Transferring Learned Control-Knowledge between Planners

Planning and learning. Control rule learning

Abstract

As any other problem solving task that employs search, AI Planning needs heuristics to efficiently guide the problem-space exploration. Machine learning (ML) provides several techniques for automatically acquiring those heuristics. Usually, a planner solves a problem, and a ML technique generates knowledge from the search episode in terms of complete plans (macro-operators or cases), or heuristics (also named control knowledge in planning). In this paper, we present a novel way of generating planning heuristics: we learn heuristics in one planner and transfer them to another planner. This approach is based on the fact that different planners employ different search bias. We want to extract knowledge from the search performed by one planner and use the learned knowledge on another planner that uses a different search bias. The goal is to improve the efficiency of the second planner by capturing regularities of the domain that it would not capture by itself due to its bias. We employ a deductive learning method (EBL) that is able to automatically acquire control knowledge by generating bounded explanations of the problem-solving episodes in a Graphplan-based planner. Then, we transform the learned knowledge so that it can be used by a bidirectional planner.

1 Introduction

Planning can be described as a problem-solving task that takes as input a domain theory (a set of states and operators) and a problem (initial state and set of goals) and tries to obtain a plan (a set of operators and a partial order of execution among them) such that, when executed, this plan transforms the initial state into a state where all the goals are achieved. Planning has been shown to be PSPACE-complete [3]. Therefore, redefining the domain theory and/or defining heuristics for planning is necessary if we want to obtain solutions to real world problems efficiently. One way to define these heuristics is by means of machine learning. ML techniques applied to planning range from macro-operators acquisition, case-based reasoning, rewrite rules acquisition, generalized policies, deductive approaches of learning heuristics (EBL), learning domain models, to inductive approaches (based on ILP) (see [10] for a detailed account).

Despite the current advance in planning algorithms and techniques, there is no universally superior strategy for planning in all planning problems and domains. Instead of implementing an Universal Optimal Planner, we propose to obtain good domain-dependent heuristics by using ML and different planning techniques. That is, to learn control knowledge for each domain from the planning paradigms that behave well in this domain. In the future, this could lead to the creation of a domain-dependent control-knowledge repository that could be integrated with the domain descriptions and used by any planner. This implies the definition of a representation language or extension to the standard PDDL3.0, compatible both with the different learning systems that obtain heuristics, and also with the planners that use the learned heuristics.

In this paper we describe a first step in this direction by studying the possibility of using heuristics learned on one specific planner for improving the performance of another planner that has different problem-solving biases. Although each planner uses its own strategy to search for solutions, some of them share some common decision points, like, in the case of backward-chaining planners, what operator to choose for solving an specific goal or what goal to select next. Therefore, learned knowledge on some type of decision can potentially be transferred to make decisions of the same type on another planner. In particular, we have studied control knowledge transfer between two backward-chaining planners: from a Graphplan-based planner, TGP [7], to a state-space planner, IPSS, based on PRODIGY [9]. We have used PRODIGY given that it can handle heuristics represented in a declarative language. Heuristics are defined as a set of control rules that specify how to make decisions in the search tree. This language becomes our starting heuristic representation language. Our goal is to automatically generate these control rules by applying ML in TGP and then translate them into IPSS control-knowledge description language. We have implemented a deductive learning method to acquire control knowledge by generating bounded explanations of the problem-solving episodes on TGP. The learning approach builds on HAMLET [2], an inductive-deductive system that learns control knowledge in the form of control rules in PRODIGY, but only on its deductive component, that is based on EBL [6].

As far as we know, our approach is the first one that is able to transfer learned knowledge between two planning techniques. However, transfer learning has been successfully applied in other frameworks. For instance, in Reinforcement Learning, macro-actions or options obtained in a learning task can be used to improve future learning processes. The knowledge transferred can range from value functions [8] to complete policies [5].

The paper is organized as follows. Next section provides background on the planning techniques involved. Section 3 describes the implemented learning system and the learned control rules. Section 4 describes the translation of the TGP learned rules to be used in IPSS. Section 5 shows some experimental results. Finally, conclusions are drawn.

