The Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS), Breckenridge, CO, April 15-17, 2000

New Advances in GraphHTN: Identifying Independent Subproblems in Large HTN Domains

Amnon Lotem and Dana S. Nau

Department of Computer Science and Institute for System Research
University of Maryland
College Park, MD 20742
{lotem, nau}@cs.umd.edu

Abstract

We describe in this paper a new method for extracting knowledge on Hierarchical Task-Network (HTN) planning problems for speeding up the search. This knowledge is gathered by propagating properties through an AND/OR tree that represents disjunctively all possible decompositions of an HTN planning problem. We show how to use this knowledge during the search process of our GraphHTN planner, to split the current refined planning problem into independent subproblems.

We also present new experimental results comparing GraphHTN with ordinary HTN decomposition (as implemented in the UMCP planner). The comparison is performed on a set of problems from the UM Translog domain - a large HTN transportation domain that is considerably more complicated than the well known "logistics" domain.

Finally, so that we could compare GraphHTN with action-based planners such as IPP and Blackbox, we translated the UM Translog domain into a STRIPS-style representation. We found that GraphHTN performed considerably better on UM Translog than IPP and Blackbox.

Introduction

In previous work (Lotem, Nau, and Hendler 1999) we developed the GraphHTN algorithm, a hybrid planning algorithm that does Hierarchical Task-Network (HTN) planning (Sacerdoti 75, Currie & Tate 1985) using a combination of HTN-style problem reduction and Graphplan-style planning-graph generation (Blum & Furst 1997).

In this paper we extend that work and present the following new results:

1. We show a new way to extract properties of HTN planning problems that speed up the search. The new properties are propagated through GraphHTN's *planning tree* – an AND/OR tree that is used to represent disjunctively all possible HTN decompositions of a planning problem. The search algorithm exploits these properties for splitting the current refined planning problem into independent subproblems. We show that by solving the independent subproblems separately and then merging their solutions the planner searches through a significant smaller search space and achieves significant performance improvement over solving the planning problem as a whole.
2. We have compared GraphHTN with UMCP (Erol *et al*. 1994a, Tsuneto *et al*. 1998), a traditional HTN planner, on a set of problems from a large HTN domain: UM Translog (Andrews *et al*. 1995). The UM Translog domain is an order of magnitude larger in size and number of features than, for example, the well known *logistics* domain. This comparison is more meaningful than previous results (Lotem, Nau, and Hendler 1999), since the original motivation for HTN planning was to handle large domains.
3. We have compared GraphHTN with Blackbox and IPP on the UM Translog domain. To perform this comparison, we needed to translate UM Translog into a STRIPS-style action representation. To do so, we developed an algorithm for translating HTN planning domains into STRIPS-style domains, that works correctly if the HTN domain contains no recursive methods. We describe our experience at using this translation. We found that the algorithm produced a domain that was too big for both IPP (Koehler *et al*. 1997) and Blackbox (Kautz & Selman 1998) to handle successfully.

   To overcome that problem we also tried encoding the UM Translog domain into a STRIPS-style domain by hand, directly from the initial domain description. The resulting domain was still too large for Blackbox on our computer. It was compact enough that IPP could handle it, but IPP still ran considerably less efficiently on this problem than GraphHTN.

This paper starts with a short background section on HTN and the GraphHTN planner. It describes how the new properties are propagated through the *planning tree* and exploited to identify independent subproblems. Then it describes the translation of UM Translog into a STRIPS-style domain. All the performance tests are presented in the Tests and Results section. We conclude with conclusions and suggestions for future work.

Background

HTN Planning

HTN planning is an AI planning methodology that creates plans by *task decomposition*. The planning problem is specified by an *initial task network,* which is a collection of tasks that need to be performed under specified constraints.

The planning process decomposes tasks in the *initial task network* into smaller and smaller subtasks until the task network contains only *primitive tasks* (*operator*s). The decomposition of a task into subtasks is performed using a *method* from the *domain description*. The *method* specifies how to decompose the task into a set of subtasks. Each *method* is associated with various constraints that limit the applicability of the method to certain conditions and define the relations between the subtasks of the method.

HTN planners traditionally use classical refinement search for finding a plan. However, new approaches for solving HTN problems include the encoding of HTN problems into a propositional satisfiability problems (Mali & Kambhampati 1998; Mali 99) and GraphHTN (Lotem, Nau, and Hendler 1999).

The GraphHTN Planner

GraphHTN is a hybrid planning algorithm that does HTN planning using a combination of HTN-style problem reduction and Graphplan-style planning-graph generation. The algorithm builds two explicit data structures that represent disjunctively all the possible solutions: the *planning graph* (Blum & Furst 1997) and the *planning tree* (Lotem, Nau, and Hendler 1999).

The *planning tree* is an AND/OR tree that expresses all the possible ways to decompose the initial task network using the HTN methods. The planning graph in the context of GraphHTN states which tasks of the *planning tree* are applicable at each time step, and which facts might be true after each time step. Figure 1 presents an example for a *planning tree* of a simplified planning problem in the UM Translog domain. The initial task network is represented by the two *transport* tasks at the top of the tree (transporting p1 from L1 to L4, and transporting p2 from L2 to L4). Transport(p1, L1, L4) can be decomposed either to the Carry-direct subtask which uses the t1 truck, or to the one which uses t2, and so forth.

