

Case-Based Recommendation of Node Ordering in Planning

Tomás de la Rosa and Angel García Olaya and Daniel Borrajo

Departamento de Informática, Universidad Carlos III de Madrid
Avda. de la Universidad, 30. Leganés (Madrid). Spain
trosa@inf.uc3m.es, agolaya@inf.uc3m.es, dborrajo@ia.uc3m.es

Abstract

Currently, among the fastest approaches to AI task planning we find many forward-chaining heuristic planners, as FF. Most of their good performance comes from the use of domain-independent heuristic functions, together with efficient search techniques. When analysing their performance, most of the time is spent precisely on computing the heuristic value of nodes. The goal of this paper is to present a way of reducing the number of calls to the heuristic function, and, therefore, the time spent on finding a solution. We use a case-based reasoning approach that automatically acquires domain-dependent typed sequences (cases) from some training problems. Then, the learned cases are used to recommend to each search node which of its successors to evaluate first. Experimental results in several competition domains show the advantages of the approach.

Introduction

Forward heuristic planning is nowadays one of the fastest approaches for AI task planning. Using this approach we find planners such as FF (Hoffmann & Nebel 2001) or SG-PLAN (Chen, Hsu, & Wah 2004). The key idea of these planners is the domain-independent heuristic function that guides their search algorithm towards the solution. To compute the heuristic function a relaxed version of the original problem is solved, ignoring the delete lists of actions in the domain. Regardless the way in which the relaxed solution to a problem is computed, the time spent to compute the heuristic value for all nodes takes most of the total planning time. Since the time computing heuristics is an open issue in heuristic planning, we propose a Case-based Reasoning (CBR) approach, as a learning technique that recommends node ordering for evaluation in order to reduce heuristic computation during the search. We assume that less node evaluations for computing heuristics will produce a total planning time improvement. This work is an extension of a previous work (DelaRosa, Borrajo, & García-Olaya 2006) in which we replayed typed sequences in a hill-climbing algorithm.

In the past, CBR seemed an interesting approach for not planning from scratch. ANALOGY (Velooso & Carbonell 1993) fully integrated CBR with generative planning based

on a derivational analogy process, in which lines of reasoning are transferred and adapted to a new problem. This idea was attractive to planning because it reasons over the planning trace rather than other CBR techniques that just search the solution using transformational adaptation such as PRIAR (Kambhampati & Hendler 1989). In forward heuristic search it is not possible to directly obtain a causal justification as it was stored in ANALOGY's cases. However, features of the planning trace can be still recorded. We suggest that sequences of visited states, stored as sequences of domain types, can be learned and later transferred as control knowledge to guide the search.

The heuristic of the relaxed plan introduced by the FF planner (Hoffmann & Nebel 2001) has proven to be accurate enough to guide an Enforced Hill-Climbing algorithm (EHC). Thus, our approach uses CBR for ordering the way in which nodes are evaluated. Since CBR will choose promising successors of nodes, EHC will find a node to follow the search with fewer heuristic computations than usual.

In the following sections we present the basic idea of EHC, and why node ordering for evaluation can improve its performance. Then we introduce the CBR cycle of our approach, explaining the typed sequences extraction and how they are stored as cases. Then, we describe the retrieval and the adaptation of the sequences that will guide the search in a replay process. We also present results in three of the International Planning Competition (IPC) domains. Finally, we discuss related work and present some conclusions.

Heuristic Search with EHC

The key ideas of FF-like planners is the heuristic function obtained from the relaxed plan graph and a systematic and efficient search algorithm such as EHC. After the relaxed plan graph expansion, a solution to the relaxed problem (a relaxed plan) is extracted, performing a backward chaining of actions through the graph. Then, the number of actions of the relaxed plan is taken as an estimate of how far the goal is from the current state. EHC performs local search in the search space of all reachable states. At each current state S , a breadth first search is performed until a state S' with a better heuristic value is found. Then, the search continues from S' until a node that achieves the problem goals is reached. One advantage of this algorithm is that not all state successors should be evaluated like in standard hill-climbing, since

when a state with a better heuristic value is found, the evaluation of the rest of successors is skipped. Then, the order in which nodes are evaluated directly influences the number of evaluations done, and therefore the total time of the search. Usually, node successors are evaluated randomly or in the fixed order in which they are generated, that is determined by the domain description and the node expansion implementation. As we will show in the following sections, this order can be guided with domain-dependent control knowledge learned through a CBR cycle.

