Using Planning Graphs for Solving HTN Planning Problems

Amnon Lotem, Dana S. Nau, and James A. Hendler

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

Abstract

In this paper we present the GraphHTN algorithm, a hybrid planning algorithm that does Hierarchical Task-Network (HTN) planning using a combination of HTN-style problem reduction and Graphplan-style planning-graph generation. We also present experimental results comparing GraphHTN with ordinary HTN decomposition (as implemented in the UMCP planner) and ordinary Graphplan search (as implemented in the IPP planner). Our experimental results show that (1) the performance of HTN planning can be improved significantly by using planning graphs, and (2) that planning with planning graphs can be sped up by exploiting HTN control knowledge.

Introduction

Recent approaches to action-based planning, such as Graphplan (Blum & Furst 1995; 1997), SATPLAN (Kautz & Selman 1996), and Blackbox (Kautz & Selman 1998), achieved significant progress in the performance of action-based planning, and extended the size of problems that action-based planners can handle. In this work, we try to transfer some of that progress to hierarchical task-network (HTN) planning (Sacerdoti 75; Currie & Tate 1985)—an AI planning methodology that creates plans by *task decomposition*. Our motivation to handle HTN planning stems from the fact that applied work in AI planning has typically favored approaches based on hierarchical decomposition rather than causal chaining approach (Wilkins & Desimone 1994; Aarup *et al.* 1994; Smith *et al.*, 1998; Nau *et al*., 1998).

We present the GraphHTN algorithm, which extends the Graphplan approach so that it can be used to solve HTN planning problems. GraphHTN compiles an HTN planning problem into two related compact data structures: a planning graph and a planning tree. The planning tree represents the HTN decomposition rules and constrains the planning graph to be consistent with the HTN constraints. The search for a plan is then performed in these two data structures.

In an initial set of experiments we compared GraphHTN with UMCP, an HTN planner which uses a classic refinement search. The planning time of GraphHTN was significantly shorter then that of UMCP, suggesting that using planning graphs is an attractive approach for HTN planning.

HTN problem specifications usually supply additional control knowledge that does not exist in the specifications for an action-based planner. For example, HTN control knowledge might include explicit specifications of the order between tasks and constraints on the possible instantiations of the tasks. The use of HTN constraints might reduce the searched space but it also introduces more nodes to the planning graph and requires additional work in handling each of these nodes. Thus, a legitimate question is whether incorporating HTN constraints in an algorithm which is based on a planning graph increases or decreases the planning time.

In order to answer that question, we did a second set of experiments that compared the time of solving HTN problems using GraphHTN with the time of solving equivalent action-based problems using IPP (Koehler *et al*., 1997). We selected IPP for this purpose because GraphHTN uses IPP's implementation of planning graphs, augmented by code to handle HTN constraints. The results of the comparison show that for small problems, the overhead of handling HTN control knowledge leads to larger planning times of GraphHTN in comparison with IPP. However, for large problems, handling HTN constraints was cost-effective, achieving an overall better performance of GraphHTN in comparison with IPP. That means that HTN control knowledge, if available, can be exploited effectively to reduce the planning time.

Background

Planning Graphs

The *Planning graph* was introduced (Blum & Furst 1995; 1997) as a compact way for representing a STRIPS planning problem. Planning graphs are currently used in some of the fastest available action-based planners, such as IPP (an extension of Graphplan) and Blackbox (as the basis

name	specification
by_truck	transport(?p, ?o, ?d):- n1: at-truck(?t, ?o) n2: ld(?p, ?t, ?o) n3: at-truck(?t, ?d) n4: uld(?p, ?t, ?d) formula: n1 < n2 and n2 < n3 and n3 < n4 and ?o ≠d and between(at-truck(?t, ?o), n1, n2) and between(at(?p, ?t), n2, n4) and between(at-truck(?t, ?d), n3, n4)
delivery	transport(?p, ?o, ?d):- n1: pckd(?p) n2: dlvr(?p, ?o, ?d) n3: vrfy(?p, ?d) formula: n1 < n2 and n2 < n3 and initially(small(?p)) and before(at(?t, ?o), n2) and between(dlvrd(?p, ?d), n2, n3) and ?o ≠ ?d
moving_truck	at-truck(?t, ?l):- n1: mv(?t, ?o, ?l) formula: before(not at-truck(?t, ?l), n1) and before(at-truck(?t, ?o), n1)
packing	pckd(?p):- n1: pck(?p) formula: before(not pckd(?p), n1)