The algorithm works as follows. Like the Graphplan algorithm, it incrementally increases the length of the plan being searched. In each iteration, it extends the planning

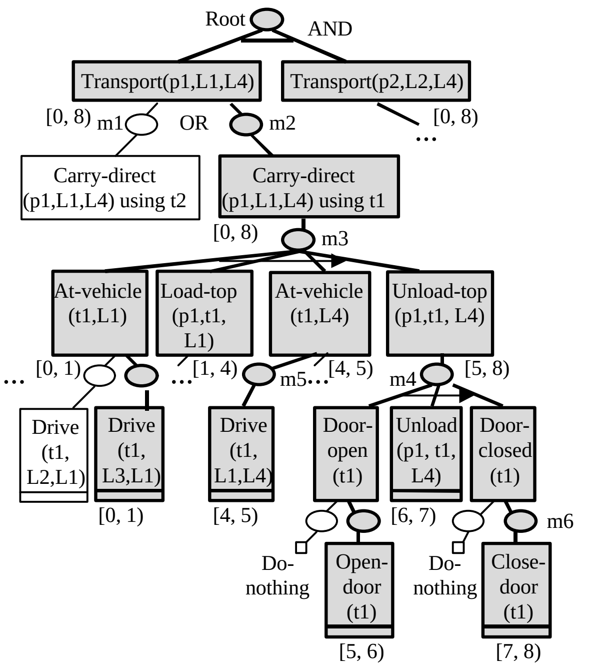


Figure 1: A planning tree for a simplified version of UM Translog. The ovals represent methods, and the rectangles tasks

tree by performing only task decompositions that might generate actions for the current time step. Only the actions that have been generated so far in that process are used for extending the planning graph. When certain conditions hold, the algorithm starts to search for a solution within the planning tree and the planning graph. If no solution is found for the current plan length, the plan length is incremented and the whole process repeats.

GraphHTN produces plans in the Graphplan format, where several actions may occur at the same time if they do not interfere with each other. The GraphHTN algorithm is sound and complete, and is guaranteed to report the plan with the shortest parallel length if a plan exists. The next subsection describes the search algorithm of GraphHTN, a required background for understanding the new enhancements.

The Search

The goal of the search algorithm is to find a *solution subtree* within the planning tree. A *solution subtree T'* is a subtree of a planning tree *T* with the following properties:

(1) The *root of T* is included in *T'*.
(2) If an instance of a compound task *t* is included in *T'* than exactly one of *t*'s children is included in *T'*.
(3) If an instance of a method *m* is included in *T'* than all the subtasks of *m* are included in *T'*.
(4) Each node *n* in *T'* is assigned with two properties: *start time* and *end time*. *start time* is the earliest time step in which an action in the subtrtee of *n* starts. *end time* is the latest time in which an action in the subtree of *n* is finished. The primitive tasks in *T'* with

their *start time* assignment define the solution plan (the "*solution*")

(5) The preconditions of the primitive tasks in *T*' and the constraints of the methods in *T'* are all satisfied when the solution plan is performed.

The highlighted subtrree in Figure 1 is an example for a *solution subtree*. The time interval [*start*, *end*) indicates the *start* and *end time* assigned to the corresponding task.

The search for a solution subtree is performed in top-down right-to-left manner. The top-down direction means that nodes are selected for the *solution tree* only after the selection of their ancestors. The right-to-left direction means that the algorithm selects the tasks that it will try to finish at time *t* before selecting the set of tasks to be finished at *t* – 1. During the search the algorithm maintains two types of sets:

$Tasks_t$ – the set of tasks the algorithm is committed to examine for selection at time *t*.

$Subgoals_t$ – the set of facts that the algorithm is committed to make true for time *t*. These commitments are the result of including in the solution tasks whose precondition have not yet been established.

At time *t* the algorithm scans the tasks in $tasks_t$ and decides for each task whether to select it (assigning to it the end time *t*+1) or to postpone its selection to an earlier time step (adding it to $tasks_{t-1}$). For a selected task, the algorithm selects a decomposition method and adds the subtasks that can be performed last (i.e., have no successor subtasks within the method) to $tasks_t$. The selection of a primitive task *pt* leads the assignment of the *start time t* to *pt* and to certain ancestors of *pt* (those that *pt* is their first offspring to be performed). When a *start time* is assigned to a task, certain predecessors of the task are added to $tasks_{t-1}$ (those whose all successors are already assigned with a *start time*). During that process the algorithm inserts the preconditions of the tasks that start at time *t* into $subgoals_t$. The algorithm avoids from selecting tasks that are mutually exclusive or tasks that delete facts in $subgoals_{t+1}$.

Decisions & Actions	$tasks_7$	$tasks_6$
Initialize $tasks_7$	{**Transport-p1**, Transport-p2}	{}
Select **Transport-p1** Decompose it using **m2**	{**Carry-direct-p1-t1**, Transport-p2}	{}
Select **Carry-direct-p1-t1** Decompose it using **m3**	{**Unload-top-p1-t1**, Transport-p2}	{}
Select **Unload-top-p1-t1** Decompose it using **m4**	{**Door-closed-t1**, Transport-p2}	{}
Select **Door-closed-t1** Decompose it using **m6**	{**Close-Door-t1**, Transport-p2}	{}
Select **Close-Door-t1** Start(Close-Door-t1) = 7 Start(Door-Closed-t1)= 7 Add Unload-p1-t1 to $tasks_6$ Add Open-door-t1 to $subgoals_7$	{**Transport-p2**}	{Unload-p1-t1}
Postpone **Transport-p2** (add it to $tasks_6$)	{}	{Unload-p1-t1, Transport-p2}

Figure 2: An example for the decisions and actions that are taken in searching for a solution that is eight time steps long, in the planning tree in Figure 1.

