Adaptation of Hierarchical Task Network Plans

Ian Warfield, Chad Hogg, Stephen Lee-Urban, Héctor Muñoz-Avila

Department of Computer Science and Engineering, Lehigh University
{ipw2, cmh204, sml3, hem4}@lehigh.edu

Abstract

This paper presents RepairSHOP a system capable of performing plan adaptation and plan repair. RepairSHOP is built on top of the HTN planner SHOP. RepairSHOP has three properties. The first property is its design modularity, which makes it is straightforward to apply the same process discussed in this paper to build plan adaptation capabilities in other HTN planners. Second, RepairSHOP can perform plan repair. Third, RepairSHOP takes into account failed traces during plan adaptation/repair. As a result, it can result in improvements in running time performance. We performed experiments demonstrating performance gains of plan adaptation over plan generation from the scratch, measured in CPU time for problem solving.[1]

Introduction

Plan adaptation is a problem-solving technique in which existing plans are modified to solve new problems. As such, it is a central part of case-based planning research (Cox and Muñoz-Avila, 2006). Plan adaptation has played a central part in research on case-based planning applications, such as manufacturing (Muñoz-Avila & Weberskirch, 1996) and military planning (Mitchell 1997).

Hierarchical task network (HTN) planning is an important, frequently studied research topic. Researchers have reported work on its formalisms and applications (Nau *et al*., 2005). In HTN planning, high-level tasks are decomposed into simpler tasks until a sequence of primitive actions solving the high-level tasks is generated. HTN planning is a natural representation for many real-world domains, including military planning (Mitchell, 1997), to encode strategies in computer games (Smith *et al*., 1998), and manufacturing processes (Nau *et al*., 2005).

Although work has been published on adaptation of hierarchical plans (Paolucci *et al*, 1999), the known adaptation approaches are tightly coupled to the particular representation or application for which they were developed. Therefore, it is difficult to use these approaches for general HTNs. Furthermore, these approaches use only the traces that led to the solution. Failed traces that might have been explored during problem solving are not taken into account during plan adaptation. Therefore, the plan adaptation process may expend resources exploring parts of the search space that have already been shown to lead to failure during the initial problem solving process.

In this paper we present RepairSHOP, a plan adaptation algorithm build on top of the HTN planner SHOP, which implements a variant of HTN planning called Ordered Task Decomposition. In this variant tasks are totally ordered and conditions are evaluated relative to the current state of the world, which is updated during planning. RepairSHOP has three novel characteristics: first, it is built on a modular system called the goal graph system. As a result, it is easy to apply the same process discussed in this paper to build plan adaptation capabilities in other variants of HTN planning. Second, RepairSHOP can also perform plan repair; That is, RepairSHOP is able to modify an existing plan on-the-fly when the situation in the world changes. Third, RepairSHOP records failed traces during the HTN planning process. This information can be exploited to improve running time performance during plan repair/adaptation. We performed experiments demonstrating performance gains of plan adaptation with RepairSHOP over plan generation from the scratch with SHOP, measured in CPU time for problem solving.

Preliminaries

To perform hierarchical decomposition, we follow the principles of Hierarchical Task Network (HTN) planning as in the SHOP system (Nau *et al.*, 2005). HTN planning achieves complex tasks by decomposing them into simpler subtasks. Planning continues by decomposing the simpler tasks recursively until tasks representing concrete actions are generated. These actions form a plan achieving the high-level tasks.

The main knowledge artifacts that indicate how to decompose tasks are called methods. A ***method***, M, is a 3-tuple: (h,Q,ST), such that: h, called the ***head*** of M, is the task being decomposed; Q, called the ***conditions***, are the preconditions required for using the method; and ST is the list of ***subtasks*** achieving h. The tasks in ST are totally ordered according to the order they are listed.