Table 1: The methods of the simple transportation domain.

for the SAT encoding of a planning problem). A planning graph contains alternate levels of proposition nodes and action nodes. An action appears at level *i* if all its preconditions appear in the *i*-th proposition level. A proposition appears at the first proposition level if it is a part of the initial state. A proposition appears at level $i > 1$ if it is an "add" effect of some action in the previous action level. Graphplan constructs the planning graph level by level, and searches for a plan in the current graph after each extension of the graph. During that process the algorithm identifies, propagates and makes use of certain *mutual exclusion* relations among nodes.

Recent works have extended the types of problems that can be solved using planning graphs beyond problem described as pure STRIPS-style operators, to problem descriptions that allow unified quantifiers and conditional effects (Gazen & Knoblock, 1997; Koehler *et al.*, 1997, Anderson *et al.*, 1998), simple numeric constraints (Koehler, 1998) probabilistic operators (Blum and Langford, 1998), and sensing actions in the face of uncertainty (Weld *et al.*, 1998).

mv(?t, ?o, ?d) pre: at-truck(?t, ?o) post: ~at-truck(?t, ?o), at-truck(?t, ?d)	pck(?p) pre: not pckd(?p) post: pckd(?p)
ld(?p, ?t, ?l) pre: at(?p, ?l), at-truck(?t, ?l) post: ~at(?p, ?l), at(?p, ?t)	dlvr(?p, ?o, ?d) pre: at(?p, ?o) post: ~at(?p, ?o), dlvrd(?p, ?d)
uld(?p, ?t, ?l) pre: at(?p, ?t), at-truck(?t, ?l) post: ~at(?p, ?t), at(?p, ?l)	vrfy(?p, ?l) pre: dlvrd(?p, ?l) post: ~dlvrd(?p, ?l), at(?p, ?l)

Table 2: The primitive tasks of the simple transportation domain.

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* (*operators*). 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. The following constraints can be associated with a method: an *order constraint* between two subtasks *n1* and *n2* (*n1* < *n2*), *designation* and *codesignation* constraints between two variables or between a variable and a constant ($u = v$, $u \neq v$, $u = c$, $u \neq c$) and the following *state constraints*:

- *initially*(*P*(*?x*)) means that a specified proposition *P* should be true in the initial state.
- *before*(*P*(*?x*), *n*)) means that *P* should be true before starting subtask *n* of the method.
- after(*P*(*?x*), *n*)) means that *P* should be true after accomplishing subtask *n* of the method.
- *between*(*P*(*?x*), *n1*, *n2*)) means that *P* should be true between subtasks *n1* and *n2* of the method.

As an example, we will use the simple transportation domain presented in Tables 1 and 2. In this example, the *transport* compound task is responsible for transporting a package ?*p* from location ?*o* to location ?*d*. It can be performed by either using a truck or a delivery service. The *by_truck* method has four subtasks: having the truck ?*t* at ?*o*, loading the package into ?*t*, having the truck at ?*d* and unloading the package. *at-truck* is a predicate task which is decomposed into the primitive task *mv* (move) if the truck is not yet at the desired location ?*l* or into *do-nothing* if the truck is already at ?*l*. The subtasks of the method are totally ordered. *Between* constraints are used to protect the effects which are generated by one subtask and are consumed by another. The *delivery* method has three subtasks: having the package packed, delivery, and verification of the package arrival.

In general, HTN planning is strictly more expressive than STRIPS-style planning (Erol *et al*. 1994b). HTN planners traditionally use classical refinement search for finding a plan. A recent work showed how an HTN planning problem can be encoded as a propositional satisfiability problem (Mali & Kambhampati 98).

GraphHTN

GraphHTN is a novel algorithm for solving HTN planning problems based on the Graphplan approach. It gets as an input an initial task network *tn0* to be accomplished, an initial state *I*, and an HTN domain description *D* which specifies the primitive tasks (operators), compound tasks and the methods which can be used. GraphHTN produces plans in the Graphplan format: a set of actions and specified times in which each action is to be carried out. Several actions may occur at the same time if they do not interfere with each other.