2 Planning models and techniques

In this section we first describe an unified planning model for learning. Then, we describe the planners IPSS and TGP in terms of that unified model. Finally, we describe our proposal to transfer control knowledge from TGP to IPSS.

2.1 Unified planning model for learning

In order to transfer heuristics from one planner to another, we needed to (re)define planning techniques in terms of a unified model, that accounts for the important aspects of planning from a perspective of learning and using control knowledge. On the base level, we have the domain problem-space P^d defined by (modified with respect to [1] and focusing only on deterministic planning without costs):

- a discrete and finite state space S^d,
- an initial state $s_0^d \in S^d$,
- a set of goals G^d, such that they define a set of non-empty terminal states as the ones in which all of them are true $S_T^d = \{s^d \in S^d \mid G^d \subseteq s^d\}$,
- a set of non-instantiated actions A^d, and a set of instantiated actions A_σ^d, and
- a function mapping non-terminal states s^d and instantiated actions a_σ^d into a state $F^d(a^d, s^d) \in S^d$.

We assume that both A_σ^d and $F^d(a^d, s^d)$ are non-empty. On top of this problem space, each planner searches in what we will call meta problem-space, P^m. Thus, for our purposes, we define the components of this problem space as:

- a discrete and finite set of states, S^m: we will call each state $s^m \in S^m$ a meta-state. Meta-states are planner dependent and include all the knowledge that the planner needs for making decisions during its search process. For forward-chaining planners each meta-state s^m will contain only a state s^d of the domain problem-space P^d. In the case of backward-chaining planners, as we will see later, they can include, for instance, the current state s^d, the goal the planner is working on $g^d \in G^d$, or a set of assignments of actions to goals
- an initial meta-state, $s_0^m \in S^m$
- a set of non-empty terminal states $S_T^m \subseteq S^m$
- a set of search operators A^m, such as "apply an instantiated action $a \in A^d$ to the current state s^d", or "select an instantiated action $a \in A^d$ for achieving a given goal g^d". These operators will be instantiated by bounding their variables (e.g. which action to apply), A_σ^m
- a function mapping non-terminal meta-states s^m and instantiated operators a_σ^m into a meta-state $F^m(a^m, s^m) \in S^m$

From a perspective of heuristic acquisition, the key issue consists of learning how to select instantiated operators of the meta problem-space given each meta-state. That is, the goal will be to learn functions that map a meta-state s^m into a set of instantiated search operators: $H(s_m) \subseteq A_\sigma^m$. This is due to the fact that the decisions made by the planner (branching) that we would like to guide are precisely on what instantiated search operator to apply at each meta-state. Thus, they constitute the learning points, from our perspective. In this paper, we learn functions composed of a set of control rules. Each rule will have the format: `if conditions(`s^m`) then select` A_σ^m. That is, if certain conditions on the current meta-state hold, then the planner should apply the corresponding instantiated search operator.

2.2 IPSS planner

IPSS is an integrated tool for planning and scheduling that provides sound plans with temporal information (if run in integrated-scheduling mode). The planning component is a nonlinear planning system that follows a means-ends analysis (see [9] for details). It performs a kind of bidirectional depth-first search (subgoaling from the goals, and executing operators from the initial state), combined with a branch-and-bound technique when dealing with quality metrics. The planner is integrated with a constrains-based scheduler that reasons about time and resource constrains.