To achieve a task that can be decomposed (called a compound task), an HTN planner searches for applicable methods. A method M is ***applicable*** to a compound task t, relative to a ***state*** S (a set of ground atoms), iff match(h,t) (i.e., h and t have the same predicate and arity, and a consistent set of bindings Θ exists, which maps variables to constants so that all terms in h match their corresponding ground terms in t) and Q are ***satisfied*** by S (i.e., there exists a consistent extension Θ' of Θ such that $\forall q \in Q \ \{q\Theta' \in S\}$ and $\forall \neg q \in Q \ \{q\Theta' \notin S\}$). To achieve a task that represents an action (called a primitive task), HTN planners use

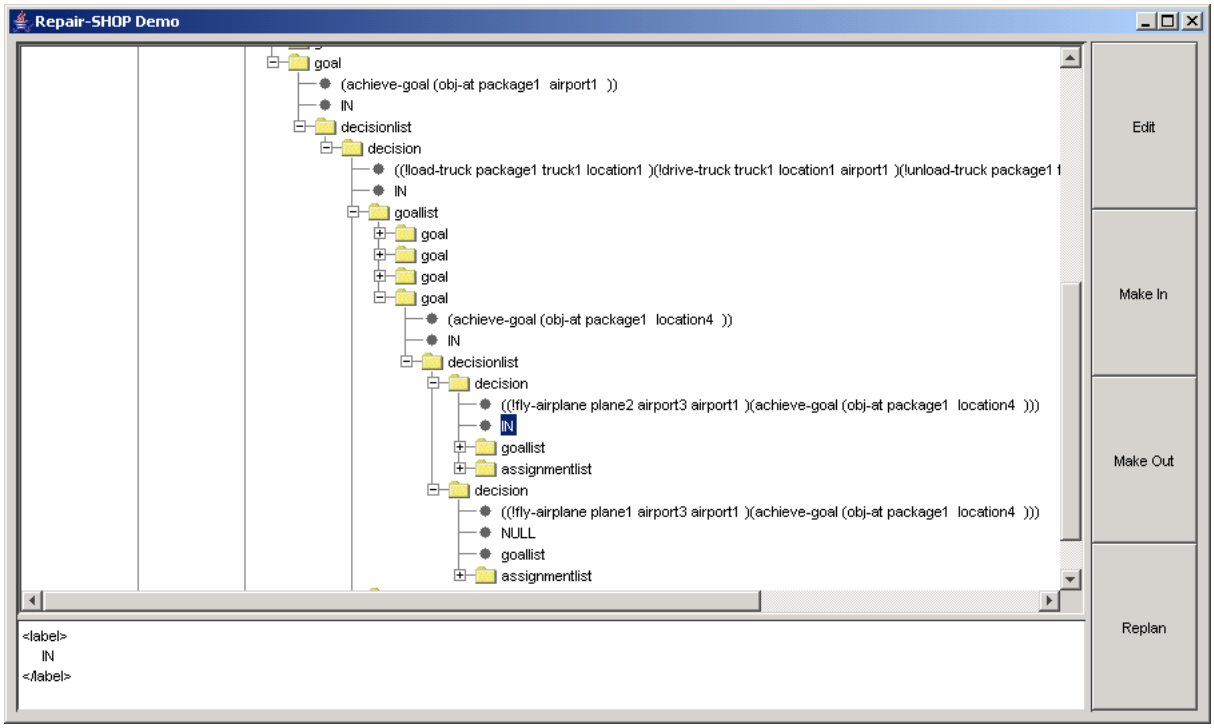

Figure 1: Snapshot of the goal graph

operators. An ***operator*** *O* is of the form *(h,al,dl)*, such that: *h* (the operator's ***head***) is a primitive task, and *al* and *dl* are the so-called ***add-list*** and ***delete-list***. The two lists define how the operator will transform the current state *S* when applied: every atom in the add-list is added to *S* and every atom in the delete-list is removed from *S*. An operator *O* is ***applicable*** to a primitive task *t*, relative to a state *S*, iff match(*h,t*).

A ***planning problem*** is a triple (*T,S,D*), where *T* is a set of tasks, *S* is a state, and *D* is a ***planning domain theory*** -- a collection of methods and operators. Since D is fixed, we refer to problems as pairs (T,S) hereon. A ***plan*** is a totally ordered collection of primitive tasks. Informally, given a planning problem *(T,S)*, the collection of primitive tasks that decompose all compound tasks in *T*, relative to *S* and *D*, is a ***correct*** plan (Nau *et al.*, 2005). Besides generating a plan for a planning problem, we are also interested in the hierarchical task network (HTN) that led to the plan. The plan can be obtained by performing a pre-order traversal of the resulting HTN collecting all primitive tasks along the way. Formally, an ***HTN*** is defined as follows:

- An expression of the form t^h is an HTN, where t^h is a task, which is represented as a logical atom.
- An expression of the form $(t^h,(T_1,\ldots,T_m))$ is an HTN, where t^h is a task and $T_1,\ldots,T_m$ are HTNs. The task network indicates that t^h is decomposed into $T_1,\ldots,T_m$. Tasks are achieved in the order they are listed.