Figure 2 presents a search scenario for the tree in Figure 1. The search is for a plan with eight time steps (0 to 7). Initially the two *transport* tasks are inserted into $tasks_7$. The algorithm then scans and handles the tasks in $tasks_7$, and builds the list $tasks_6$ of tasks to be scanned at time step six.

Propagating Properties in the Planning Tree

In earlier versions of GraphHTN, the search process pruned the search space by using information that was propagated forward through the planning graph:

1. Which facts are reachable from the initial state, and which actions are applicable at each time step. Only applicable actions could be selected for a solution.
2. Which actions are exclusive to each other and therefore cannot be selected for the same time step.

This information is associated with time steps and does not supply clues to the search algorithm on the possible consequences of selecting a specific decomposition method for a time step. We assumed that by propagating certain properties through the *planning tree*, in the opposite direction to the search (bottom-up, left-to-right) the search process will be able to make better decisions on how to search as it goes right-to-left, top-down.

Therefore, we associate to each node *n* of the *planning tree* information about which facts could be added (PosAdd), deleted (PosDel), or required as a precondition (PosPre) as a result of selecting *n* for the solution. The next section describes how that information can be used to identify independent subproblems.

We explain how the PosAdd set of facts is computed. PosDel and PosPre are computed in a similar way. Since the search is performed top-down, right-to-left, we define the PosAdd facts to be either facts that could possibly be added by actions in the subtree of *n* (PosAddBy), or facts that could possibly be added by actions in the subtrees of nodes, that are necessarily ordered before *n* (PosAddBefore).

For example, in the planning tree described in Figure 3, the PosAdd set of door-closed(t1) includes four facts that are computed as follows:

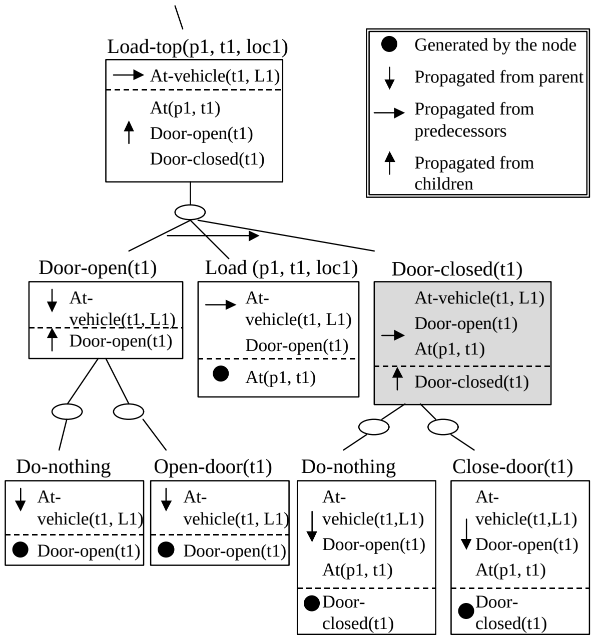


Figure 3: Computing the PosAdd set by propagating the PosAddBy and PosAddBefore set through the planning tree.

- Door-closed(t1) is part of the PosAddBy set and is propagated upward from subtasks of the Door-closed(t1) task.
- At-vehicle(t1, L1), Door-open(t1), and At(p1, t1) are part of the PosAddBefore set and are propagated from subtasks that are ordered before the Door-closed(t1) task.

The exact equations for computing PosAddBy, PosAddBefore, and PosAdd appear in Figure 4. The sets are computed by a single scan of the planning tree in roughly a depth first manner, at the beginning of the search process. Therefore, the time complexity for computing the properties is O(*N*) where *N* is the number of nodes in the planning tree. That computation does not change the overall time complexity of GraphHTN since *O(N)* is already required for building the tree. Empirically, the computation of the properties added less than 1% to GraphHTN's running time. For efficiency purposes, each property was encoded as a bit-vector, where each bit represents a different possible fact in the domain.

Identifying Independent Subproblems

Several works in the past (Korf 1987, Yang, Nau, and Hendler 1992, desJardins and Wolverton 1998) dealt with different ways for reducing the complexity of planning by working separately on smaller subsets of the original problem whose solutions can be merged. While our motivation is similar, our approach is different. We try to identify splitting opportunities that occur as a result of refining of the planning problem during the search, opportunities that do not necessarily exist in the original planing problem prior to its refinement. For example, in a two-package transportation problem, if the planner assigns each package to a different truck at some stage of the search, then the problem can be split into independent subproblems: each of which is responsible for transporting a single package.

PosAddBy(t) = AddEffects(t)
    if t is an instance of a primitive task

PosAddBy(t) = $\cup_{m\in \text{methods}(t)}$ PosAddBy(m)
    if t is an instance of a non-primitive task

PosAddBy(m) = $\cup_{t\in \text{subtasks}(m)}$ PosAddBy(t)
    if m is an instance of a method.

PosAddBefore(t) = PosAddBefore(m)
    if t is a subtask of method m and no other subntask of m is ordered before t.

PosAddBefore(t) = $\cup_{p\in \text{predecessor}(t)}$ PosAdd (p)
    if t is a subtask of method m that at least one other subtask of m ordered before t.

PosAddBefore(m) = PosAddBefore(t)
    if m is a method of t.

PosAdd(n) = PosAddBefore(n) $\cup$ PosAddBy(n)
    if n in is a node in the planning tree (method or task)

Figure 4: The set equations for computing PosAddBy, PosAddBefore, and PosAdd.