Unlike Graphplan, GraphHTN has no goals that a valid plan should achieve. Instead a valid plan should be one which can be generated from the initial task network. More precisely, a plan *p* is valid iff there is a task network *tn* such that*:*

- *tn* can be generated from the initial task network *tn0* by a sequence of task decompositions and valid instantiation of variables.
- *tn* consists of only ground primitive tasks.
- The set of ground primitive tasks in *tn* is identical to the set of actions in *p*.
- The set of constraints associated with *tn* are satisfiable and *p* is consistent with these constraints.

The way GraphHTN finds valid plans will be illustrated using the simple transportation problem specified in Figure 1.

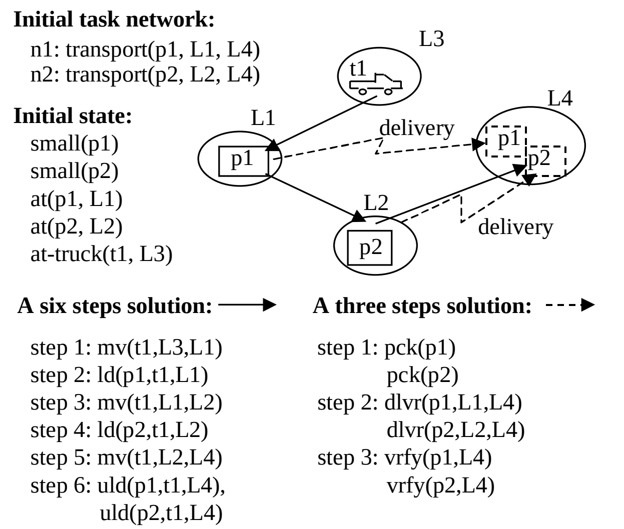

Figure 1: A simple transportation problem: two packages p1 and p2 should be transferred to L4 by either using a truck or a delivery service. Two possible solutions are presented.

The Planning Tree

A *planning tree* is an And/Or tree that represents the set of all possible decompositions of the initial task network up to a certain depth. Planning trees have two kinds of nodes: *task nodes* (the "or" nodes) and *method nodes* (the "and" nodes). The root of the planning tree is a special *method* node that is labeled *root.* The nodes immediately under the root represent the tasks in the initial task network. The

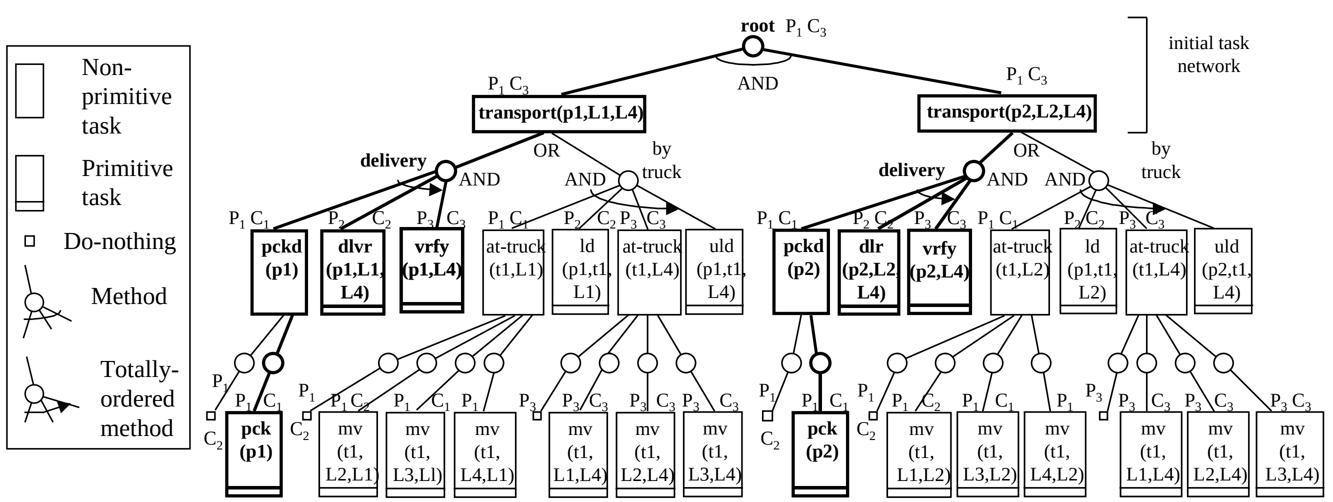

Figure 2: The planning tree for the simple transportation problems. Nodes which can generate actions for the *i*-th time step are marked by P_I. Nodes that *could be accomplished* at time *i* are marked by C_i.

children of a *task node* are all the instantiated methods that can be used for decomposing the task. The children of a *method node* are all the subtasks of the method. Figure 2 shows a planning tree for the simple transportation problem. Right under the root we have the two *transport* tasks. Both of them should be carried out in order to solve the problem. Each *transport* task can be decomposed using either the *delivery* or the *by truck* method, the *at-truck*(*t1*, *L1*) task can be decomposed to *do-nothing* or to *mv* operations from various locations, and so forth.

