Transfer of Learned Heuristics among Planners*

Susana Fernández, Ricardo Aler and **Daniel Borrajo**
Departamento de Informática, Universidad Carlos III de Madrid
Avda. de la Universidad, 30. Leganés (Madrid). Spain
email:{susana.fernandez, ricardo.aler, daniel.borrajo}@uc3m.es

Abstract

This paper presents a study on the transfer of learned control knowledge between two different planning techniques. We automatically learn heuristics (usually, in planning, heuristics are also named control knowledge) from one planner search process and apply them to a different planner. The goal is to improve this second planner efficiency solving new problems, i.e. to reduce computer resources (time and memory) during the search, or to improve quality solutions. The learning component is based on a deductive learning method (EBL) that is able to automatically acquire control knowledge by generating bounded explanations of the problem solving episodes in a graph-plan based planner. Then, we transform the learned knowledge so that it can be used by a bidirectional planner.

Introduction

Planning is a problem solving task that consists of given a domain theory (set of states and operators) and a problem (initial state and set of goals) obtain a plan (set of operators and a partial order of executing among them) such that, when this plan is executed, it transforms the initial state in a state where all the goals are achieved. Planning has been shown to be a hard computational task (Backstrom 1992), whose complexity increases if we also consider trying to obtain "good" quality solutions (according to a given user-defined quality metric). Therefore, redefining the domain theory and/or defining heuristics for planning is necessary if we want to obtain solutions to real world problems.

Planning technology has experienced a big advance, ranging from totally-ordered planners, partially-ordered planners, planning based on plan graphs, planning based on SAT resolution, heuristic planners,HTN planning (Hierarchical Task Networks) to planning with uncertainty or planning with time and resources (Ghallab, Nau, & Traverso 2004). Despite this advance, there is no universally superior strategy of search commitments for planning that outperforms all others in all planning problems and domains. For example, Veloso and Blythe demonstrated that totally-ordered planners sometimes have an advantage over partially-ordered planners (Veloso & Blythe 1994) and Nguyen and Kambhampati implement a partially-ordered planner called REPOP that *convincingly outperforms Graphplan in several "parallel" domains* (Nguyen & Kambhampati). Kambhampati and Srivastava implemented the Universal Classical Planner for unifying the classical plan-space and state-space planning approaches (Kambhampati & Srivastava 1996).

Machine learning (ML) techniques applied to planning range from macro-operators acquisition, case-based reasoning, rewrite-rules acquisition, generalized policies, deductive approaches for learning heuristics (EBL), learning domain models, to inductive approaches (ILP based) (Zimmerman & Kambhampati 2003). A classification of these techniques separates them into the ones whose goal is acquiring a domain model (theory), and the ones that acquire heuristics. All the systems that learn heuristics are implemented just for one planner and the learned knowledge only improves the planning task of that planner.

Instead of implementing an Universal Planner we propose to generate "universal" heuristics obtained from different planning techniques. That is, to learn control knowledge for each domain from the planning paradigm that better behaves in that domain to generate a domain-dependent control-knowledge repository that encompasses the advantages of each planning techique and overcomes its weakness. The first step in this ambitious challenge is to use the learned heuristics on one specific planner for improving the planning task of another planner. Although each planner uses its own strategy to search for solutions, some of them share some common decision points, like 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 from a graph-plan based planner, TGP (Smith & Weld 1999), to IPSS that integrates a planner and a scheduler (Rodríguez-Moreno *et al.* 2004) which is based on PRODIGY (Veloso *et al.* 1995). The main goal of the PRODIGY project, which spans almost 20 years including current research (Fink & Blythe 2005), has been the study of ML techniques on top of a problem solver. It incorporates declarative language to define heuristics in terms of a set of control rules that specify how to make decisions in the search tree. Our goal is to automatically generate these IPSS

*This work has been partially supported by the Spanish MCyT under project TIC2002-04146-C05-05, MEC project TIN2005-08945-C06-05, and CAM-UC3M project UC3M-INF-05-016.

control rules by applying ML in a graph-plan based planner and then translating 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 (Borrajo & Veloso 1997), 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 (Mitchell, Keller, & Kedar-Cabelli 1986).

The paper is organized as follows. Next section provides necessary background on the planning techniques involved. Then we compare TGP and IPSS search algorithms before describing the implemented learning system and the learned control rules. Afterward, we discuss related work. Finally, some experimental results are shown and conclusions are drawn.