By identifying independent subproblems during the search and solving them independently we can achieve a significant complexity reduction. If a problem has two independent subproblems but the planner does not take advantage of the independence, then the cost of solving the problem behaves approximately according the following model:

TotalCost = Cost(subp1) * Cost(subp2)

If the planner handles the subproblems separately, then the cost is:

TotalCost = SplitCost +Cost(subp1) + Cost(subp2) + MergeCost

If SplitCost and MergeCost are relatively small, the splitting approach is cost effective. So the question is how to efficiently identify subproblems that can be split.

During the search process, GraphHTN selects at each time step *t* the set of tasks that it will try to finish at time *t* and defines a *refined problem* to be solved at time *t* –1.

Definition 1 (a refined problem). A *refined planning problem* in GraphHTN is the *n*-tupple:

$$<G, T, t, tasks, subgoals>$$

where *G* is the planning graph, *T* is the planning tree, *t* is the current time, *tasks* is the set of tasks the planner is committed to examine at time t, and s*ubgoals* are the set of subgoals that should be true at time $t + 1$.

Splitting the current problem into subproblems means to split the *tasks* and sub*goals* sets into separate sets. If certain conditions hold we call such a split an *independent split*.

Definition 2 (an independent split). Let *subp* be the refined planning problem GraphHTN has to solve at time t,

$subp \equiv <G, T, t, tasks, subgoals>$

and let $subp_1$, $subp_2$, … $subp_k$ be a set of subproblems of the form:

$subp_i = <G, T, t, tasks_i, subgoals_i> \quad \forall 1 \leq i \leq k.$

Then $subp_1$, $subp_2$, … $subp_k$ are an *independent split* of *subp*, iff the following properties are held:

(a) $tasks = \cup_{1\leq i \leq k} tasks_i$;

(b) $tasks_i \cap tasks_j = \emptyset$ for each $1 \leq i, j \leq k, i \neq j$;

(c) $subgoals = \cup_{1\leq i \leq k} subgoals_I$;

(d) $subgoals_i \cap subgoals_j = \emptyset$ for each $1 \leq i, j \leq k, i \neq j$;

(e) For each i, j such that $1 \leq i, j \leq k, i \neq j$:

(e1) $\text{PosAdd}(tasks_i) \cap \text{PosAdd}(tasks_j) = \emptyset$;[1]

(e2) $\text{PosPre}(tasks_i) \cap (\text{PosAdd}(tasks_j) \cup \text{PosDel}(tasks_j)) = \emptyset$;

(e3) $\text{PosAdd}(tasks_i) \cap \text{PosDel}(tasks_j) = \emptyset$;

(e4) $\text{PosPre}(tasks_j) \cap (\text{PosAdd}(tasks_i) \cup \text{PosDel}(tasks_i)) = \emptyset$;

(e5) $\text{PosAdd}(tasks_j) \cap \text{PosDel}(tasks_i) = \emptyset$;

(e6) $subgoals_i \cap \text{PosAdd}(tasks_j) = \emptyset$;

(e7) $subgoals_j \cap \text{PosAdd}(tasks_i) = \emptyset$;

Properties (a)-(d) assures that $subp_1$, $subp_2$, … $subp_k$ define a real split of the *tasks* and *subgoals* sets. Property (e1) is a technical way to assure that the search for each two subproblems will be performed in separate subtrees of the planning tree. Properties (e2)-(e5) assure that the solutions of the subproblems cannot interfere with each other. Properties (e6) and (e7) imposes that the subgoals will be split according to the subproblems that can achieve them.

An algorithm for finding an *independent split* according to the above definition is defined in the next subsection. Each independent subproblem that is found in the split is solved separately. The following proposition guarantees that the planner can solve the independent subproblems instead of the original problem.

Proposition. If $subp_1$, $subp_2$, … $subp_k$ is an *independent split* of *subp* then

(a) The union of every set of solutions sol_1, …, sol_k for $subp_1$, … $subp_k$ respectively, is a solution for *subp*.

(b) Every solution *sol* of *subp* is a union of solutions sol_1, sol_2, …, sol_k for $subp_1$, … $subp_k$ respectively

An outline of the proof.

Part (a) (soundness): the union of every set of solutions sol_1, … sol_k for $subp_1$, … $subp_k$, respectively, is a solution for *subp* because: (1) *subp* is the union of $subp_1$, … $subp_k$ (properties a-d) and, (2) there are no dependencies or interference between the solutions of the different subproblems (properties e2-e5)
Part (b) (completeness): we define sol'_i to be the set of elements in *sol* that are used for accomplishing the tasks of $subp_i$ or their predecessors. Based on properties e1-e7 we show that there is no overlap between sol'_i and that for each i, sol'_i is a solution for $subp_i$.

The Split Algorithm

The split algorithm makes use of the PosAdd, PosDel, and PosPre sets computed for the nodes of the *planning tree* at the beginning of the search. The search process calls the split algorithm after it completes to work on the current time step (t+1), and intends to work on the refined problem for the earlier time step (t). At that point the search has already defined $tasks_t$ and $subgolas_{t+1}$ and it passes these sets as arguments to the split algorithm.

The algorithm is shown in Figure 5. The general idea behind the algorithm is to create implicitly a dependency graph between the tasks in *tasks* and to find the connected components in that graph. These components represent the independent subproblems. The algorithm scans the tasks in *tasks*, and for each task t that has not been classified yet, it finds its closure – the set of tasks that depends on t or on other tasks in the closure of t. In the worst case, the dependency between each two tasks in *tasks* is checked. The cost of a single dependency check is $O(|F|)$ where F is the set of facts that are used as in the planning problem Therefore, the complexity of the split algorithm is $O(|F| * |tasks|^2)$. The algorithm is invoked each time the search process completes the assignment of tasks to a time step. Empirically, GraphHTN spent less than 3% of the overall planning time in that algorithm.