A specific solution for the planning problem corresponds to a subtree of the *planning tree* in which each non-primitive *task node* has exactly one *method node* as it child. For example, the subtree shown in boldface in Figure 2 represents a solution based on the *delivery* method.

The Planning Graph

GraphHTN uses a similar planning graph to the one used by Graphplan. The difference is that an action node might also represent the preconditions of a non-primitive task (as defined by *before* or *between* constraints). Such an action has no *add* or *delete* effects. Its appearance in the graph indicates that the corresponding task can start at that level.

A Description of the Algorithm

GraphHTN builds the *planning tree* and the *planning graph* synchronically and uses both of them in the search process. GraphHTN is sound and complete (Lotem and Nau 1999). However, as the HTN planning is semi-decidable (Erol 1994b), the termination of the planning process when no solution exists is not guaranteed. When no recursive method exists, the planner is guaranteed to report the *shortest* plan if a plan exists, and to halt if there is no solution.

An outline of the algorithm appears in Figure 3. GraphHTN starts with a planning tree consisting of the tasks in the *initial task network*, and a planning graph that has a single proposition level describing the *initial state.* Like Graphplan, the algorithm runs in stages. In each stage it appends to the planning graph one action level and one proposition level and searches backward for a plan in the extended graph. The HTN constraints are introduced by using the planning tree which guides both processes.

Extending the planning graph. Rather than using every applicable action for extending the graph, GraphHTN only uses actions that can be generated for the current time step by decomposing recursively the initial task network. That keeps the graph consistent with the HTN constraints (a similar idea was used (Barrett and Weld 1994) in extending UCPOP to handle tasks decomposition). Technically, this is done by extending first the prefix of the planning tree: expanding recursively the planning tree by decomposing each task *t* in the tree which holds the following conditions:

- Every task t_i that is ordered before *t* (i.e. $t_i < t$) has already marked as *could be accomplished* in a previous time step (tasks are marked as *could be accomplished* as part of extending the planning graph).
- All the preconditions of *t* (i.e., *before* and *between* constraints of *t*) are satisfied at the current proposition level.

Algorithm GraphHTN(*tn0*, *I*, *D*)
Input: An initial task network *tn0*, an initial state *I*, and
    an HTN domain description *D*.
Output: a valid plan or “Failure”.
Data Structures: A planning tree *T* and a planning graph *G*.
  Initially, *G* is empty and *T* has a single node: *root*.
Insert each $p \in I$ into the first action level of *G*.
Insert each $t \in tn0$ as a child of *T*’s root.
for *i* := 1 to max-length
  extend the prefix of the planning tree *T*;
  extend the planning graph *G*;
  if the root of the tree *could be accomplished* then
    search for plan *p* of length *i*;
    if solution was found then return “Success, plan is ”, *p*;
  end;
end;
return “Failure”.

Figure 3: an outline of the GraphHTN algorithm.

The expansion of a task *t* is done by generating a child node for representing every applicable instantiation of *t*’s methods. The instantiation of a method takes in account the current binding of free variables and the *designation*, *codesignation* and *initially* constraints of the method. For each new *method* node the algorithm creates the corresponding subtask nodes and continues to expand these new nodes recursively only if they hold the above conditions. The P_i label in Figure 2 indicates that the corresponding node was part of the prefix of the tree at time-step *i*.

The new actions that are created while extending the prefix of the tree are added to the *active set* of actions. Only actions from the *active set* are used (if applicable) to extend the planning graph. Including an action that represents a primitive task at level *i* of the graph means that the action *could be accomplished* in that time step. We extend that notion to methods and non-primitive tasks:

- A method *could be accomplished* at level *i* if all its subtasks *could be accomplished* at level *i* or earlier.
- A non-primitive task *could be accomplished* at level *i* if at least one of its methods *could be accomplished* at level *i* or earlier.