A ***case*** (T,S,GG) is defined as a collection of tasks T, a state S, and a graph structure GG, called the goal graph. The goal graph GG represents the HTN generated when solving (T,S), augmented by other relations. We will explain the goal graph in detail later.

Overview of Adaptation Procedure

The adaptation procedure, ***RepairSHOP***, receives as input a case (T,S,GG) and a new problem (T',S'), and reuses the case's GG to generate an HTN for the new problem. GG captures dependencies between elements in the HTN solving (T,S). These dependencies are evaluated relative to (T',S') resulting in a partial HTN that is enhanced by using standard HTN planning techniques.

Our motivation for integrating the goal graph with SHOP was to enable plan adaptation. SHOP has no intrinsic method of determining the effect of a change on its conditions aside from reformulating the entire plan. We proposed incorporating a structure, the goal graph, to maintain the dependencies among the SHOP task nodes that allows SHOP to monitor changes in a task's conditions. This structure propagates changes in conditions to the appropriate task nodes, allowing SHOP to replan the affected sections.

The Goal Graph

The goal graph was conceived to augment SHOP with replanning capabilities. The goal graph takes the form of a directed dependency graph with a one-to-one mapping between each goal in the graph and each task in SHOP.

There are a number of different possible representations of a hierarchical goal structure that maintain a one-to-one correspondence between goals in the goal graph and tasks in SHOP. Based on our previous work (e.g., (Muñoz-Avila

& Weberskirch, 1996)), we chose to implement the REDUX architecture (Petrie, 1991). Redux combines the theory of Justification-based Truth Maintenance System (JTMS) and Constrained Decision Revision (CDR). In a Truth Maintenance System (TMS), assertions (called nodes) are connected via a tree-like network of dependencies. The combination of JTMS and CDR provides the ability to perform dependency-directed backtracking, which is adopted in GG to propagate changes.

In JTMS, each assertion is associated with a justification (Doyle, 1979). A justification consists of two parts: an IN-list and an OUT-list. Both the IN-list and OUT-list of a justification are sets of assertions. The assertions in the IN-list are connected to the justification by "+" links, while those in OUT-list are linked by "-" links. The validation of an assertion is supported by the justification that it is associated with, i.e., an assertion is believed when it has a valid justification. A justification is valid if every assertion in its IN-list is labeled "IN" and every assertion in its OUT-list is labeled "OUT". If the IN- and OUT-lists of a justification are empty, it is called a *premise justification*, which is always valid. A believable assertion in JTMS is labeled "IN", and an assertion that cannot be believed is labeled "OUT". To label each assertion, two criteria about the dependency network structure need to be met: *consistency* and *well-foundness*. Consistency means that every node labeled IN is supported by at least one valid justification and all other nodes are labeled OUT. Well-foundness means that if the support for an assertion only depends on an unbroken chain of positive links ("+" links) linking back to itself, then the assertion must be labeled OUT. A node is labeled IN when it has a valid justification, i.e., the assertions in the IN-list of the justification are all labeled IN, and the assertions in the OUT-list of the justification are all labeled OUT. A node is labeled OUT if either it has an invalid justification (which means that either some assertions in the IN-list are labeled OUT, or some in the OUT-list are labeled IN, or both situations occur), or it has no associated justification that supports it.

The goal graph represents relations between goals, operators and decisions. A goal is decomposed into subgoals by applying an operator. The applied operator is called a decision. The assignments represent conditions for applying the operator.

Figure 1 shows a snapshot of our implementation of the goal graph. A goal may have several decisions, one for each possible operator that can achieve the goal. In the goal graph, ***decisions*** decompose ***goals*** into the ***subgoals***. A decision contains a ***goal list***, storing all the subgoals of the goal. ***Assignments*** needed for applying the decision to the goal are collected in an ***assignment list***, which is also contained in the decision.

A JTMS mechanism is built on the goal graph. A decision is valid if all the assignments in its assignment list are valid, and all the subgoals in its goal list are valid. A valid decision will be labeled "IN". For a goal, all the decisions in its decision list labeled "IN" are applicable, which means the goal can be decomposed by those valid decisions. If for some reason the validity of some assignments of a valid decision change, then that decision may become invalid as well.