Background

This section is organized as follows. First, IPSS is described focusing only on the planning component and the heuristics it can incorporate. Second, TGP is described focusing on the search algorithm.

IPSS planner

IPSS is an integrated tool for planning and scheduling (Rodríguez-Moreno *et al.* 2004).The planning component is a nonlinear planning system that follows a means-ends analysis (see (Veloso *et al.* 1995) for details on the planning algorithm). 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 constraints-based scheduler (Cesta, Oddi, & Smith 2002) that reasons about time and resource constraints.

From a perspective of heuristics acquisition, IPSS planning cycle, involves several decision points (all can lead to backtracking and, therefore, to learning points):

- *select a goal* from the set of pending goals; i.e. preconditions of previously selected operators that are not true in the current state (by default, it selects the order on which to work on goals according to their definition in the problem file);
- *choose an operator* to achieve the selected goal (by default, they are explored according to their position in the domain file);
- *choose the bindings* to instantiate the previously selected operator (by default, it selects first the bindings that lead to an instantiated operator that has more preconditions true in the current state);
- *apply* an instantiated operator whose preconditions are satisfied in the current state (default decision in the version we are using), or continue *subgoaling* on another pending goal.

When an instantiated operator is *applied* it becomes the next operator in the solution plan and the current state is updated to account for its effects (bidirectionality).

Each decision point above can be guided by control rules that specify how to make such decision. The conditions (left-part) of control rules refer to facts that can be queried to the meta-state of the search. These queries are called meta-predicates. PRODIGY already provides the most usual ones, such as knowing whether: some literal is true in the current state (`true-in-state`), some literal is a pending goal (`current-goal`), there is one pending goal between a literal set (`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.

TGP planner

TGP is a temporal planner that enhances Graphplan algorithm (Blum & Furst 1995) to handle actions of different durations (Smith & Weld 1999). Both planners alternate between two phases:

- Graph expansion: they extend a planning graph until it has achieved necessary (but potentially insufficient) conditions for plan existence. Binary "mutex" constraints are computed and propagated to identify operators whose preconditions cannot be satisfied at the same time. Two action instances are mutex if one action deletes a precondition or effect of another, or the actions have preconditions that are mutually exclusive. Two propositions are mutex if all ways of achieving the propositions are pairwise mutex.
- Solution extraction: they perform 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.

TGP changes the multi-level Graphplan planning graph altogether to a graph with temporal action and proposition nodes. Action, proposition, mutex and nogood structures[1] are all annotated with a numeric label field: for proposition and action nodes this number denotes the first planning graph level at which they appear. For mutex or nogood nodes, the label marks the last level at which the relation holds. When using this representation, there is no longer any need to have actions take unit time. Instead, labels can be real numbers denoting start time leading to temporal reasoning. The zeroth level consists of the propositions that are true in the initial state of the planning problem and the actions whose preconditions are present on that level. The action effects generate proposition nodes at level d, where d is the action duration. Action and effect proposition nodes are similarly generated in consecutive levels until all goal propositions are present and none are pairwise mutex. Then, TGP performs a backward chaining search for a plan at that level. For each goal proposition, TGP chooses an action that achieves the goal. If it is consistent (nonmutex) with all actions that have been chosen so far, then TGP proceeds to the

[1] When a set of goals for a level is determined to be unsolvable, they are *memorized* generating the so-called *nogoods*, *memoizations* or *memos* at that level.

next goal; otherwise it chooses another action. An action achieves a goal if it has the goal as effect and the time when the action first appears in the planning graph is equal or less than the current time (search level). Instead of choosing an action to achieve a goal at one level, TGP can also choose to persist the goal; i.e. it will be achieved 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.

The search tree can be viewed as search nodes, or decision points, where each node is composed of a goal set and an action set (currently constructed plan). The algorithm finds actions to achieve some of the goals in a decision-point goal set and it persists others. The child-node goal set is the union of the preconditions of all the actions found in the decision point for achieving the *achieved* goals and the *persisted* goals. The new actions are added to the partial plan (child-node action set). All possible combinations of achieved and persisted goals constitute the different branches of the search tree.