Finding Solutions for the Independent Subproblems

The set of independent subproblems $subp_1$, $subp_2$, … $subp_k$ of a subproblem *subp* are searched in sequence. A failure to find a solution for one of these subproblems means that there is no solution for *subp*, and a backtracking from *subp* should be performed. Solving an

[1] We define PosAdd for a set of tasks as the union of the PosAdd's of the individual tasks in the set. A similar extension is used for the definitions of PosDel and PosPre.

```
Algorithm Split(tasks, subgoals)
  candidates = tasks;
  subp_num = 0;
  Mark all candidates as unclassified;
  foreach task t in candidates do
     candidates = candidates – {t}
     if t is unclassified then // define a new subproblem
        subp_num = subp_num + 1;
        $tasks_{subp\_num}$ = {t}
        $subgoals_{subp\_num}$ = subgoals $\cap$ PosAdd(t);
        Mark t as classified;
        FindClosure(t, $tasks_{subp\_num}$, $subgoals_{subp\_num}$,
                    candidates, subgoals);
     end if;
  end foreach;
end.

Procedure FindClosure(t, tasks, subgoals,
                      candidates, all_subgoals)
  foreach task s in candidates do
     if Depend(t, s) then  // add the task to the closure
        tasks = tasks $\cup$ {s}
        subgoals = subgoals $\cup$
                (all_subgoals $\cap$ PosAdd(s));
        Mark s as classified;
        FindClosure(s, tasks, subgoals, candidates,
                    all_subgoals);
     end if;
  end foreach;
end;

Procedure Depend(s,t)
   Return false if all the properties e1-e5 hold for s
   and t; otherwise, return true.
end;
```

Figure 5: The split algorithm

independent subproblem resulting in assigning *start* and *end time* to nodes in the planning tree that participate in the solution. As a result, when all the subproblems are solved, the solution subtree is already marked, and no extra merging effort is required.

Experimental Comparison

In order to examine the value of splitting planning problems into independent subproblems we compared the performance of GraphHTN with and without the new splitting ability. As handling large domains is one of the important motivations for using HTN planners, we performed the tests on a set of problems from the UM Translog domain (Andrews et al. 1995). The UM Translog domain was inspired by the CMU logistics-transportation domain (Veloso 1992), but since UM Translog deals with various types of packages, vehicles, and procedures for handling the packages, it is an order of magnitude larger the CMU logistics. The UM Translog domain is specified in HTN using 41 operators and 45 decomposition methods. Typical problems in that domain deal with a large number of vehicles and locations (22 and 20, respectively in our test problems). The initial state of the test problems includes about 300 facts. The results of the performance tests are described in the Test and Results section.

Translating HTN Problems into STRIPS-style Problems

In general, HTN planning is strictly more expressive then STRIPS-style planning (Erol *et al*. 1994). That means that every STRIPS-style planning problem can be translated into an HTN planning problem, but not vice versa. The kind of HTN problems which are not translatable are those that include unbounded recursion among their methods[2]. However, in UM Translog like in many real life applications there are no recursive methods at all. Thus such domains can be translated in principle into a STRIPS representation.

In order to compare GraphHTN with action based planners such as IPP and Blackbox, we have defined an algorithm for translating non-recursive HTN domains into a STRIPS-style representation, and followed that algorithm in translating UM Translog into STRIPS. The resulting domain was too big for both IPP and Blackbox to handle successfully.

As a second trial we encoded the UM Translog domain into a STRIPS-style domain by hand, directly from the initial problem description. Using that domain we were able to run IPP on the set of problems and get performance results. These two experiences are summarized in the next sections.

An Algorithm for Translating HTNs to Operators

When no recursive methods exist in the domain an HTN planning domain can be translated into a STRIPS-style domain as follows[3]:

- Find all the *root tasks* in the domain – those tasks which not appear as subtasks of a method. For example, in the UM-Translog domain *transport* is the only *root* task.
- Decompose each *root task* according to its available HTN methods.

[2] By a “recursion” we mean a situation where one instance of a method is an ancestor in the planning tree of another instance of the same method.

[3] The algorithm can be extended to handle a bounded recursion, by starting the decomposition from the tasks in the initial task network (i.e., translating a problem and not just a domain)

- Continue to decompose the generated tasks recursively, until no more tasks can be decomposed (termination is guaranteed as there is no recursion among the methods).
- During that process, translate each occurrence of a method as described in the following paragraph; and translate each occurrence of a primitive task ad described in the paragraph after next.

Translating occurrences of methods. If m is an occurrence of a method than it is translated into a STRIPS operator o_m as follows:

- The preconditions of o_m assures the completions of the subtasks of m. If m decomposes the task instance t_0 into the subtasks $t_1,\ldots, t_k$, then the precondition of o_m consist of the propositions t_1-completed,…, t_k-completed, where t_j-completed is an artificial proposition that is added to the domain to indicate the completion of the t_j task.
- If the subtasks of the m are ordered, then preconditions are produced only for those subtasks that can appear last in the method. The completion of the other subtasks is taken care by the translation of the order constraints, as described in the paragraph after next.
- o_m has a single add effect t_0-completed which indicates the completion of t_0.

As an example see Figure 6. In that example, *door-closed*, which is the last subtask of the method, is the only subtask that generates a precondition for the operator. However, that precondition should be specific enough to represent the exact occurrence of the method and therefore has to be augmented by two additional arguments.