Repair-SHOP: Adapting HTNs

We mapped the elements of HTN plans into goals graphs (see Table 1). This allows us to construct the goal graph automatically during HTN planning. Because tasks are decomposed into subtasks, tasks were mapped into goals. A task may be associated with some resources. These resources are mapped as assignments in GG. Ordering relationships between tasks are also mapped into assignments.

The mapping of HTNs into goal graphs results in the following dependencies represented in the goal graph:

- Subtasks depend on their parent tasks
- Methods/operators depend on the tasks they decompose/achieve.
- Subtasks depend on the method introducing them
- Decisions depend on the task they accomplish
- Preconditions and task orderings depend on the decision that added them

Table 1: Map of HTNs into GG

HTNs	Goal Graph
Task	goal
Subtask	subgoal
preconditions	assignment
Subtasks orderings	assignment
Method	operator
Operator	operator

These dependencies determine the next elements that are accessed in the JTMS-propagation process. The advantage of using the goal graph alongside SHOP is that goal graphs preserve information about the state of the plan for each task and subtask that SHOP attempts to solve. Leaves in the goal graph correspond to primitive tasks in the HTN. Internal nodes in the goal graph correspond to compound tasks in SHOP, culminating in the original compound task at the root of the goal graph. Repair-SHOP can return to an arbitrary planning state by navigating in the goal graph. Repair-SHOP adds overhead to the HTN planning process as shown in the experimental evaluation.

Because the goal graph is constructed during planning, Repair-SHOP constructs justifications for branches where a failure might occur (and thus require backtracking). These ***justifications*** are conjunctions of assignments that couldn't be fulfilled and caused the decision to fail. When backtracking occurs these justifications are propagated until a goal is reached where an alternative decision can be explored. At this point the justifications are added to the failed decision and are checked during replanning.

To construct the goal graph on-the-fly, SHOP was expanded as follows:

- As each task is obtained from the beginning of the task list *T'*, a goal is created to encapsulate that task.

The goal's identifier name is assigned as the string representation of that task. The goal is then added to the parent goal list.

- For primitive tasks, the planner is recursively called for the remainder of the task list. If the planner returns successfully, the new goals (which may contain goal trees of their own) are added to the parent goal list.
- For compound tasks, each possible reduction is considered in turn. Each reduction is encapsulated in a new decision, with the decision's identifier name containing the string representation of the task being reduced. Each reduction's conditions are stored in that decision's assignment list. If a reduction succeeds, its plan is added to the decision's goal list. If a reduction does not succeed, its decision is marked OUT.
- If a compound task succeeds in choosing a decision, and there are additional decisions left untried, these are all marked NULL. This signals to Repair-SHOP that the decisions were not evaluated but that they might be revisited in the future.

Repair-SHOP's operation is straightforward. When Repair-SHOP monitors a change in conditions, it propagates the result to the highest affected goal. It then checks to see if an alternate decision (one previously marked NULL) is available from that goal. If no alternate decision is available, GoalGraph navigates up the tree, returning the first available alternate decision from the nearest goal node. If an alternate decision is eventually found, replanning is potentially possible. The pseudo-code of this process, called adjustGoalGraph, is shown below.

Repair-SHOP then obtains the stored SHOP state from the chosen alternate decision. It then restores the saved SHOP state and restarts SHOP's planning algorithm. The effect is the same as if SHOP had been planning along this path from the beginning. If the plan is not successful, Repair-SHOP searches for a new alternate decision. If it is successful, the returned plan is saved as a new subplan. This new subplan is then spliced into the original plan beginning with the first affected goal node.

adjustGoalGraph(G, S, GG)
// **Input**: G is a goal; S is the current state; GG is the goal graph;
// **Output**: GG is updated with a valid alternative for G, if any exists

0. O ← currentOperator(G,GG)
1. **if** invalidOperator(G,D,S) **then** label(O, OUT)
2. **else** return GG
3. **if** a valid operator, O_G of G with state S exists **then**
4. select (G, O_G, GG)
5. label(O_G, IN)
6. **return** GG
7. **else**
8. label(G, OUT)
9. P ← parent(G, GG)
10. **return** adjustGoalGraph (P, S, GG)

The adjustGoalGraph algorithm operates as follows. First, the operator achieving the goal G is obtained (Step 0). When the operator is invalid (e.g., if the goal is a compound task, and its method is not applicable in S), the operator O for G will be labeled as invalid (Step 1).