IPSS versus TGP

In order to understand the search performed by TGP and IPSS we use a simple example in the Zenotravel domain. The Zenotravel transportation domain involves transporting people among cities in planes, using different modes of movement: fast and slow Suppose a problem consisting on transporting two persons: `person0` from `city0` to `city1`, and `person1` from `city1` to `city0`. There are 7 fuel levels (`fli`) ranging from 0 to 6 and there is a plane in `city1` with a fuel level of 3. Assuming that actions have duration 1, TGP expands the planning graph until time 5 where both problem goals are consistent (nonmutex). Figure 1 shows the TGP search tree for solving the problem. In this example there are no failures and therefore no backtracking. The first decision point has the problem goals `((at person1 city0) (at person0 city1))` and actions (NIL). The search algorithm starts at time (level) 5 finding the action `(debark person0 plane1 city1)` to achieve the goal `(at person0 city1)` and persisting the goal `(at person1 city0)`. This generates the child node represented in line *Time:(4)* that has the persisted goal and the preconditions of the previously selected operator (`debark`), i.e. `(at plane1 city1) (in person0 plane1)`, as the goal set. The action `(debark person0 plane1 city1)` is added on top of the child-node action set. Then, the algorithm starts a new search at time 4. It finds the action `(fly plane1 city0 city1 fl1 fl0)` to achieve the goal `(at plane1 city1)` and it persists the other goals. That generates the child-node in line *Time:(3)* with the preconditions of the action `fly` and the persisted goals. The new action is added on top of the currently constructed plan (child-node action set). The algorithm continues until it reaches level 0, and the last node action set represents the solution plan.

Figure 2 shows the IPSS search tree for solving the same problem. The algorithm starts by selecting a goal to work on. By default, IPSS selects it according to their definition in the problem file, so it selects the goal `(at person0 city1)`. Then, it chooses the operator `debark` to achieve it (node 6) and the bindings (i.e. lists of variable/values) to instantiate the operator (node 7), `(debark person0 plan1 city1)`. Its preconditions are added to the pending goals list, `(in person0 plane1)`. The second precondition, `(at plane1 city1)`, is not added because it is already true in the current planning state.[2] The following decision points chooses the goal `(in person0 plane1)` (node 8) to work on, selecting the instantiated operator `(board person0 plan1 city0)` to achieve it (nodes 9 and 10). Its preconditions generate a new goal node, `(at plane1 city0)` (node 12) and again, it chooses an operator to achieve it (node 14) and the bindings (node 15), `(fly plane1 city1 city0 fl3 fl2)`. At this point, all the preconditions of the instantiated operator are satisfied in the current state and IPSS can apply the instantiated operator (default decision) or continue *subgoaling* on another pending goal. When an instantiated operator is *applied* it becomes the next operator in the solution plan and the current state is updated to account for its effects. Therefore, IPSS applies first the operator `fly` (node 16) and then the operator `board` (node 17) generating the first actions in the solution plan, `(fly plane1 city1 city0 fl3 fl2)` and `(board person0 plane1 city0)`. The algorithm continues combining goal-directed backward chaining with simulation of plan execution until it solves the problem.

Comparing both algorithms there are two types of decision shared by both planners:

- To decide which goal to work on first. TGP achieves goals or persists them at each decision point and IPSS has decision points for choosing goals.
- To choose an instantiated operator to achieve a goal.

For example, in the first decision point in Figure 1 at time 5, TGP persists the goal `(at person1 city0)`, that it is achieved at time 2, closer to the initial state. However, IPSS first decision point selects the goal `(at person0 city1)` because it is the first one in the problem definition file. This is not the best option, since it implies an unnecessary flight. A rule learned from TGP first decision point could guide IPSS to select the correct goal. On the other hand, all the decision points where TGP selects an operator to achieve a goal have two equivalent decision points in IPSS: one to select the operator, and another one to select the bindings, except for the goal `(fuel-level plane1 fl1)` that it never becomes an IPSS pending goal, because it is already true in the current planning state.

Therefore, there are three kinds of rules that can be learned in TGP to guide the IPSS search process: `SELECT GOALS`, `SELECT OPERATOR` and `SELECT BINDINGS` rules. The following section explains the learning mechanism to learn those rules.