Translating occurrences of primitive tasks. If t is an occurrence of a primitive task, then its translation o_t has the same sets of preconditions, add-effects, and delete-effects, except that the add-effects of o_t also include the proposition t-completed.

Translating constraints. An order constraint between two instances of subtasks: $t_1 < t_2$ is translated by adding the precondition t_1-completed to every operator that is generated from the decomposition of t_2. For example, in the *load-top* method, the *load* subtask is ordered before the *door-closed* subtask. As a result, the precondition *load-completed*(?*p*, ?*t*, ?*o*) is added to operators that are generated from decomposing door-closed (such as the close-door operator).

To make things shorter we are not describing here how the rest of the method constraints (*before*, *after*, *between* and *initially*) are translated. In general, they contribute additional add-effects and preconditions to existing operators.

Translating the planning problem. An HTN planning problem in an HTN domain is specified by an initial task network to be performed and by an initial state. The initial task-network is translated into a STRIPS operator (*solve-problem*) exactly the way HTN methods are translated into STRIPS operators:

If m is an occurrence of the following method:

```
load-top(package ?p, truck ?t, location ?o):-
  n1: door-open(?t)
  n2: load(?p, ?t, ?o)
  n3: door-closed(?t)
 formula:
  n1 < n2 and n2 < n3 and …
```

then o_m is the following STRIPS operator:

```
op-load-top-method(package ?p, truck ?t, location ?o)
   :pre door-closed-completed-for-load(?p,?t,?o)
   :add load-top-completed(?p, ?t, ?o)
```

Figure 6: The translation of an occurrence of a method into a STRIPS-operator

- The preconditions of the *solve-problem* operator assure the completion of all the tasks of the initial task network.
- *solve-problem* has a single add effect: *solve-problem-completed*().

The STRIPS planning problem is specified by an initial state identical to that of the HTN problem and by a single goal: *solve-problem-completed*().

Limitations. We used that translation algorithm to translate a subset of the UM Translog domain into a STRIPS-style representation. We found that:

1) The readability of the resulting domain is poor. This is the result of adding operators, add-effects and preconditions to maintain the HTN constraints.
2) In addition to “real” operators, the resulting plans also include “artificial” operators that represent methods. It is quite easy to filter the “artificial” operators out of the plan. However, we cannot guarantee that the filtered plan still has the minimal parallel length, a property that is usually guaranteed by planners like IPP and Blackbox.
3) The performance of IPP and Blackbox on the resulting problems was very poor. They were able to solve only very simplified versions of the test problems. The main difficulty for these planners was the large number of instantiated actions. Since the STRIPS operators were used to represent occurrences of HTN methods (and occurrences of primitive tasks), they had to include additional variables to distinguish between the different occurrences of the same method (or between the different occurrences of the same task). These additional variables caused to a dramatic increase in the number of instantiated actions.

To overcome some of these problems we encoded the UM Translog problem into a STRIPS-style problem by hand, directly from the initial problem description (Andrews *et al*. 1995).

The Manual Encoding

In the manual STRIPS encoding of UM Translog we avoided using artificial operators for representing methods. We also tried to keep the number of instantiated actions small (although with a somewhat limited success).

The limited expressivity of STRIPS required:

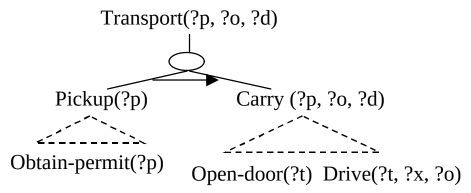
1) Splitting some general purpose operators such as *drive* into more specific operators such as *drive-regular, drive-flatbed*, etc.
2) Adding artificial add-effects and artificial preconditions to the operators to impose order requirements that exist in the domain.

The resulting domain was more compact than the one generated using the translation algorithm. IPP was able to solve planning problems in that domain, but only after invoking its RIFO preprocessor (Nebel, Dimopoulos, and Koehler 1997). RIFO filters out initial facts and operators that are not relevant for solving the problem. Since Blackbox is currently not distributed with such a pre-processor, it was not able to solve problems in that domain. The performance results for IPP are presented in the Tests and Results section.

Besides these performance issues, our experience exposed several difficulties in encoding big planning domains in the STRIPS representation. The following example illustrates some of these difficulties.

Example. In UM Translog it is required to obtain a permit for transporting a hazardous package before starting the actual procedure of transporting the package. This is a global requirement as it is not a natural pre-requisite of any of the actions involved in transporting of the package (*drive, load, unload,* etc.*)*. In HTN planning, we can simply express that requirement by creating two compound tasks: *pickup* which is responsible for preparation activities such as *obtain-permit*; and *carry,* which is responsible for carrying the package (see the illustration in the next column). An order constraint between the two tasks assures that any of the *carry* actions starts after the completion of all the *pickup* actions.