Learning Typed Sequences

The idea of learning typed sequences comes from the observation of typical state transitions that each type of object has in a plan. PDDL, the standard language for describing planning tasks, permits the definition of domain types to group objects appearing in a problem. In a domain, objects of the same type frequently share features when transiting from one state to another. Since we can obtain these transitions from the solution path of a problem, a sequence of these transitions can be considered as a case for the object type. Viewing the whole process as a CBR cycle, typed sequences are extracted from previously solved problems and are stored in the case base. With a new problem to solve, a retrieve process selects the most similar sequences to the new problem. These sequences are adapted and used in the replay in order to guide the search. In the explanation of these CBR processes, we are going to use the Depots domain (part of the IPC collection), in which trucks transport crates around depots and distributors, and crates must be stacked onto pallets or on top of other crates at their destination.

Storing

A typed sequence (case) is formed by an ordered list of structures that we call sub-state relations. A sub-state relation of an object is a pair (typed sub-state, action to reach the state) in which the relevant information of the state for an object (object sub-state) and the applied action are abstracted in terms of the object type. Typed sequences are extracted from the solution to problems (plans, or sequences of instantiated actions that transforms the initial object sub-state into a sub-state in which a goal is met). To perform this abstraction, we extract first the object sub-state, which is the set of all facts in a state in which the object is present. Then, this object sub-state is translated to a typed sub-state of the type of the object. Thus, a subset of an initial state like `[(on crate0 pallet0) (at crate0 depot0) (clear crate0)]`, can be translated to a typed sub-state of "crate" as `[(on <x>_) (at <x>_) (clear <x>)]` where `<x>` represents any instance of a crate and the underscore sign represents another object in the literal not relevant to the typed sub-state. Formally, we define for an object o of type t , an object sub-state $U_o(o, S) = \{f \in S \mid o \text{ is an argument of } f\}$ and a typed sub-state $U_t(o, S) = \{l_t = T(l, o) \mid l \in U_o(o, S)\}$ where $T(l, o)$ is a function that transforms the fact l replacing the object o with the variable `<x>` and the other arguments with `_`. Thus, a sub-state relation $R(o, S) = [U_t(o, S), A_s]$

where A_s is the applied action to reach S . Since many actions in the sequence are not relevant to the object (the object is not a parameter of the action), and assuming that the object sub-state does not change either, a `no-op` is stored with the same sub-state. Then, from the solution path to a problem we can compute the typed sequence of each object, storing all sub-state relations in the order they appear.

Function Store-Case (O, S_0, P): new-cases

O : the set of objects of the current problem
 S_0 : the initial state
 P : the plan to solve the problem

For each o of type t in O
 $case = \emptyset$
 $S = S_0$
 For each a in $P = \{a_1, \dots, a_n\}$
 $case = case \cup substate_relation(o, S, a)$
 $S = Apply(S, a)$
 $new_cases_t = new_cases_t \cup case$
 return new_cases

Figure 1: Extraction algorithm.

Figure 1 shows the algorithm for extracting cases. For each object of the problem a typed sequence is generated. Figure 2 shows a typed sequence for `crate0` and the plan from which it was generated. The first step has no applied action and corresponds to the crate initial sub-state. Dots in the applied actions represent the action parameters not relevant to the sub-state relation. The two `no-op` in the sequence represent the two actions in which `crate0` is not relevant. The number stored with the `no-op` is the number of steps in which the current object sub-state has not changed. Then, cases are grouped by type of objects and a merge process determines if the new case to store is part of the case base or it is just a different case. Being n the number of steps of the new case and k the number of stored cases, the computational complexity of the storing phase is determined by the merge process which has $O(n^2k)$ complexity. In practice, this time is irrelevant compared to the time spent on solving the planning problems.