When an action is added to the planning graph the *could be accomplished* property is propagated upward in the tree. The C_i label in figure 2 indicates that the corresponding node was first marked as *could be accomplished* in step *i* of the algorithm.

The Search. In Graphplan a search is performed only if the last proposition level includes all the goals. In an HTN planning problem there are no goals to be achieved.

Instead, all the tasks in the initial task network should be accomplished. Therefore, the search starts when the root of the tree is marked as *could be accomplished*. Methods that could not be accomplished within the current number of time steps are filtered out of the planning tree and the planing graph (in the example: the two instances of the *by_truck* method are ignored by the search for a plan of length three). The search uses a level-by-level approach going backward from the last level of the planning graph toward its first level. However, in order to get a solution which is consistent with the HTN constraints, the search is guided top-down by the planning tree. This is done as follows:

- For each level i of the planning graph the algorithm selects the set of tasks to be *accomplished* at level i before proceeding to level i - 1. The algorithm does not decide at that point at which levels the selected tasks *start*.
- For each non-primitive task t that is selected to be accomplished at level i of the graph, the algorithm also selects a method for performing t. This is a backtracking point.
- The algorithm can select a task t to be *accomplished* at level i of the graph (i.e., t is *selectable*) only if:
  1) t is a subtask of a selected method m.
  2) Every other subtask t_i of m that is ordered after t ($t < t_i$) starts after time i.
  3) t is not mutual exclusive to any other task that has already been selected for level i.
  4) t does not delete any goals required for level i+1.
- The decision whether to select a selectable task t for the current level or to postpone its selection for a level smaller than i is also a backtracking point.
- Selecting a primitive task sets the *start* time of that task and possibly the start time of non-primitive tasks which are ancestors of t. The preconditions of the tasks that start at the i-th level of the graph, together with the open goals from levels greater than i, define the required goals for level i–1.

In the case of no recursive methods, the algorithm halts when a plan is found or when the length of the planning graph exceeds the maximum number of planning steps that can be generated by the planning tree. When the domain includes recursive methods, an *iterative deepening* approach is used in order to preserve the completeness of the algorithm. In the k-th deepening iteration, expansions are constrained to at most k recursion levels. If no solution was found in iteration k, the limit of the recursion level is increased by a predefined constant and the whole process repeats[1].

Due to space considerations, we will only mention here some of the additional extensions made to the algorithm to accommodate the requirements of HTN planning:

- mutual exclusiveness – eliminating *mutex* relations between an action and its ancestors;
- memoization – the set of tasks that can be selected for level i-1 is also recorded in the memoized configuration (in addition to the current set of goals);
- handling the *do-nothing* task;
- handling *after* and *between* constraints.

The implementation of GraphHTN consists of two major components: the HTN component, which is responsible for building and searching the planning tree, and the planning graph component. The planning graph component was adopted with some modifications from the IPP planner (Koehler *et al.*, 1997).

Experiments

Methodology

We evaluated the performance of GraphHTN by comparing it against UMCP (Erol 1994a), an HTN planner which uses a classic refinement search. We used two types of transportation problems. In problem 1, n packages in n different locations should be transferred to a common destination using a single truck. In problem 2 the packages should be transferred to n different destinations.

We used the same set of problems to assess the contribution of HTN control knowledge to the performance of the planner. We compared the time of solving the HTN problems using GraphHTN with the time of solving equivalent action-based problems using IPP (Koehler *et al*., 1997). We chose IPP for that purpose because GraphHTN uses IPP's implementation of planning graphs, augmented by code to handle HTN constraints.

Results

The running time and the number of search nodes explored by each planner are presented in Table 3. The times were measured on Sun Ultra with 143 MHZ clock and 64 MB RAM. Figures 3 and 4 present the running times graphically using a logarithmic scale for the time.

The comparison between GraphHTN and UMCP is not absolutely fair, as UMCP is written in lisp and GraphHTN is written in C and C++. However, the performance difference between the planners is so great that re-coding UMCP in C would probably not make a big difference. We also present in Table 3 the number of search nodes explored by GraphHTN and UMCP. However, this comparison is somewhat misleading, since creating a new node in UMCP requires duplication of the whole task

[1] We are currently examining an alternative approach in which the bound on the recursion level is dynamically deducted from the current length of the graph. As the length increases, nodes might be added to earlier levels of the graph. That excludes the need for iterations on the recursion level and assures the optimality of the extracted plan in terms of the number of time steps.