[2] IPSS only tries to achieve a literal if it is not true in the current state. This makes IPSS non complete in some problems (Fink & Blythe 2005). The *select* rules make IPSS even more incomplete.

```
Time:(5)
Goals=((at person1 city0) (at person0 city1))
Actions=NIL

Time:(4)
Goals=((at person1 city0) (at plane1 city1) (in person0 plane1))
Actions=(((debark person0 plane1 city1) (at person0 city1)))

Time:(3)
Goals=((at person1 city0) (in person0 plane1) (fuel-level plane1 f
       (at plane1 city0))
Actions=(((fly plane1 city0 city1 fl1 fl0) (at plane1 city1))
         ((debark person0 plane1 city1) (at person0 city1)))

Time:(2)
Goals=((in person1 plane1) (fuel-level plane1 fl1) (at plane1 city
       (at person0 city0))
Actions=(((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)))

Time:(1)
Goals=((in person1 plane1) (at person0 city0) (fuel-level plane1 f
       (at plane1 city1))
Actions=(((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)))

Time:(0)
Goals=((at person1 city1) (at person0 city0) (fuel-level plane1 fl
       (at plane1 city1))
Actions=(((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.

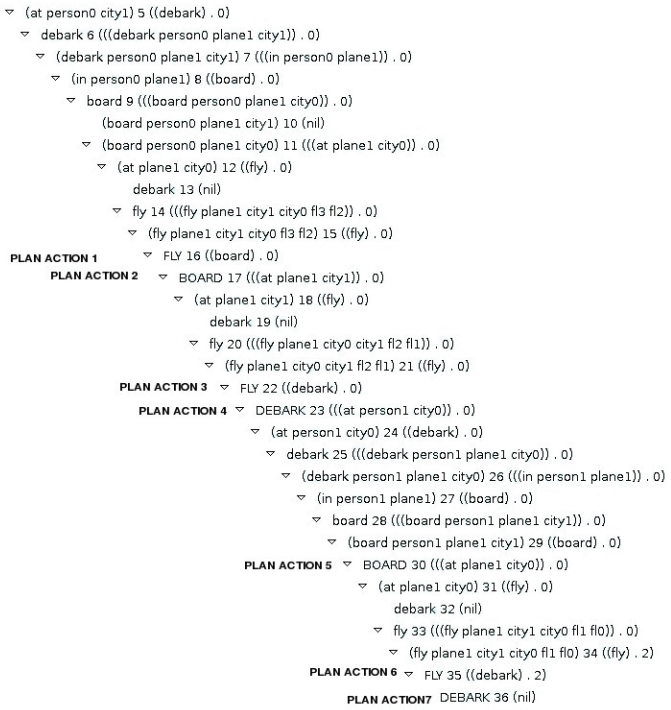

Figure 2: Example of IPSS search tree.

The learning mechanism

In this section, we propose a ML mechanism, based on EBL, to obtain control knowledge from TGP. We have called it GEBL (Graphplan EBL). It generates explanations for the local decisions made during the search process. These explanations become control rules. In order to learn control knowledge for a graph-based planner, we follow a three standard steps approach:

1. TGP runs a planning problem, generating a trace of the search tree. The planning search tree is labelled so that the successful decision nodes are identified.
2. From two consecutive successful decision points control rules are created, by selecting the relevant preconditions.
3. Constants in the control rules are generalized, so that they can be applied to other problems.

Now, we will present each step in more detail.

Labelling the search tree

There are three kinds of nodes in the search tree:

- *success*, if the node belongs to a solution path. Its actions form the solution.
- *failure*, if it is a failed node. The failure takes place when there is not a valid assignment for all the goal set 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 its children fail.
- *memo-failure*, if the planner did not expand the node because the goals are memorized as *nogoods* 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*.

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 and 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.

Generating control rules

Control rules have the same format as in PRODIGY. One of the most important decisions to be made in systems that learn heuristics for planning refer to the language to be used to represent target concepts. The condition part is made of a conjunction of meta-predicates which check for local features of the search process. The problem is that these features must be operational in the IPSS search process (they should be meaningful to IPSS search process). So, it is necessary to find equivalences between TGP and IPSS planning

meta states during the search. This is not a trivial task because IPSS changes the planning state when executing parts of the currently constructed plan (bidirectional planning) while TGP does not modify the planning state; it only uses the information gathered in the planning graph that remains unchanged during each search episode. We make two assumptions that are not always true; they are more accurate when there are fewer goals to solve:

1. In the equivalent IPSS planning meta state where TGP persists goals, those goals would have already been achieved. TGP starts the search in the last graph level, finding first the last actions in the final plan and persisting the goals that will be the first actions in the plan (see Figure 1). However, in IPSS the sooner an operator is *applied* during the search, the sooner it appears in the solution plan (see Figure 2). To apply an operator implies to change the current planning state according to its effects.
2. In the equivalent IPSS planning meta state where TGP achieves more than one goal, those goals would be pending goals. In the example shown in Figure 1, if we use a rule for selecting the goal `(at person1 city0)` this assumption is only true in the first two decision points.

The module that generates control rules receives as input two consecutive success decision points with their goal and action sets. There are two kinds of rules learned from them:

1. `SELECT GOALS` rules: if there is only one persisted goal, a rule for selecting it is generated.
2. `SELECT OPERATOR` rules: it generates one of these rules for each one of the achieved goals in the decision point.

From the first two decision points of the example in Figure 1, two rules would be generated; one to select the goal `(at person2 city0)` and another one to select the operator `(debark person1 plane1 city2)` to achieve the goal `(at person1 city2)`.

`SELECT GOALS` rules have the following meta-predicates in the condition part that check for local features of the IPSS search process:

- `(TARGET-GOAL goal1)`: to identify a pending goal the planner has to achieve. Its argument, `goal1`, is the persisted goal.
- A `TRUE-IN-STATE` meta-predicate for each literal that is true in the planning state. 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 (DeJong & Mooney 1986). Only those literals in the state which are required, directly or indirectly, by the preconditions of the operator involved in the rule (the operator that achieves `goal1`) are included. The following section explains how we perform the goal regression.
- `SOME-CANDIDATE-GOALS (rest-goals)`: to identify what other goals need to be achieved. Its argument, `rest-goals`, are the achieved goals in this decision point.

Figure 3 shows the `SELECT GOAL` rule generated from the first 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 is in a city where there is a plane with enough fuel to fly. The rule selects to work on the goal referring to the person who is in the same city as the plane.