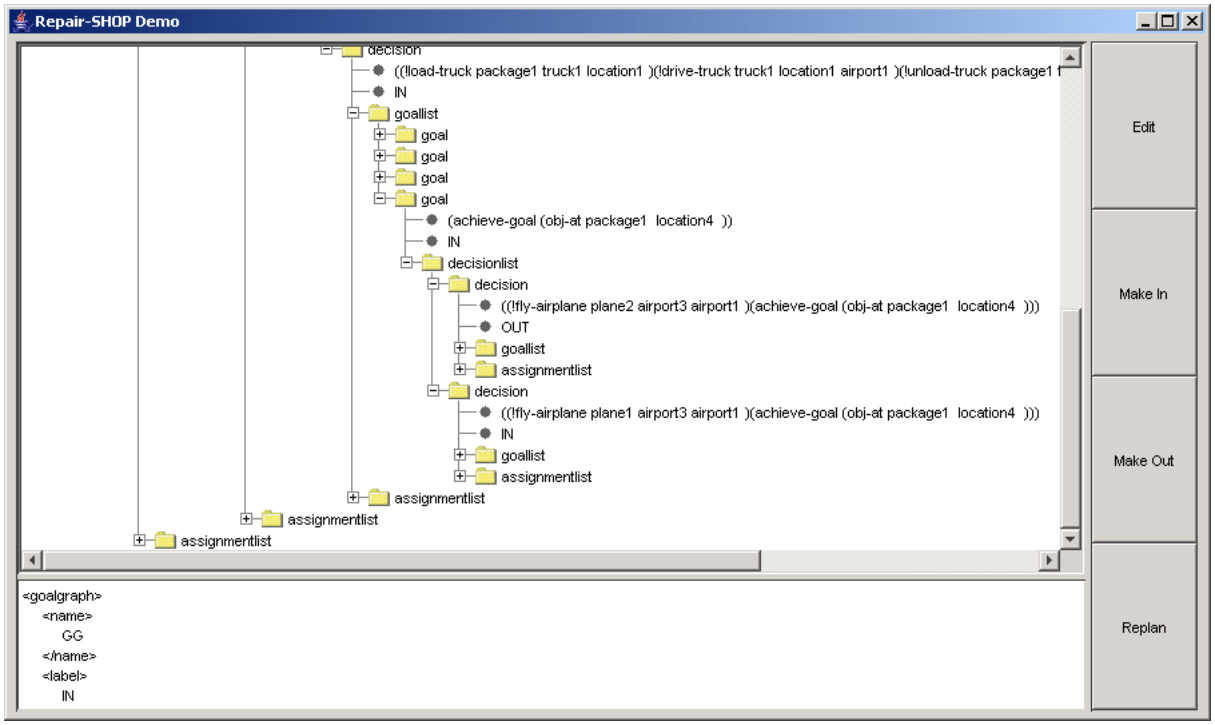

Figure 2: Goal graph after repair

Otherwise the algorithm terminates (Step 2). If a valid alternative operator O_G exists, O_G is labeled as valid, and GG is updated to record the modifications (Steps 3-6). If there are no alternative goals available, which means G cannot be repaired (Step 7), then a recursive call is made with the parent goal of the current goal to propagate the effect of T's loss of validity (Steps 9-10).

In summary, adjustGoalGraph propagates the result of the change towards the root via the goal graph. We close this section by making two notes about the Repair-SHOP procedure:

1. Repair-SHOP must notify all nodes of any change in assignment synchronously. This is necessary because certain plans may require the same condition in more than one place, and all instances of the same condition must remain consistent. The synchronous updates are done by looping through the list of identical assignment nodes stored in a specialized data structure.
2. Even though there are multiple valid decisions, they may be marked NULL. This reflects the fact that typically the planner stops considering alternatives once one is found to be successful. During adaptation, adjustGoalGraph must therefore examine the validity of alternative decisions should the one currently selected fail.

Example

We now discuss a sample run of the Repair-SHOP system incorporating both the goal graph and SHOP. The plan under consideration is a shipping problem where the computer must determine a way to move a package from City A to City B using an HTN version of the logistics transportation domain (Veloso, 1994). A typical problem in this domain is to place the objects at different locations, starting from a configuration of objects, locations, and transportation means. There are different sorts of locations and means of transportation. The means of transportation have certain operational restrictions. For example, a truck can only be moved between two places located within the same city.