Problem	# of packages	Plan length	Total Elapsed Time			# of Search Nodes		
			Graph HTN	UMCP	IPP	Graph HTN	UMCP	IPP
1	2	6	0.10	1.2	0.04	94	65	47
	3	8	0.26	81.1	0.09	2541	2219	1092
	4	10	1.89	> 1 h	0.64	$3.9 * 10^4$	> $2 * 10^4$	$3.8 * 10^4$
	5	12	20.29	> 1 h	25.84	$4.8 * 10^5$	-	$1.7 * 10^6$
	6	14	192	> 1 h	1501	$5.2 * 10^6$	-	$8.6 * 10^7$
	7	16	2012	> 1 h	> 1h	$4.9 * 10^7$	-	-
2	2	8	0.14	1.7	0.07	278	107	133
	3	12	0.53	327	0.31	4883	7754	10^4
	4	16	3.27	> 1 h	14.19	$4.7 * 10^4$	–	10^6
	5	20	20.22	> 1 h	1731	$3.3 * 10^5$	-	10^8
	6	24	118	> 1 h	> 1h	$2.0 * 10^6$	-	-
	7	28	1419	> 1h	> 1h	$1.0 * 20^7$	-	-

Table 3 – The total elapsed time (in seconds) and the number of searched nodes for GraphHTN, UMCP and IPP.

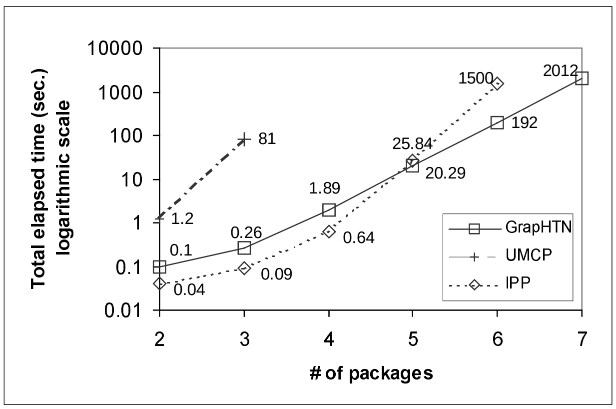

Figure 3: Total elapsed times for problem 1.

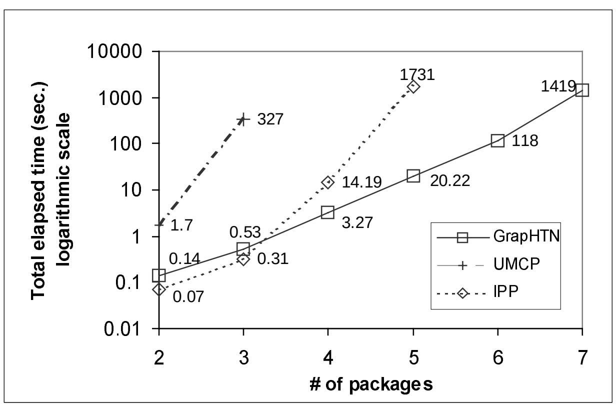

Figure 4: Total elapsed times for problem 2.

network, which involves much computation and memory. The failure of UMCP to solve the problems for more than three packages is basically due to its high consumption of memory.

The comparison with IPP shows that for small problems (up to 4 packages) the overhead of handling the HTN control knowledge led to larger planning times (using GraphHTN) than planning without it (using IPP). However, for large problems (more than 4 packages) handling HTN control knowledge was cost-effective, and led to shorter planning times than planning without it. The numbers of search nodes present a similar picture.

Discussion and Conclusions

We have presented GraphHTN—an HTN planner which compiles an HTN planning problem into a planning graph and a planning tree, and searches in this combined data structure for a plan. In our experiments, GraphHTN solved HTN planning problems significantly faster then UMCP. In addition, GraphHTN found plans that are optimal in terms of the number of time steps (this is not necessarily true for the plans found by UMCP). The primary reason for GraphHTN's fast performance relative to UMCP is somewhat similar to the reason for Graphplan's fast performance relative to UCPOP:

- The planning graph makes properties like reachability from the initial state and mutual exclusiveness explicitly available to the search phase, reducing significantly the amount of search needed.
- When failures occur for the same reason in different parts of a search space, the memoization reduces the amount of time spent in searching those different parts of the search space.