```
(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 3: Example of `SELECT GOALS` rule in the Zeno-travel domain.

`SELECT OPERATOR` rules have the following meta-predicates:

- `(CURRENT-GOAL goal1)`: to identify which goal the planner is currently trying to achieve. Its argument, `goal1`, is the achieved goal of the TGP decision point.
- A `TRUE-IN-STATE` meta-predicate for each literal that is true in the planning state. A goal regression is also carried out on the operator involved in the rule like in the `SELECT GOALS` rules (see the following section).
- `SOME-CANDIDATE-GOALS (rest-goals)`: to identify what other goals need to be achieved. Its argument, `rest-goals`, are the other achieved goals in this TGP decision point.

Figure 4 shows the `SELECT OPERATOR` rule generated from the example above. This rule selects the operator `debark` for moving a person from one city to another. IPSS would try (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.

The TGP rules cannot directly be used in IPSS. It is necessary to build a translator to PRODIGY control-rules syntax. The translator makes two transformations: to split the operator rules in two, one to select the operator and another one to select its bindings; and, to translate `TRUE-IN-STATE` meta-predicates referring to variable types into `TYPE-OF-OBJECT` meta-predicates.[3]

[3] TGP does not account for variable types explicitly; it represents them as initial state literals. However, IPSS domains require

```
(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 4: Example of SELECT OPERATOR rule in the Zenotravel domain.

Goal regression

Before performing the goal regression, we have to define the planning state, the equivalent IPSS meta state. We consider two possibilities: the simplest one is that the planning state is just the problem initial-state; or, applying our first assumption, that the meta state is the one reached after executing the actions needed to achieve the persisted goals in the TGP decision point. To compute it, we look into the solution plan (the learning starts after TGP solves a problem); i.e. the partial plan where all the persisted goals are achieved. Then, we progress the problem initial-state according to each action effects in such partial plan.

The algorithm to obtain the regressed state R of an action A_i considering a planning state S and a solution plan P is shown in Figure 5.

Q = preconditions (A_i)
$R = \varnothing$
while $Q \neq \varnothing$
  $q = 1^{st}$ element of Q
  if $q \in S$ then $R = R \cup q$
  else
    A_q = action which achieve q according with P
    Q_r = preconditions (A_q)
    $Q = Q \cup Q_r$
  remove q from Q

Figure 5: Goal regression algorithm.

The planning state considered in the SELECT GOALS rules is just the problem initial-state and in the SELECT OPERATOR rules, by default, it is the represented state explained above. The goal regression is also carried out on the arguments of the SOME-CANDIDATE-GOALS meta-predicates. The regressed state of a goal is the regressed state of the action that achieves that goal according with the solution plan.

variable type definitions as in typed PDDL.

Experimental Results

We carried out some experiments to show the usefulness of our 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.

In these experiments, we have used four commonly used benchmark domains from the repository of previous planning competitions:[4] Zenotravel, Miconic, Logistics and Driverlog (the STRIPS versions, since TGP can only handle the plain STRIPS version). In the Miconic domain, the goal is to move passengers from one floor to another with an elevator. In the Logistics domain, packages have to be moved from one location to another among cities. There are two types of carriers: planes, that can move packages between two airports; and trucks, that can only move packages among locations in the same city. And the Driverlog domain involves driving trucks for delivering packages. The trucks require drivers, who must walk from one place to another. The paths for walking and the roads for driving form different maps on the locations. The Logistics and the Driverlog domains are specially hard for IPSS.