In the STRIPS representation, it is not clear in advance what will be the first action of carrying the package, so the precondition *has-permit*(?*p*) should be added to any action that can come first in the carrying phase. One of these actions is *open-doo*r(?*t*). Because *has-permit*(?*p*) is added



as a precondition of open-door, the package variable ?*p* should be added to the *open-door* operator:

```
open-door(truck ?t,  package ?p)
  :pre has-permit(?p)
       door-closed(?t)
  :add ….
```

That means that *open-door* becomes package-specific, and that it encodes a global requirement. Situations of that kind cause the following problems:

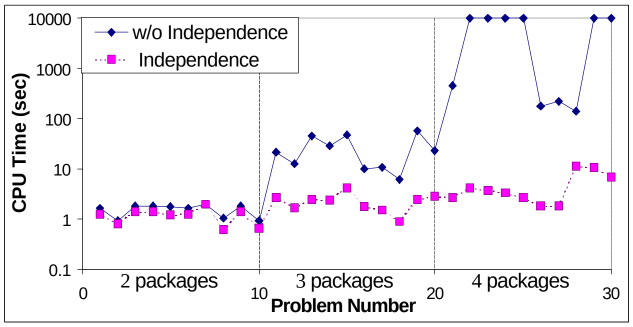
1) Encoding and debugging the STRIPS-style domain becomes difficult.
2) Since the basic operators encode global requirements, it is more complicated to make changes to the domain. For example, basic operators like open-door must be changed when the global requirement they encode is changed (e.g. removing the requirement for a permit for transporting a hazardous package).
3) The need to add additional variables to operators (as shown in the example) increases significantly the number of the instantiated actions.

Tests and Results

Methodology

Two experiments were conducted. The first compared the performance of GraphHTN with and without its new ability to split a planning problem into independent subproblems. The second, compared GraphHTN with UMCP (Erol 1994a), (an HTN planner which uses a classic refinement search), IPP (Koehler *et al*. 1997), and Blackbox (Kautz & Selman 1998). All tests were performed on a 400MHz Pentium II with 128MB RAM. No performance results are shown for Blackbox as it was not able to solve any of the problems on our machine.

The same set of problems was used for both experiments. All the problems are from the UM Translog domain. The problems were generated automatically for two, three, and four packages according to the same problem patterns that were used for testing UMCP (Tsuneto *et al.* 1998). The HTN version of the UM Translog domain was used for running GraphHTN and UMCP. The manual STRIPS encoding of UM Translog was used for running IPP.



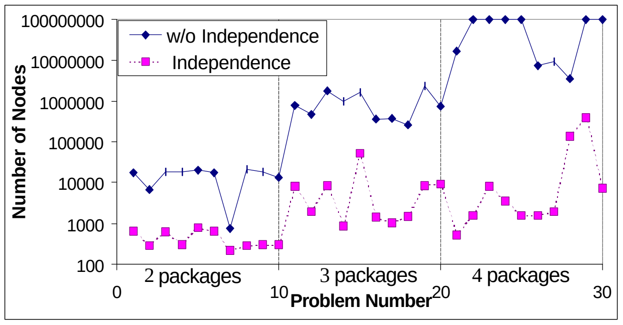
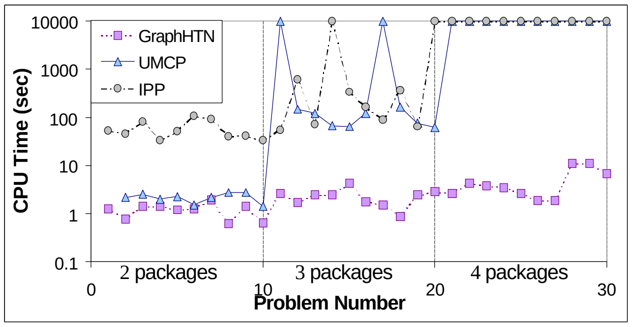


Figure 7: Solving the problem with and without identifying independent subproblems – CPU time and the number of search nodes



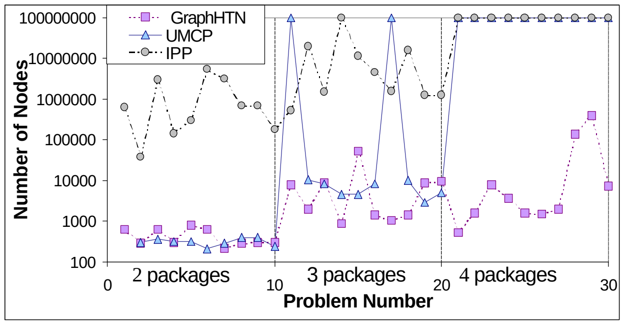


Figure 8: GraphHTN, UMCP, and IPP – CPU time and the number of search nodes. A maximal value is assigned to problems that were not solved by the tested planner (memory problems or others)

Results

Figure 7 presents graphically the total CPU time and the number of search nodes required for solving the problems by GraphHTN with and without identifying independent subproblems. Note that the diagrams are plotted with a logarithmic scale for the time and the number of nodes.

The number of search nodes explored by GraphHTN with the independence component was significantly smaller than running GraphHTN without it. For problems with three and four packages, the CPU time for solving the problems with the independence component was significantly shorter than solving them without it. For problems with two packages, building the graph was the dominant time, so there was no significant difference in the total running times.

Figure 8 presents graphically the total CPU time and the number of search nodes explored in solving the problems by GraphHTN, UMCP and IPP. For GraphHTN, we used the version with the independence component.

In comparison with IPP, GraphHTN ran significantly faster and explored significantly smaller number of nodes.

In comparison with UMCP, GraphHTN found plans with an optimal parallel length, while UMCP found plans, which are not necessarily optimal. GraphHTN also solved the problems significantly faster then UMCP. Furthermore, UMCP was not able to solve any of the problems with four packages. Although GraphHTN is written in C++ and UMCP is written in Lisp, it seems that due to the big performance gap, re-coding UMCP in C would probably not make a big difference. Regarding the number of search nodes, there was no big difference between GraphHTN and UMCP. Note however, that the amount of work and space which are required for exploring a search node of UMCP are inherently bigger than those which are required for exploring a node in GraphHTN.