In terms of the unified model, it can be described as:

- each meta-state s^m is a tuple $\{s^d, G_p^d, L, g^d, a^d, a_\sigma^d, P\}$ where s^d is the current domain state, G_p^d is the set of pending (open) goals, L is a set of assignments of instantiated actions to goals, g^d is the goal in which the planner is working on, a^d is an action that the planner has selected for achieving g^d, a_σ^d is an instantiated action that the planner has selected for achieving g^d, and P is the current plan for solving the problem
- the initial meta-state, $s_0^m = \{s_0^d, G^d, , , , , \}$
- a terminal state will be of the form $s_T^m = \{s_T^d, , L_T, , , , P\}$ such that $G^d \subseteq s_T^d$, L_T will be the causal links between goals and instantiated actions, and P is the plan
- the set of search operators A^m is composed of (given the current meta-state $\{s^d, G_p^d, L, g^d, a^d, a_\sigma^d, P\}$): "select a goal $g^d \in G_p^d$", "select an action $a^d \in A^d$ for achieving the current goal g^d", "select an instantiation $a_\sigma^d \in A_\sigma^d$ of current action a^d", and "apply the current instantiated action a_σ^d to the current state s^d"

In IPSS, as in PRODIGY, there is a language for defining heuristic function $H(s_m)$ in terms of a set of control rules. Control rules can *select*, *reject*, or *prefer* alternatives (ways of instantiating search operators, A^m_σ). The conditions of control rules refer to queries (called meta-predicates) to the current meta-state of the search $s^m = \{s^d, G^d_p, L, g^d, a^d, a^d_\sigma, P\}$. PRODIGY already provides the most usual meta-predicates, such as knowing whether: some literal l is true in the current state $l \in s^d$ (`true-in-state`), some literal l is the current goal $l = g^d$ (`current-goal`), some literal l is a pending goal $l \in G^d_p$ (`some-candidate-goals`) or some instance is of a given type (`type-of-object`). But, the language for representing heuristics also admits coding user-specific functions.

2.3 TGP planner

TGP is a temporal planner that enhances Graphplan algorithm to handle actions of different durations [7]. Again, we will only use in this paper its planning component, and we leave working with the temporal part for future work.

The planner alternates between two phases: *graph expansion*, it extends a planning graph until the graph has achieved necessary (but potentially insufficient) conditions for plan existence; and *solution extraction*, it performs a backward-chaining search, on the planning graph, for a solution; if no solution is found, the graph is expanded again and a new solution extraction phase starts.

When TGP performs a backward-chaining search for a plan chooses an action that achieves each goal proposition. If it is consistent (nonmutex) with all actions that have been chosen so far, then TGP proceeds to the next goal; otherwise it chooses another action. An action achieves a goal if it has the goal as effect. Instead of choosing an action to achieve a goal at one level, TGP can also choose to persist the goal (selecting the *no-op* action); i.e. it will be achieved the goal in levels closer to the initial state. After TGP has found a consistent set of actions for all the propositions in one level, it recursively tries to find a plan for the action's preconditions and the *persisted* goals. The search succeeds when it reaches level zero. Otherwise, if backtracking fails, then TGP extends one level the planning graph with additional action and proposition nodes and tries again. In terms of the unified model, it can be described as:

- each meta-state s^m is a tuple $\{PG, G^d_p, L, l\}$ where PG is the plan graph, G^d_p is the set of pending (open) goals, L is a set of assignments of instantiated actions to goals (current partial plan), and l is the plan-graph level where search is
- the initial meta-state, $s^m_0 = \{PG_n, G^d, , l_n\}$ 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^d is the set of top level goals, and l_n is the last level generated of the plan graph
- a terminal state will be of the form $s^m_T = \{PG_i, , L, 0\}$ such that the solution plan can be extracted from L (actions contained in L)
- the set of search operators A^m is composed of only one operator (given the current meta-state $\{PG, G^d_p, L, l\}$): "for each goal $g^d \in G^d_p$ select an instantiated action $a^d_\sigma \in A^d_\sigma$ for achieving it". If it persists a goal, the goal will still be in the G^d_p of the successor meta-state. Otherwise, the preconditions of each a^d_σ are added to G^d_p and each g^d is removed from G^d_p. Also, the links (a^d_σ, g^d) are added to L.