In addition, GraphHTN searches within a single (but complex) data structure, while UMCP maintains and explores hundreds of candidate task networks in its search process and thus exhausts the available memory quite fast.

We also tried to assess the value of the HTN control knowledge for reducing the planning time. There are two competing factors here:

- On one hand, there is an overhead in handling the HTN constraints: additional nodes should be introduced into the planning graph to represent preconditions of compound tasks and more time is spent in selecting a node for the solution in order to assure its consistency with the planning tree.
- On the other hand, even a small amount of HTN control knowledge might reduce the size of the searched space. For example, the *transport* method states that each package is loaded once and that this is done in the initial location of the package. In the action-based version of the problem this constraint is missing and the construction phase of the planning graph introduces for each package "load" actions that load the package at every possible location. As a result, the backward search spends extra time in eliminating these alternatives—time that is not needed in the GraphHTN algorithm.

In our experiments, for big enough problems, handling the HTN control knowledge was cost-effective and led to smaller planning times. We actually used in these experiments very limited amount of additional control knowledge—for example, we did not use GraphHTN's methods and operators as a vehicle for writing domain-specific planning algorithms, as has been done in some other HTN planners (Nau *et al.*, 1999). We believe that if we had made more control knowledge available to GraphHTN, its planning performance would have been even better.

For the future, we plan to compare GraphHTN and UMCP on a wider set of problems. We also plan to explore more systematically the value of different levels of HTN control knowledge on the performance of GraphHTN.

We intend to examine the role of propagating different HTN properties through the planning tree in limiting the search space and investigate alternative search strategies.

Another research direction is to replace GraphHTN's current search strategy by a process that encodes the planning graph and the planning tree as a propositional satisfiability problem and then uses a SAT solver for performing the search (similar to the way it is done in Blackbox for action-based planning problems).

Acknowledgments

We thank Jana Koehler for the permission to embed the code of IPP within GraphHTN. 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

Anderson, C.; Smith, D.; Weld, D. 1998. Conditional effects in Graphplan. In *Proc. AIPS-98*, 44-53.

Aarup, M. and Arentoft, M. and Parrod, Y. and Stader, J. and Stokes, I. 1994. OPTIMUM-AIV: A knowledge based planning and scheduling system for spacecraft AIV. *Intelligent Scheduling*, Morgan Kaufmann, Fox M. and Zweben M., 451-469.

Barrett, A. and Weld, D. 1994. Task decomposition via plan parsing. In *Proc. AAAI-94*, 1117-1122.

Blum, A. and Furst, M. 1995. Fast planning through planning graph analysis. In *Proc.14th Int. Joint Conf. AI*, 1636-1642.

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

Blum, A. and Langford, C. 1998. Probabilistic planning in the Graphplan Framework. Working notes of the Workshop on Planning as Combinatorial Search held in conjunction with AIPS-98, Pittsburgh, PA, 1998, 8-12.

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

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*.

Gazen, C. and Knoblock, C. 1997. Combining the expressivity of UCPOP with the efficiency of Graphplan. In *Proc. ECP-97*, Toulouse, France, 1997.

Kambhampati, S.; Knoblock, C.; and Yang, Q. 1995. Planning as refinement search: a unified framework for evaluating design tradeoffs in partial order planning. *Artificial Intelligence*, 76(1-2):167-238.

Kautz, H. and Selman, B. 1996. Pushing the envelope: Planning prepositional logic, and stochastic search. In *Proc. AAAI-96*, Portland, OR. 1996.

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.

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

Koehler, J. 1998. The IPP planner: exploring the possibilities of planning graphs. Working notes of the Workshop on Planning as Combinatorial Search held in conjunction with AIPS-98, Pittsburgh, PA, 1998, 71-74.

Lotem, A. and Nau., D. 1999. The Soundness and Completeness of GraphHTN. Working Paper.

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

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

Nau, D.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple Hierarchical Ordered Planner. *IJCAI-99*.

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

Smith, S. J.; Nau, D.; and Throop, T. 1998. Computer bridge: a big win for AI planning. *AI Magazine* 19(2), 93–105.

Weld, D; Anderson, C; Smith, D. 1998. Extending Graphplan to handle uncertainty & sensing actions. In *Proc. AAAI-98*, 897-904.

Wilkins, D. and Desimone R. 1994. Applying an AI planner to military operations. *Intelligent Scheduling*, Morgan Kaufmann, M. Fox and M. Zweben, 685-709.