Retrieving

When the planner tries to solve a new problem, cases for each object appearing in the goals are retrieved. We compare the goal sub-state of an object with the last sub-state of all sequences in the case base of the corresponding type. Then, the initial sub-state of an object is compared with the first sub-state of the sequences that met the first criteria. If more than one sequence meet both criteria, the shortest sequence is retrieved. All retrieved sequences are kept in a replay table that will be used in the search process. The size of the replay table depends on how many sequences are retrieved, but there is at most one sequence for each different object appearing in the goals. Future work will use more sophisti-

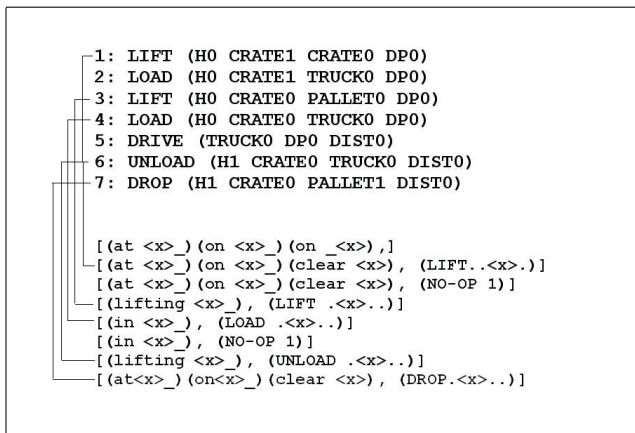


Figure 2: An example of a typed sequence relevant to a crate.

cated retrieval methods, but this retrieval scheme works reasonably well for now. Figure 3 shows the retrieval algorithm in which the replay-table is built with the retrieved cases.

<p>Function Retrieval (S_0, G, CB): <i>replay-table</i></p>
<p>S_0: the initial state of the current problem G: the conjunction of goals to be achieved CB: the Case Base</p>
<p>$replay-table = \emptyset$ For each o, o is an argument of $g \in G$ $candidates = \emptyset$ For each $\phi_t = \{R_0, \dots, R_n\}$ in CB, o is of type t If $U_t(o, G) \subseteq R_n$ then $candidates = candidates \cup \phi_t$ <i>sort_by_length</i> ($candidates$) For each C in $candidates$ If $R_0 \subseteq U_t(o, S_0)$ and o not in $replay-table$ then insert (o, C) in $replay-table$ return $replay-table$</p>

Figure 3: Retrieval algorithm.

As an example of the retrieval, suppose a set of goals $G = [(on\ cratel\ pallet1) (on\ crate2\ cratel)]$. We search in the case base for a sequence of type `pallet` for `pallet1`, and sequences of type `crate` for `cratel` and `crate2`. To retrieve a sequence for `cratel`, the selected case must have a sequence ending with the sub-state `(on <x>_) (on _<x>)` and must start with the corresponding initial sub-state. If no sequence matches, the object is not considered in the construction of the replay table.

Adapting

Typed sequences could represent many instantiated sequences in the new problem, since sub-state relations only take care of a particular referenced object and the other ob-

jects in the literals could receive as many bindings as objects of the ignored type. To address this issue, we perform an early extraction of a relaxed plan, and obtain the goal membership layers of the relaxed plan graph from the initial state. This goal membership is built during the relaxed plan extraction, and is represented with the sequence G_1, \dots, G_t , where G_i is the set of facts achieved by the applied actions of the relaxed plan at time step $i - 1$. With this goal membership we can transform typed sequences to real objects sequences, searching the right bindings through the goal layers. Suppose that the sequence in Figure 2 is retrieved for `cratel` appearing in the goal `(on cratel pallet1)`. In the seventh step, the literal `(lifting cratel _)` could match with any hoist of the problem, but the G_{t-1} layer has the right binding `(lifting cratel hoist1)` since it is a precondition to achieve the goal `(on cratel pallet1)` that is present in the G_t layer. Since the relaxed plan graph does not always represent the real order of achieving goals and subgoals, not all sequences are fully instantiated. When this happens, typed sub-state rather than instantiated object sub-states are compared during the replay. Figure 4 shows the instantiating algorithm for the adaptation phase. The algorithm searches for each step in the retrieved sequences if the typed sub-state corresponds to any object sub-state taken from each layer of the goal membership. If it does, the object sub-state is kept as the instantiation of the sequence step.