2.4 Transferring heuristics from TGP to IPSS

Our proposal to transfer control knowledge between two planners P_1 and P_2 requires five steps: (1) a language for representing heuristics (or function $H_1(s^m)$) for P_1; (2) a method for automatically generating H_1 from P_1 search episodes; (3) a translation mechanism between meta problem-space of P_1 and meta problem-space of P_2, which is equivalent to providing a translation mechanism between H_1 and H_2; (4) a language H_2 for representing heuristics for P_2; and (5) an interpreter (matcher) of H_2 in P_2. Given that IPSS already has a declarative language to define heuristics, H_{Ipss} (step 4), and an interpreter of that language (step 5), we had to define the first three steps. In relation to step 1, it had to be a language as close as possible to the one used in IPSS, so that step 3 could be simplified. Therefore, we built it using as many meta-predicates as possible from the ones in IPSS declarative language. We will now focus on steps 2 and 3, and include some hints on that language.

3 The learning technique

In this section, we describe the ML technique we built, based on EBL, to generate control knowledge from TGP. We have called it GEBL (Graphplan EBL). It generates explanations for the local decisions made during the search process (in the meta problem-space). These explanations become control rules. In order to learn control knowledge for a Graphplan-based planner, we follow a standard four-steps approach: first, TGP solves a planning problem, generating a trace of the search tree; second, the search tree is labelled so that the successful decision nodes are identified; third, control rules are created from two consecutive successful decision points, by selecting the relevant preconditions; and, fourth, constants in the control rules are generalized to variables. In this discussion, the search tree is the one generated while solving the meta problem-space problem. So, each decision point (or node) consists of a meta-state, and a set of successors (applicable instantiated operators of the meta problem-space).

Now, we will present each step in more detail.

3.1 Labelling the search tree

We define three kinds of nodes in the search tree: *success*, *failure* and *memo-failure*. *success* nodes are the ones that belong to a solution path. The *failure* nodes are the ones where there is not a valid assignment for all the goals in the node; i.e. it is not possible to find actions to achieve all the goals with no mutex relation violated among them or the ones in the currently constructed plan. A node also fails if all of its children fail. And if the planner did not expand a node, the node is labeled as *memo-failure*. This can happen when the goals were previously memoized as *nogoods* (failure condition of Graphplan-based planning) at that level.

All nodes are initially labelled as *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 labelled as *success*.

Figure 1 shows an example of a labelled search tree in the Zenotravel domain solving one problem. The Zenotravel domain involves transporting people among cities in planes, using different modes of flight: fast and slow. The example problem consists of transporting two persons: `person0` from `city0` to `city1`, and `person1` from `city1` to `city0`. Therefore, $G^d = \{$(at person1 city0) (at person0 city1)$\}$. There are 7 fuel levels (`fli`) ranging from 0 to 6 and there is a plane initially in `city1` with a fuel level of 3. TGP expands the planning graph until level 5 where both problem goals are consistent (nonmutex). In this example there are no failures and therefore no backtracking. The initial meta-state $s0$ is composed of the expanded plan graph, PG_5, the problem goals G^d, assignments () and level 5. The search algorithm tries to apply an instantiation of the search operator of the meta problem-space (selecting an instantiated action for each goal). So, it finds the instantiated action of the domain problem-space (`debark person0 plane1 city1`) to achieve the goal (`at person0 city1`) and persists the goal (`at person1 city0`). This generates the child node (meta-state $s1$), that has, as the goal set, the persisted goal and the preconditions of the previously selected action (`debark`), i.e. (`at plane1 city1`) (`in person0 plane1`). The pair (assignment) action (`debark person0 plane1 city1`) and goal (`at person0 city1`) is added on top of the child-node assignments. Then, the algorithm continues the search at meta-state $s2$. It finds the action (`fly plane1 city0 city1 fl1 fl0`) to achieve the goal (`at plane1 city1`) and it persists the other goals. That operator generates the child node $s3$, with the preconditions of the action `fly` and the persisted goals. The new pair action-goal is added on top of the currently constructed plan. The algorithm continues until it reaches level 0 where the actions in the assignment set of the last node (meta-state $s5$) represents the solution plan.