In all the domains, we trained separately both HAMLET and GEBL with randomly generated training problems and tested against a different randomly generated set of test problems. Before testing, we translated the GEBL control rule sets, as it is explained in the previous section, and we also refined them by deleting 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 as HAMLET does.

The number of training problems and the complexity of the test problems varied according to the domain difficulty for IPSS. We should train both system 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 sets used in the experiments are:

- Zenotravel: we used 200 training problems of 1 or 2 goals to train HAMLET. GEBL was trained with the problems solved by TGP, 110 in total. For testing, we used 100 problems ranging from 2 to 13 goals.
- Miconic: we used the problems defined in the planning competition; 10 for training HAMLET of 1 or 2 goals, a

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

subset of 3 of them for training GEBL of 1 goal, and 140 for testing ranging from 3 to 30 goals.

- Driverlog: we used 150 training problems from 2 to 4 goals for HAMLET and the 2 goals ones to train GEBL, 23 in total. For testing, we used 100 problems ranging from 2 to 6 goals.
- Logistics: we used 400 training problems from 1 to 3 goals in HAMLET and the first 200 problems of 2 goals for training GEBL. For testing, we used 100 problems ranging from 1 to 5 goals.

Table 1 shows the average of solved (*Solved*) test problems by IPSS without using control knowledge (*IPSS*), using the HAMLET learned rules (*HAMLET*) and using the GEBL learned rules (*GEBL*). Column *Rules* displays number of generated rules. The time limit used in all the domains was 30 seconds except in the Miconic domain that was 60s.

Domain	IPSS	HAMLET		GEBL	
	Solved	Solved	Rules	Solved	Rules
Zenotravel	37%	40%	5	98%	14
Logistics	12%	25%	16	57%	406
Driverlog	14%	3%	7	77%	115
Miconic	4%	100%	5	99%	13

Table 1: Results of percentage of IPSS 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 worse behaviour are the intrinsic problems of HAMLET learning techniques: EBL techniques have the utility problem and in inductive techniques generalizing and specializing incrementally do not assure the convergence.

Related Work

Kambhampati (Kambhampati 2000) applied EBL from failures in Graphplan to learn rules that are valid only in the context of the current problem and do not learn *general rules* that can be applied to other problems. When a set of goals for a level is determined to be unsolvable, instead of *memorizing* all of them in that search node, Kambhampati explains the failure restricting the number of *memorized* goals to the ones that really made the valid allocation of actions impossible. GEBL generates explanations for the local decisions made during the search process that lead to the solution, and it learns from success decision points. These explanations include information about the problem initial-state, so the rules can be applied to other problems.

Upal (Upal 2003) implemented an analytic learning technique to generalize Graphplan *memoizations* including assertions about the initial conditions of the problem and assertions about the domain operators. He performs a static analysis of the domain before starting the planning process to learn general memos. The general memos are instantiated when solving a problem in the context of that problem. Our system differs on the learning method (EBL instead of domain analysis) and on the generated control knowledge. We learn heuristics for guiding the search through the correct branches that can be transferred to other planning techniques, no general memos.

As far as we know, GEBL is the first system 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 (Sutton, Precup, & Singh 1999) obtained in a learning task can be used to improve future learning processes. The knowledge transferred can range from value functions (Taylor & Stone 2005) to complete policies (Fernández & Veloso 2006).

Conclusions