<p>Function Instantiate-Seq(RT, M): <i>instantiated-seq</i></p>
<p>RT: The <i>replay-table</i> from the retrieval phase M: The goal membership of the initial relaxed plan graph</p>
<p>For each G_i, G is the i layer of M For each $Q(o) = \{R_0, \dots, R_n\}$, Q is the sequence for the object o in RT For $k = n, \dots, 1$ If $U_t(o, G_i) \subseteq R_k$ then insert $U_o(o, G_i)$ in <i>instantiated-seq</i> return <i>instantiated-seq</i></p>

Figure 4: Adapting algorithm.

Replaying

The replay process of sequences is integrated in EHC, so it can reduce heuristic computation of nodes. The search goes as follows: at any state S (with a heuristic value $h(S)$ computed previously), the node successors are generated. Then, the process looks if the successor to evaluate is recommended by CBR. If it does, the successor is evaluated for computing its heuristic value $h(S')$. If $h(S')$ is strictly less than $h(S)$, this successor is selected, and the search continues from it, until a state achieving the problem goals is reached. If $h(S')$ is equal or greater than $h(S)$, a second attempt with the next successor is done, and so on, until a node with a better heuristic is found. If the CBR module

could not find a recommended node, all skipped successors are evaluated and the standard EHC is followed.

To analyse if a successor is a recommended node, the replay table holds the current pointer to all of the retrieved sequences, and if a state successor matches the next sub-state relation in the sequences, it is recommended for evaluation. This match is performed by converting the state successor to a sub-state relation of the same type of the compared sequence. If a recommended node is not a good choice, it will be discarded after its heuristic computation, and the next option will be suggested. Moreover, since retrieved sequences may share facts of their sub-states, if a sequence is advanced to the next sub-state relation, all sequences that share this sub-state are also advanced. Otherwise, if a node in the search is selected by the heuristic, the replay table is traversed to see if any sequence should be also advanced. If a sequence points to a no-op, it is not used again until it is advanced as many times as the number with the no-op. This guarantees that at least a number of actions, not relevant to the object, are applied before the sequence suggests a new sub-state. Figure 5 shows the algorithm used to know if a node is recommended for heuristic evaluation.

Function Recommended(RT, N): *recommended*

RT : The *replay-table* from the retrieval phase
 N : A successor of the current node

recommended = false

For each o , parameter of the action a applied in N

ϕ = object-sequence(o, RT)

R the sub-state relation pointed in ϕ

if R is instantiated then

If $(U_o(o, N), a) = R$ then
 $recommended = true$

else

If $(U_t(o, N), a) = R$ then
 $recommended = true$

If $recommended = true$

advance-sequences in RT

return *recommended*

Figure 5: Advise algorithm.

To illustrate the replay process with an example, we will use a simple problem that uses the sequence in Figure 2. We use only one sequence for clarity. In this problem there are one depot and two distributors. Each place has a hoist and a pallet. Initially *crate1* is on *crate0* at *depot0* Figure 6 shows the initial state, the problem goals, and first steps of the search tree. Static literals have been removed for clarity. In each node the heuristic value h is displayed, (“h:?” if not computed) as well as the applied action to reach the state. The sequence is used in reference to *crate0*. As we can see in the search tree, state *S3* is the right decision recommended by CBR and states *S1* and *S2* (with gray background), were not evaluated. They were skipped since stan-

dard EHC would evaluate them first. In the next two steps (*S4* and *S9*), no nodes were recommended, but *S12* was recommended whereas *S10* and *S11* were skipped.