Discussion and Conclusions

Our work presents a new way to extract knowledge on an HTN planning problem from its disjunctive representation, knowledge that can be used to speed up the search. We define a novel set of properties, associated with the nodes of the *planning tree* – a disjunctive data structure used by the GraphHTN planner to represent all the possible decompositions of an HTN planning problem. We show how these properties can be used by the search to split its current refined problem into independent subproblems. We show experimentally that by solving the independent subproblems separately and then merging their solutions, the planner achieves a significant performance improvement over the alternative of not splitting the problem. Since our approach for propagating properties through the *planning tree* is quite general, we believe it will be possible to use it for propagating other useful properties as well.
This work also makes a step toward examining disjunctive planners in more complex planning domains. In particular, we have compared GraphHTN, UMCP, IPP and Blackbox using the UM Translog domain, which contains the following complications: (1) large number of

operators, (2) large domains of variables, (3) large initial state, and (4) requirements on the desired order between actions in the plan.

On the tested problems GraphHTN had a significantly better performance than UMCP, IPP, and Blackbox (which was unable to solve any of the problems using our computer).

One of the important advantages GraphHTN had over UMCP was its ability to use, during the search, disjunctive information gathered on the problem prior to the search. In particular, GraphHTN used information on when it is possible to split the problem into independent subproblems.

The comparison between GraphHTN and action-based planners was more problematic, as the UM Translog domain had to be translated first into a STRIPS-style representation. For HTN problems like UM Translog in which no recursive method calls occur, such a translation can be done automatically. However, the result of that translation was a non-compact STRIPS-style domain that was too big for both IPP and Blackbox to handle. On the other hand, the alternative approach of performing the STRIPS encoding by hand (directly from the domain description) was difficult and error-prone. One of the reasons for this is the limited expressivity of the STRIPS language, which forces one to project high level requirements on the order of tasks to the level of actions.

In conclusion, UM Translog seems to represent a class of planning problems that can be solved much better by HTN planning than by action-based planning. However, some of the approaches that were used so successfully for action based planning (such as disjunctive planning) can be adopted successfully to improve the efficiency of HTN planners.

For the future, we plan to explore additional opportunities for using HTN specific properties, and to examine alternative approaches for searching the planning tree.

Acknowledgments

We wish to thank Jim Hendler for his helpful input on our work. This research was supported in part by the following grants and contracts: Army Research Laboratory DAAL01-97-K0135, Naval Research Laboratory N00173981G007, Air Force Research Laboratory F306029910013, and NSF DMI-9713718.

References

Andrews, S.; Kettler, B,; Erol, K.; and Hendler, J. 1995. Um Translog: A Planning Domain for the Development and Benchmarking of Planning Systems. Technical Report Department of Computer Science, University of Maryland College Park CS-TR-3487.

Blum, A. and Furst, M. 1997. Fast planning through planning graph analysis. *J. Artificial Intelligence*, 90(1–2):281–300.

Currie, K. and Tate, A. 1985. O-Plan - control in the open planning architecture. BSC Expert Systems Conference, Cambridge University Press.

DesJardins, M. and Wolverton, M. 1998. Coordinating Planning Activity and Information Flow in a Distributed Planning System, AAAI Fall Symposium on Distributed, Continual Planning (AAAI Technical Report FS-98-??), AAAI Press, 1998.

Erol, K.; Hendler, J.; and Nau, D. 1994a. UMCP: a sound and complete procedure for Hierarchical Task-Network planning, In *Proc. AIPS-94*, 249-254.

Erol, K.; Nau, D.; Hendler, J. 1994b. HTN planning: complexity and expressivity. *Proc. AAAI-94.*

Kautz, H. and Selman, B. 1998. Blackbox: A new approach to the application of theorem proving to problem solving. Working notes of the Workshop on Planning as Combinatorial Search held in conjunction with AIPS-98, Pittsburgh, PA, 1998, 58-60.

Korf, R. 1987. Planning as Search: A Quantitative Approach. *Artificial Intelligence* (33), 1987, 65-88.

Koehler, J. ; Nebel, B.; Hoffman, J; and Dimopoulos Y. 1997. Extending planning graphs to an ADL subset. In *Proc. ECP-97*, Toulouse, France, 1997.

Lotem, A., Nau., D., and Hendler J. 1999. Using Planning Graphs for Solving HTN Planning Problems. In *Proc. AAAI-99*

Mali, A.; and Kambhampati, S. 1998. Encoding HTN planning in propositional logic. In *Proc*. AIPS-98, 190-198.

Mali, A. 1999. Hierarchical Task Network Planning as Satisfiability. *In Proc. ECP-99.*

Nebel, B.; Dimopoulos, J.; and Koehler, J 1997. In *Proc. ECP-97*, 338-350.

Nau, D.; Smith, S. J.; and Erol, K. 1998. Control Strategies in HTN Planning: Theory versus Practice. *AAAI-98/IAAI-98*, 1127–1133.

Sacerdoti, E. 1975. The nonlinear nature of plans. In *Proc. IJCAI-75*, 206-214.

Tsuneto R.; Hendler, J.; and Nau, D. 1998. Analyzing External Conditions to Improve the Efficiency of HTN Planning. In *Proc. AAAI-98.*

Veloso, M. 1992. Learning by Analogical Reasoning in General Problem Solving. *PhD thesis, School of Computer Science, Carnegie Mellon University. Technical report CMU-CS-92-174.*

Yang, Q.; Nau, D.; and Hendler, J. 1992. Merging Separately Generated Plans with Restricted Interactions. Computational Intelligence Journal. Vol. 8, No. 4, pp. 648-676. 1992.