This papers presents an approach to transfer control knowledge (heuristics) learned from a graph-plan based planner to another planner that uses a different planning technique. 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 compared it with HAMLET, an inductive-deductive learning system that learns heuristics in PRODIGY. In all the domains, GEBL rules notably improve IPSS planning task and outperforms HAMLET, except in the Miconic domain where the behaviour is almost equal.

The aim of the paper is to test if transferring learned heuristics among planners is possible. The experiments confirm that heuristics learned from a graph-plan based planner improve a bidirectional planner. The scheme proposed for the test in its current implementation depends on the planners and on the learning system used. However, we intend to show in the future that it is general enough to work with other combinations, including to learn in IPSS and transfer to TGP. In some initial experiments, that we do not report here because they are not the objective of this paper, we applied the control knowledge learned in TGP to improve TGP itself. However, even if this learned knowledge helps IPSS it does not help TGP. So, we are still working on the different topic of learning in a graph-plan based planner control knowledge to improve its own performance.

References

Backstrom, C. 1992. *Computational Complexity of Reasoning about Plans*. Ph.D. Dissertation, Linkoping University, Linkoping, Sweden.

Blum, A., and Furst, M. 1995. Fast planning through planning graph analysis. In Mellish, C. S., ed., *Proceedings of the 14th International Joint Conference on Artificial Intelli-*

gence, IJCAI-95, volume 2, 1636–1642. Montréal, Canada: Morgan Kaufmann.

Borrajo, D., and Veloso, M. M. 1997. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review Journal. Special Issue on Lazy Learning* 11(1-5):371–405.

Cesta, A.; Oddi, A.; and Smith, S. 2002. A Constrained-Based Method for Project Scheduling with Time Windows. *Journal of Heuristics* 8(1).

DeJong, G., and Mooney, R. J. 1986. Explanation-based learning: An alternative view. *Machine Learning* 1(2):145–176.

Fernández, F., and Veloso, M. 2006. Probabilistic policy reuse in a reinforcement learning agent. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06)*.

Fink, E., and Blythe, J. 2005. Prodigy bidirectional planning. *Journal of Experimental & Theoretical Artificial Intelligence* 17(3):161–200(40).

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning - Theory and Practice*. San Francisco, CA 94111: Morgan Kaufmann.

Kambhampati, S., and Srivastava, B. 1996. Universal classical planner: An algorithm for unifying state-space and plan-space planning. In Ghallab, M., and Milani, A., eds., *New Directions in AI Planning*. IOS Press (Amsterdam).

Kambhampati, S. 2000. Planning graph as a (dynamic) CSP: Exploiting EBL, DDB and other CSP search techniques in Graphplan. *Journal of Artificial Intelligence Research* 12:1–34.

Mitchell, T.; Keller, R. M.; and Kedar-Cabelli, S. T. 1986. Explanation-based generalization: A unifying view. *Machine Learning* 1(1):47–80.

Nguyen, X., and Kambhampati, S. Reviving partial order planning. In *Proceeding of the IJCAI-2001*.

Rodríguez-Moreno, M. D.; Oddi, A.; Borrajo, D.; Cesta, A.; and Meziat, D. 2004. IPSS: A hybrid reasoner for planning and scheduling. In de Mántaras, R. L., and Saitta, L., eds., *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI 2004)*, 1065–1066. Valencia (Spain): IOS Press.

Smith, D., and Weld, D. 1999. Temporal planning with mutual exclusion reasoning. In *IJCAI*, 326–337.

Sutton, R. S.; Precup, D.; and Singh, S. 1999. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence* 112:181–211.

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

Upal, M. A. 2003. Learning general graphplan memos through static domain analysis. In *In Proceedings of the Sixteenth Canadian Conference on Artificial Intelligent*.

Veloso, M. M., and Blythe, J. 1994. Linkability: Examining causal link commitments in partial-order planning. In *Proceedings of the Second International Conference on AI Planning Systems*, 170–175. Chicago, IL: AAAI Press, CA.

Veloso, M. 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 AI* 7:81–120.

Zimmerman, T., and Kambhampati, S. 2003. Learning-assisted automated planning: Looking back, taking stock, going forward. *AI Magazine* 24(2):73–96.