Once the search tree has been labelled, a recursive algorithm generates control rules from all pairs of consecutive success nodes (eager learning). GEBL can also learn only from non-default decisions (lazy learning). In this case, it only generates control rules if there is, at least, one *failure* node between two consecutive success nodes. The *memo-failure* nodes in lazy learning are not considered, because the planner does not explore them. Also, from a "lazyness" perspective they behave as success nodes. Lazy learning usually is more appropriate when the control knowledge is obtained and applied to the same planner to correct only the wrong decisions.

3.2 Generating control rules

As we said before, control rules have the same format as in PRODIGY. The module that generates control rules receives as input two consecutive success decision points (meta-states) with their goal and assignment sets. There are two kinds of possible rules learned from them: a `select goals` rule to select the goal that persists in the decision point (when only one goal persists) and `select operator` rules to select the instantiated actions that achieve the goals in the decision point (one rule for each achieved goal).

```
s0:
Level:(5) SUCCESS
Goals=((at person1 city0) (at person0 city1))
Assignments=NIL

s1:
Level:(4) SUCCESS
Goals=((at person1 city0) (at plane1 city1) (in person0 plane1))
Assignments=(((debark person0 plane1 city1) (at person0 city1)))

s2:
Level:(3) SUCCESS
Goals=((at person1 city0) (in person0 plane1) (fuel-level plane1 fl1)
       (at plane1 city0))
Assignments=(((fly plane1 city0 city1 fl1 fl0) (at plane1 city1))
             ((debark person0 plane1 city1) (at person0 city1)))

s3:
Level:(2) SUCCESS
Goals=((in person1 plane1) (fuel-level plane1 fl1) (at plane1 city0)
       (at person0 city0))
Assignments=(((board person0 plane1 city0) (in person0 plane1))
             ((debark person1 plane1 city0) (at person1 city0))
             ((fly plane1 city0 city1 fl1 fl0) (at plane1 city1))
             ((debark person0 plane1 city1) (at person0 city1)))

s4:
Level:(1) SUCCESS
Goals=((in person1 plane1) (at person0 city0) (fuel-level plane1 fl2)
       (at plane1 city1))
Assignments=(((fly plane1 city1 city0 fl2 fl1) (fuel-level plane1 fl1))
             ((board person0 plane1 city0) (in person0 plane1))
             ((debark person1 plane1 city0) (at person1 city0))
             ((fly plane1 city0 city1 fl1 fl0) (at plane1 city1))
             ((debark person0 plane1 city1) (at person0 city1)))

s5:
Level:(0) SUCCESS
Goals=((at person1 city1) (at person0 city0) (fuel-level plane1 fl3)
       (at plane1 city1))
Assignments=(((board person1 plane1 city1) (in person1 plane1))
             ((fly plane1 city1 city0 fl2 fl1) (fuel-level plane1 fl1))
             ((board person0 plane1 city0) (in person0 plane1))
             ((debark person1 plane1 city0) (at person1 city0))
             ((fly plane1 city0 city1 fl1 fl0) (at plane1 city1))
             ((debark person0 plane1 city1) (at person0 city1)))
```

Figure 1: Example of TGP success search tree.

As an example, from the first two decision points $s0$ and $s1$ of the example in Figure 1, two rules would be generated; one to select the goal (`at person1 city0`) and another one to select the operator (`debark person0 plane1 city1`) to achieve the goal (`at person0 city1`).

In order to make the control rules more general and reduce the number of `true-in-state` meta-predicates, a goal regression is carried out, as in most EBL techniques [4]. Only those literals in the state which are required, directly or indirectly, by the preconditions of the instantiated action involved in the rule (the action that achieves `goal`) are included.

Figure 2 shows the `select goal` rule generated from the first two decision points in the example of Figure 1. This rule chooses between two goals of moving persons from one city to another (the arguments of the meta-predicates `target-goal` and `some-candidate-goals`). One person <`person1`> is in a city where there is a plane <`plane1`> with enough fuel to fly. The rule selects to work on the goal referring to this person giving that s/he is in the same city as the plane.