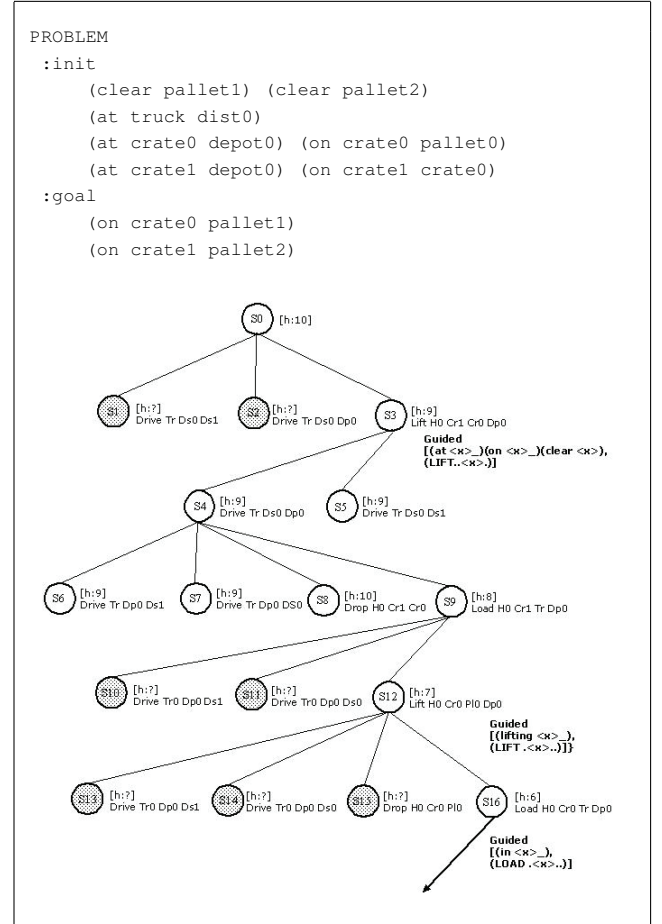


Figure 6: An example of the replay process.

Experiments

We have implemented this approach in SAYPHI, a learning architecture in which several techniques for control knowledge acquisition can be integrated with a common heuristic planner. The SAYPHI planner is an FF-like heuristic planner developed in LISP. It performs an EHC algorithm using the relaxed planning graph heuristic. For the experiments we have given the planner 10 random training problems from the Depots domain for extracting the typed sequences. These problems had up to 14 objects instances and up to 3 goals. In the case base 11 cases of type crate and 3 cases of type pallet were stored.

Then, we generated the test set with 100 random problems up to 30 objects instances and up to 10 goals. All these problems were generated with the random problem generator supplied by the IPC. Then, we solved each problem with EHC, and then supported by the CBR recommendation described in the previous sections, both with a time bound of 300 seconds.

The EHC algorithm solved 86% of the problems and with the CBR support 92%. Figure 7 shows the accumulated time used in problems solved by both techniques.

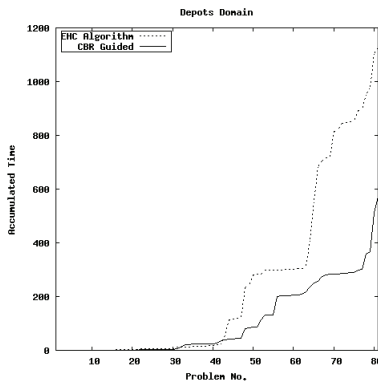


Figure 7: Accumulated time solving Depots problems.

We also tested our approach with the Logistics domain, in which packages need to be transported to given destinations. Destinations can be either post offices or airports. Airplanes fly between airports, and, trucks transport packages within the same city. For comparison, we used the IPC 2000 problem set. We populated the case base with the first three problems and obtained 4 cases of type package, 12 cases of type airport and 3 cases of type post-office. We trained the case base with the simplest problems of the competition (first ones in the competition set), because the training problems must be solvable by the planner in order to generate cases. We used the other 37 problems to test the EHC algorithm and the CBR supported search.