The goal graph shown in Figure 1 shows two alternative decisions for the task *(achieve-goal (obj-at package1 location4))*. These decisions are decomposing the task into: (1) *(!fly-airplane plane2 airport3) (achieve-goal (obj-at package1 location4))*, and (2) *(!fly-airplane plane1 airport3) (achieve-goal (obj-at package1 location4))*. *location4* is in the same city as *airport3* (not shown in Figure 1). Therefore, these two alternatives relocate the plane containing *package1* in *airport3*, from where *package1* can be relocated to *location4*. The first decision is selected in the goal graph (labeled IN).

Now suppose that this goal graph is stored in a case and that a new problem is given whose task is the same as the one in the case and whose initial state is almost identical – with the one exception that *plane2* is not available. Repair-SHOP will reconstruct the goal graph relative to the new problem, thereby sparing the need for SHOP to search for an HTN. When it reaches the first decision of the task *(achieve-goal (obj-at package1 location4))*, it notes that this decision is not valid. As a result, *adjustGoalGraph* is called, with the goal graph goal corresponding to this task. *adjustGoalGraph* finds that the second decision is valid and labels it accordingly. The resulting goal graph is shown in Figure 2.

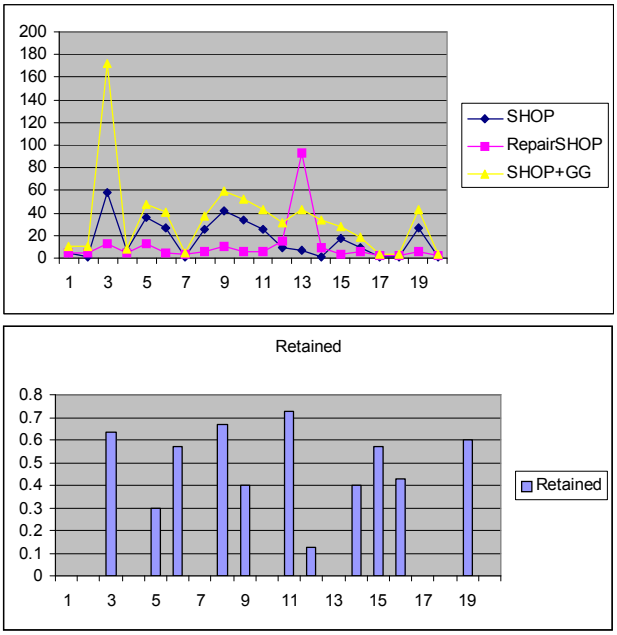

Figure 3: Empirical results

Empirical Evaluation

We conducted experiments to measure (1) the overhead added to the first-principles planning process by constructing the goal graph, (2) any gains in problem-solving performance of adapting the plans, and (3) the percentage of steps in original HTN retained. The domain was the logistics transportation domain.

In order to evaluate Repair-SHOP's performance as compared to SHOP, we ran a series of experiments to test its execution time and ability to handle plan adaptation on different problem spaces. We obtained a series of 20 problems P_i using a custom-designed random problem generator and solved them using SHOP, while automatically constructing the goal graph. A collection of 20 new problems P_i' were generated by taking each problem P_i, randomly choosing a condition, and removing it from the initial state. We measured the time required by SHOP to solve each P_i', the time required by SHOP to solve P' while generating the goal graph, the time required to replan by Repair-SHOP to solve each P_i' by adapting the goal graph generated for P_i, and the number of steps retained. Since time measurements change between runs, we report the average over 5 runs. The tests were performed on a 2.5 GHz Pentium 4 computer running Windows XP and executing code within JBuilder©. Figure

3 summarizes the results. The top figure compares the running times and the bottom figure indicates the percentage of steps retained after adaptation.

Several facts can be observed from this figure. First, we confirmed that constructing the goal graph causes an overhead to the plan generation process. As a result SHOP+GG generally takes longer than SHOP to generate plans. This was not unexpected, given the additional overhead required. Second, we observed that in most problems the time to replan by Repair-SHOP is smaller than SHOP. The only exceptions were problems in which no steps were retained. Noteworthy is Problem # 13 in which no steps were retained. However, on average the improvement RepairSHOP took 63.7% of the time it took SHOP to solve the same problems (10.7 seconds versus 16.9 seconds). An average of 27.2% of the steps from the original plan where retained in the final solution. Although admittedly, the problems in P_i where very similar to the ones in P_i, the results however show the potential for speed-up in problem solving by adapting HTNs.