Figure 3 shows the `select operator` rule generated from the example above. This rule selects the action `debark` for moving a person from one city to another. IPSS would try

```
(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 2: Example of `select goals` rule in the Zeno-travel domain.

(by default) to debark the person from any plane in the problem definition. The rule selects the most convenient plane; a plane that is in the same city as the person with enough fuel to fly.

```
(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 (debark <person0> <plane1> <city1>)))
```

Figure 3: Example of `select operator` rule in the Zeno-travel domain.

4 Translation of learned knowledge

Given that meta-states and operators of the meta problem-space of two different planners differ, in order to use the generated knowledge in P_1 (TGP) by P_2 (IPSS), we have to translate the control rules generated in the first language to the second one. The translation should have in mind 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 conditions in the left-hand side of rules, and types of decisions - operators of the meta problem-space). We have to consider the translation of the left-hand side of rules (conditions referring to meta-states) and the right-hand side of rules (selection of a search operator of the meta problem-space).

Therefore, in relation to the translation of the right-hand side of control rules, we found that the equivalent search operators between IPSS and TGP are:

- to decide which goal to work on first. When TGP selects the *no-op* to achieve a goal, this is equivalent to persist the goal (it will be achieved in levels closer to the initial state). IPSS has an equivalent search operator for choosing a goal from the pending goals.
- to choose an instantiated operator to achieve a particular goal (both planners have that search operator, though, in the case of IPSS it splits it in two: select an action, and select an instantiated action).

So, according to IPSS language to define control rules, there are three kinds of rules that can be learned in TGP to guide the IPSS search process: `select goals`, `select operator` (select an action) and `select bindings` (select an instantiated action) rules.

The equivalence between meta-states is not straightforward (for translating the conditions of control rules). When IPSS selects to apply an instantiated action, the operator of the meta problem-space changes the state s^d and the action is added at the end of the current plan P. However, when TGP selects an instantiated action to achieve a goal, it is added at the beginning of the plan L (given that the search starts at the last level) and TGP does not modify the state s^d. The difficulty arises in defining the state s^d that will create the `true-in-state` conditions of the control rules. When the rules are learned in TGP, we considered two possibilities: the simplest one is that the state s^d is just the problem initial-state s_0^d; and, in the second one, we assume that when TGP persists a goal that goal would have already been achieved in IPSS, so the state s^d is the one reached after executing the actions needed to achieve the persisted goals in the TGP meta-state. To compute it, we look in the solution plan, and progress the problem initial-state according to each action effects in such partial plan.

Equivalent meta-states are computed during rule generation and a translator makes several transformations after the learning process finishes. The first one is to split the select-operator control rules in two: one to select the action and another one to select its instantiation. The second transformation is to translate `true-in-state` meta-predicates referring to variable types into `type-of-object` meta-predicates.[1] 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.

5 Experimental Results

We carried out some experiments to show the usefulness of the approach. Our goal is on transferring learned knowledge: that GEBL is able to generate control knowledge that improves IPSS planning task solving unseen problems. We compare our learning system with HAMLET, that learns control rules from IPSS problem-solving episodes. According to its authors, HAMLET performs an EBL step followed by induction on the control rules.

In these experiments, we have used four commonly used benchmark domains from the repository of previous planning competitions:[2] Zenotravel, Miconic, Logistics and Driverlog (the STRIPS versions, since TGP can only handle the plain STRIPS version).

In all the domains, we trained separately both HAMLET and GEBL with randomly generated training problems and

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

[2]http://www.icaps-conference.org