In this case both techniques solved all problems. Figure 8 shows the accumulated time solving the problems. We can observe a much better performance of CBR supported search. As problems increase their difficulty the branching factor of applicable actions also grows. Therefore, in these harder problems the number of evaluations avoided is more significant.

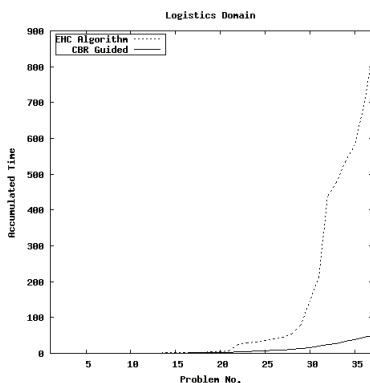


Figure 8: Accumulated time for the Logistics problems

Another domain we have used is the Satellite domain. It

involves planning a set of observation tasks between multiple satellites, which can point to different directions, supply power to one selected instrument, calibrate it to one target and take images of that target. For the experiment we used the IPC 2002 set. We populated the case base with 3 problems obtaining 6 cases for type direction and 5 cases for type mode. Then, we used the other 17 problems to test the EHC algorithm and the CBR supported search.

EHC solved 14 problems and the CBR supported search solved 16. All problems solved by EHC were also solved with CBR. Figure 9 shows the accumulated time for problems solved by both techniques. Again, there is a much better performance with the CBR supported search, mainly when problem difficulty increases.

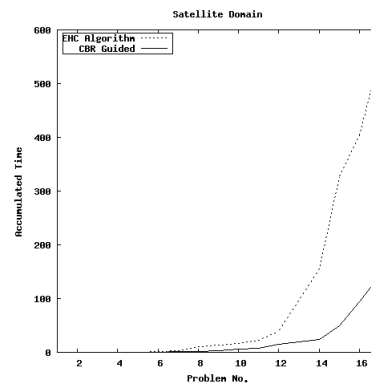


Figure 9: Accumulated time for the Satellite problems

Since the time spent to solve a problem depends also on the implementation and the development language, it is not obvious to compare planning time between a learning technique implemented within a planner, and a different planner. In this case FF outperforms our system in planning time (FF is implemented in C and incorporates more domain independent heuristics than SAYPHI). Instead, we decided to measure the number of evaluated nodes, assuming this is the key issue of our approach, since we could implement CBR in any other FF-like planner. We have run the test sets with FF. It has solved 99 problems of the Depots domain, and all problems of the Logistics and the Satellite domain. Table 1 shows the average of evaluated nodes in the tested domains for both FF and CBR-Sayphi.

Table 1: Average of evaluated nodes.

Heuristic	FF Planner	CBR-Sayphi
Depots	5671.7	1729.8
Logistics	200.4	84.1
Satellite	240.4	294.1

We can see that in the Depots domain and the Logistics domain the CBR supported search leads to a clear reduction of node evaluations. In the Logistics domain CBR-Sayphi evaluated fewer nodes than FF in all problems. Although

CBR outperforms the SAYPHI planner in the Satellite domain, it is not quite enough to win FF with respect to number of node evaluations. This might be because FF incorporates additional heuristics than our planner and because our current representation ignores some plan steps that should be in the sequences. These missing steps are part of the sequences of other types not present in the goal objects, which is one criteria for the retrieval phase. Another issue we have to address relevant to performance is adding some quality measure to used cases in the case base, since the stored sequences are extracted from non-optimal plans, and therefore we are re-using sub-optimal sequences. Thus, we could later remove from the base the low quality cases.