Related Work

This work is heavily inspired by work on plan adaptation that takes into account the path leading to a solution and failure traces (Veloso, 1994). During problem-solving, a first-principles planner may explore branches of the search space that resulted in failures. These failures together with the path that led to a solution are stored in the case. Such information was first implemented for a state-space planner (Veloso, 1994), and later implemented for partial-order planners (Muñoz-Avila & Weberskirch, 1996). Plan adaptation of hierarchical plans has been proposed before (Kambhampati & Hendler, 1992; Muñoz-Avila et al., 2001). Whereas these works use the path that led to the solution, the distinguished characteristic of RepairSHOP is that it takes into account failure traces as well.

RepairSHOP can be used to perform plan repair. In plan repair an existing plan must be modified because of changes in the world conditions (van der Krogt *et al*., 2005). In fact, our experiments illustrate plan repair capabilities as well as plan adaptation because the tasks are the same and what changes is the initial state. The RETSINA system (Paolucci *et al*, 1999) repairs hierarchical plans in the context of a multi-agent system. However, it does not takes into account failed traces.

We have not discussed the topic of similarity and in particular its impact on adaptation for the sake of focus. The strong relation between the retrieval and adaptation effort is well known (Veloso, 1994).

Final Remarks

Repair-SHOP, the result of the integration of the goal graph system and SHOP, is a powerful tool that allows plan adaptation and plan repair, facilitating using SHOP in dynamic environments. Although creating the goal graph requires a certain amount of overhead, the costs are greatly outweighed by the benefits of the plan adaptation and replanning capabilities. While Repair-SHOP is now a complete system, there are still areas in which it could be upgraded or improved. Currently, only replanning from invalidated assignments has been implemented. Therefore, the system can only consider situations when conditions in the case are missing. Clearly, it would be desirable to consider situations where new conditions are added (e.g., additional resources are made available).

One possible application of our system could be to allow computer-controlled units to react to changes in the environment in a sensible way. Hierarchical planning in this area has already been explored (Hoang *et al*., 2005). Many games include decision trees for research, manufacturing, or combat where one objective must be completed for another to be possible.

References

Cox, M. T., Muñoz-Avila, H., & Bergmann, R. Case-based planning. *Knowledge Engineering Review. 20*(3): 283-287. 2006.

Doyle, J. Truth Maintenance System. *Artificial Intelligence,* 12, 1979.

Hoang, H., Lee-Urban, S., and Muñoz-Avila, H. Hierarchical Plan Representations for Encoding Strategic Game AI. *Proceedings of AIIDE-05*. AAAI Press.

Kambhampati, S. & Hendler, J. A Validation Structure-Based Theory of Plan Modification and Reuse. *Artificial Intelligence Journal.* Vol 55. 1992.

Mitchell, S.W. A hybrid architecture for real-time mixed-initiative planning and control. *Proceedings of the IAAI-97*. AAAI Press, 1997.

Muñoz-Avila, H., Aha, D.W., Nau D. S., Breslow, L.A., Weber, R., & Yamal, F. SiN: Integrating Case-based Reasoning with Task Decomposition. In *Proceedings of IJCAI-2001*. AAAI Press, 2001.

Muñoz-Avila & Weberskirch, Planning for manufacturing workpieces by storing, indexing and replaying planning decisions. In *Proceedings of AIPS-96*. AAAI-Press, 1996.

Nau, D., Au, T.-C., Ilghami, O., Kuter, U., Muñoz-Avila, H., Murdock, J. W., Wu, D., and Yaman, F. Applications of SHOP and SHOP2. *IEEE Intelligent Systems* 20(2). 2005.

Paolucci, M.; Kalp, D.; Pannu, A. S.; Shehory, O.; and Sycara, K. A planning component for RETSINA agents. In Intelligent Agents VI. Springer. 1999.

Petrie, C. *Planning and Replanning with Reason Maintenance*. PhD thesis, University of Texas at Austin, Computer Science Dept. 1991.

van der Krogt, R.P.J. and de Weerdt, M.M.. Plan Repair as an Extension of Planning. In *Proceedings of ICAPS-05*. 2005.

Veloso, M. *Planning and learning by analogical reasoning*. Springer-Verlag, 1994.