tested against a different randomly generated set of test problems. The number of training problems and the complexity of the test problems varied according to the domain difficulty for IPSS. We should train both systems with the same learning set, but GEBL learning mode is eager; it learns from all decisions (generating many rules), and it does not perform induction. So, it has the typical EBL problems (the utility problem and the overly-specific generated control knowledge) that can be attenuated using less training problems with less goals. However, HAMLET incorporates an inductive module that diminishes these problems. Also, it performs lazy learning, since HAMLET obtains the control knowledge for the same planner where the control knowledge is applied. Therefore, if we train HAMLET with the same training set as GEBL, we were loosing HAMLET capabilities. So, we have opted for generating an appropriate learning set for both systems: we train HAMLET with a learning set, and we take a subset from it for training GEBL. The number and complexity (measured as a number of goals) of training problems are shown in Table 1. We test against 100 random problems in all the domains (except in the Miconic that we used the 140 problems defined for the competition) but varying the number of goals: from 2 to 13 goals in the Zenotravel, 3 to 30 in the Miconic, 2 to 6 in the Driverlog and 1 to 5 in the Logistics.

Table 1 shows the average of solved (*S*) test problems by IPSS without using control knowledge (*IPSS*), using the HAMLET learned rules (*HAMLET*) and using the GEBL learned rules (*GEBL*). Column *R* displays the number of generated rules and column *T* displays the number of training problems, together with their complexity (range of number of goals). The time limit used in all the domains was 30 seconds except in the Miconic domain that was 60s.

Domain	IPSS	HAMLET			GEBL		
	S	S	R	T	S	R	T
Logistics	12%	25%	16	400 (1-3)	57%	406	200 (2)
Driverlog	26%	4%	7	150 (2-4)	77%	71	23 (2)
Zenotravel	37%	40%	5	200 (1-2)	98%	14	200 (1-2)
Miconic	4%	100%	5	10 (1-2)	99%	13	3 (1)

Table 1: Results of percentage of solved random problems with and without heuristics.

The results show that the rules learned by GEBL greatly increase the percentage of problems solved in all the domains compared to HAMLET rules, and plain IPSS, except in the Miconic domain where HAMLET rules are slightly better. Usually, learning improves the planning task, but it can also worsen it (as HAMLET rules in the Driverlog domain). The reasons for this behaviour are the intrinsic problems of HAMLET learning technique: EBL techniques have the utility problem and in inductive techniques generalizing and specializing incrementally do not assure the convergence, unless they continuously check for performance against a problem set.

6 Conclusions

This paper presents an approach to transfer control knowledge (heuristics) learned from one planner to another planner that uses a different planning technique and bias. First, we have defined a model based on meta problem-spaces that permits to reason about the decisions made during search by the different planners. Then, for each decision we propose to learn control knowledge (heuristics) to guide that planner. But, instead of applying the learned knowledge to that planner, we focus on transferring that knowledge to another planner, so that it can use it.

We have implemented a learning system based on EBL, GEBL, that is able to obtain these heuristics from TGP, a temporal Graphplan planner, translate them into IPSS, and improve IPSS planning task. To our knowledge, this is the first system that is able to transfer learned knowledge between two planning techniques. We have tested our approach in four commonly used benchmark domains and compare it with HAMLET, an inductive-deductive learning system that learn heuristics in PRODIGY. In all the domains, GEBL rules notably improve IPSS planning task and outperform HAMLET, except in the Miconic domain where the behaviour is similar.

We intend to show in the future that the approach is general enough, so that it works also with other combinations, including to learn in IPSS and transfer to TGP.

References

[1] B. Bonet and H. Geffner. Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to MDPs. In *Proceedings of ICAPS'06*, 2006.

[2] D. Borrajo and M. Veloso. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review Journal. Special Issue on Lazy Learning*, 1997.

[3] T. Bylander. Complexity results for planning. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, 1991.

[4] G. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1986.

[5] F. Fernández and M. Veloso. Probabilistic policy reuse in a reinforcement learning agent. In *Proceedings of the AAMAS'06*, 2006.

[6] T. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1986.

[7] D. Smith and D. Weld. Temporal planning with mutual exclusion reasoning. In *Proceedings of the IJCAI'99*, pages 326–337, 1999.

[8] M. E. Taylor and P. Stone. Behavior transfer for value-function-based reinforcement learning. In *The Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, 2005.

[9] M. Veloso, J. Carbonell, A. Pérez, D. Borrajo, E. Fink, and J. Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical AI*, 1995.

[10] T. Zimmerman and S. Kambhampati. Learning-assisted automated planning: Looking back, taking stock, going forward. *AI Magazine*, 2003.