Related Work

The idea of extracting information from the domain types description is not new to the planning community. Our work is related to state invariants extracted from a domain analysis as in TIM (Fox & Long 1998). With a pre-processing tool, they obtain Finite State Machines that represent states in which a type of object can be and can move to. In our case this knowledge is obtained dynamically while the case base is being populated in the form of sequences.

Other approaches integrate CBR with planning. ANALOGY (Veloso & Carbonell 1993) and PRIAR (Kambhampati & Hendler 1989) that we have mentioned earlier, PARIS (Bergmann & Wilke 1996) which stores cases in different abstraction levels of a solved problem, CAPLAN-CBC (Muñoz-Avila, Paulokat, & Wess 1994) which performs plan-space search and replay and STEP-PINGSTONE (Ruby & Kibler 1989) which learns sub-goals sequences for a means-ends planner. The novel contribution of our approach is that the knowledge of the case base is abstracted in domain types and these cases represent sequences of state transitions. Moreover, cases do not represent directly plans of solved problems because a single plan can generate multiple typed sequences. Our approach also provides an additional way to address heuristic planning through heuristic ordering of evaluations. We can find also a similar approach in MACRO-FF (Botea *et al.* 2005) if macro-operators are seen as fixed cases that keep a sequence of applicable actions. The difference is that these macro-operators can not interleave actions, nor deal with node ordering for computing heuristic function, as we can do with typed sequences.

Conclusions and Future Work

In this work we have shown a new approach to forward heuristic planning, introducing a CBR component that helps an Enforced Hill-Climbing algorithm to decide the order in which nodes should be evaluated to compute the heuristic function. The CBR component uses typed sequences learned from previous solved problems to suggest promising next states in the search process. We have seen that the planner performance time is improved with this technique, since the planner does less heuristic computations. In many steps the suggested node is the only one evaluated and other nodes are ignored for evaluation. The key idea of this

work opens a variety of possibilities for helping heuristic planning. Any kind of learning technique that could predict somehow a next state in the search, would be applicable within the EHC algorithm.

Our future work will enrich the way sub-states relations are stored, in order to address the issue of domain representation mentioned before. We also want to include a quality measure of cases in the case base for a better retrieval process. A sequence that frequently predicts the right node to evaluate, can be marked as a good case, so it could be preferred when there is more than one possibility for retrieving.

Acknowledgments

This work has been partially supported by the Spanish MEC project TIN2005-08945-C06-05 and regional CAM-UC3M project UC3M-INF-05-016.

References

- Bergmann, R., and Wilke, W. 1996. Paris: Flexible plan adaptation by abstraction and refinement. In Voss, A., ed., *ECAI (1996) Workshop on Adaptation in Case-Based Reasoning*. John Wiley & Sons.
- 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 (JAIR)* 24:581–621.
- Chen, Y.; Hsu, C.-W.; and Wah, B. 2004. Sgplan: Subgoal partitioning and resolution in planning. In *Proceedings of the 4th International Planning Competition (IPC4), in Conference ICAPS'04*, 30–33.
- DelaRosa, T.; Borrajo, D.; and García-Olaya, A. 2006. Replaying type sequences in forward heuristic planning. In Ruml, W., and Hutter, F., eds., *Technical Report of the AAAI'06 Workshop on Learning for Search*. Boston, MA (USA): AAAI Press.
- Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research* 9:317–371.
- Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.
- Kambhampati, S., and Hendler, J. 1989. Flexible reuse of plans via annotation and verification. In *Proceedings of 5th International Conference on Artificial Intelligence for Applications*, 37–43.
- Muñoz-Avila, H.; Paulokat, J.; and Wess, S. 1994. Controlling nonlinear hierarchical planning by case replay. In *in working papers of the Second European Workshop on Case-based Reasoning*, 195–203.
- Ruby, D., and Kibler, D. 1989. Learning subgoal sequences in planning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 609–614. San Mateo, CA: Morgan Kaufmann.
- Veloso, M. M., and Carbonell, J. G. 1993. Derivational analogy in PRODIGY: Automating case acquisition, storage, and utilization. *Machine Learning* 10(3):249–278.