J. 2009. Transfer Learning. In *Handbook of Research on Machine Learning Applications*. Hershey, PA: IGI Global.
- Veloso, M.; Carbonell, J.; Pérez, A.; Borrajo, D.; Fink, E.; and Blythe, J. 1995. Integrating Planning and Learning: The PRODIGY Architecture. *Journal of Experimental and Theoretical Artificial Intelligence*. 7(1): 81–120.
- Yoon, S.; Fern, A.; and Givan, R. 2008. Learning Control Knowledge for Forward Search Planning. *Journal of Machine Learning Research* 9(4): 683–718.
- Yoon, S.; Fern, A.; and Givan, R. 2007. Using Learned Policies in Heuristic-Search Planning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*. Menlo Park, CA: AAAI Press.
- Zimmerman, T., and Kambhampati, S. 2003. Learning-Assisted Automated Planning: Looking Back, Taking Stock, Going Forward. *AI Magazine* 24(2): 73–96.

**Susana Fernández Arregui** is an associate professor in the Department of Computer Science at Universidad Carlos III. She received her PhD in learning search control for classical planning in 2006. She belongs to the Planning and Learning research group. Her research topics are learning search control for planning and the study of the role of emotions in AI. She worked for 10 years in companies such as Indra and Eliop S.A.

**Ricardo Aler** is an associate professor in the Department of Computer Science at Universidad Carlos III. He has researched in several areas, including automatic control-knowledge learning, genetic programming, and machine learning. He has also participated in several projects related to optimization and evolutionary computation. He has published over 80 journal and conference papers. He holds a PhD in computer science from Universidad Politécnica de Madrid (Spain) and a MSc in decision-support systems for industry from Sunderland University (UK). He graduated in computer science from the Universidad Politécnica de Madrid.

**Daniel Borrajo** has been a full professor of Computer Science at Universidad Carlos III de Madrid since 1998. He received his Ph.D. in computer science in 1990 and B.S. in computer science both at Universidad Politécnica de Madrid. He has published over 150 journal and conference papers mainly in the fields of problem-solving methods (heuristic search, automated planning, and game playing) and machine learning. He was the conference cochair of the International Conference of Automated Planning and Scheduling (ICAPS'06), chair of the Spanish conference on AI (CAEPIA'07), and a program committee member of conferences such as IJCAI (senior program committee member at IJCAI'11 and 07), AAAI, ICAPS, ICML, and ECML.