Encoding HTN Planning as a Dynamic CSP*

Student: Pavel Surynek
Supervisor: Roman Barták

Charles University in Prague, Faculty of Mathematics and Physics
Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic
pavel.surynek@seznam.cz, roman.bartak@mff.cuni.cz

Abstract. Constraint satisfaction problems provide strong formalism for modeling variety of real life problems. This paper presents a work currently in progress of which the goal is an application of the CSP formalism on hierarchical task network planning domain. An encoding of HTN planning problems as a dynamic CSP is presented. We suppose that such encoding would provide a way for search space reduction through constraint propagation. The paper also deals with a search strategy and constraint combination suitable for the model.

1 Introduction

Constraint satisfaction methodology has proven to be a successful technique for solving variety of combinatorial and optimization problems. Despite this fact, it was exploited very little in the planning domain. In particular hierarchical task network planning (HTN) [3, 4] seems to be suitable for use of constraint programming. The formulation of HTN planning problem involves a lot of structural information which can be used to prune the search space. Encoding of this structural information by means of constraint programming would provide an effective way for such pruning during the search for solution.

This paper presents a work currently in progress of which the goal is to develop a framework and techniques for solving HTN planning problems using constraint programming methodology. The first step to achieve the goal is to propose a suitable encoding of HTN planning problems into constraints. The paper concentrates mainly on proposing such encoding. We also suggest a search strategy build upon the proposed encoding. Finally we discuss the construction of combined constraints which would provide stronger propagation.

2 HTN planning

From the traditional view of planning, a planning problem is posed as finding of a sequence of actions which transforms a specified initial state of the world into a desired goal state provided that only actions from a set of allowed actions can be used [1]. For

* This work is supported by the Czech Science Foundation under the contract 201/04/1102.

each action a set of preconditions and effects is specified. This approach to planning corresponds to the way, how so called STRIPS-style planners work [5].

The disadvantage of the traditional STRIPS-style approach is that it is not easy to incorporate any kind of domain knowledge into the description of the input planning problem. Since the planning problems are typically very hard to solve (EXPSPACE-complete), techniques for reduction of the search space are necessary. To overcome this drawback of STRIPS-style planning another model was developed.

HTN planning augments STRIPS-style action based model with a grammar of legal solutions and with reduction schemas. The key term of HTN planning formalism is so called *task network*. The task network is a syntactic construct of the form $(n_1{:}\alpha_1)(n_2{:}\alpha_2)...(n_m{:}\alpha_m) \mid \varphi$, where n_i are task symbols and each α_i is a task. φ is an additional Boolean formula that puts into relation the objects in the task network.

A task can be *primitive*, *compound,* or *goal*. Primitive tasks are similar to actions in STRIPS-style planning. Each primitive task has its preconditions and effects (specified as sets of literals that must hold). Each compound task is associated with a task network which must be solved to fulfill the compound task. The pair compound task and its task network defines a *reduction schema*. Similarly every goal task has specified the task network that must be solved. Primitive tasks are denoted as $do[f(v_1,v_2,...,v_n)]$, $perform[t(v_1,v_2,...,v_n)]$ denotes a compound task, where f and t are task symbols and v_i are variables. A goal is denoted as $achieve[l]$, where l is a literal. Compound and goal tasks are often referred as non-primitive tasks.

An *HTN planning problem* is specified by an initial state and a task network to solve. An example of a simple HTN problem is shown in example 1.

Example 1. A package must be transported from location L_1 to location L_2. A truck and a train are available for transportation. The truck can go to any place, while the train moves only between goods stations. It is necessary to bring the package to the station first if we want to transport it by the train. To simplify the description several obvious constructs are omitted.

TransportPackage :- d_1 = ((t:perform[TransportPackageByTruck]) | φ_1 = additional Boolean formula) **or** d_2 = ((t:perform[TransportPackageByTrain]) | φ_2)
TransportPackageByTruck :- d_3 = ((a_1:do[moveTruck(L_3,L_1)]), (a_2:do[loadTruck(L_1)]), (a_3:do[moveTruck(L_1,L_2)]), (a_4:do[unloadTruck(L_2)]) | φ_3)
TransportPackageByTrain :- d_4 = ((a_1:do[moveTruck(L_3,L_1)]), (a_2:do[loadTruck(L_1)]), (a_3:do[moveTruck(L_1,S_1)]), (a_4:do[moveTruck(S_3,S_1)]), ... | φ_3)
achieve[packageAtDestination] :- d_5 = ((t:perform[TransportPackage]) | φ_5)
Constant symbols for three locations and three good stations: {L_1,L_2,L_3,S_1,S_2,S_3}
Initial state: I = {packageAt(L_1), truckAt(L_3), trainIn(S_3)}
Task network we want to plan for: d = (g:achieve[packageAtDestination])
Example of **additional formula**: φ_3 = {(a_1 before a_2) & (a_2 before a_3) & (a_3 before a_4)}
Example of **primitive task**: moveTruck(L_1,L_2)= (preconditions:truckAt(L_1)) (effects:truckAt(L_2),¬truckAt(L_1)).

---

3 Dynamic constraint model of HTN planning problem

We encode HTN planning problem as a dynamic constraint satisfaction problem. The dynamicity of our constraint model consists in changes that we made during the

search for solution. As the search proceeds and earlier decisions of the search algorithm become fixed, the model is extended with the parts modeling later decisions.

A dynamic constraint satisfaction problem [2] is a sequence of CSPs $P_1,P_2,...,P_\alpha,...$, where each problem is a result of a modification of the preceding one. All the static problems in the sequence must be solved to solve the DCSP. However, in our modification it is sufficient to solve the last static problem in the sequence.

Given an HTN problem Q, the corresponding DCSP model encodes a problem of finding k-step plan for Q. First we will describe static characteristics of our encoding.

Three types of variables are used to encode the k-step planning problem. Variables of the first type encode ground instances of primitive tasks. Their domains contain possible steps of execution of the corresponding primitive task. We will refer to these variables as *primitive task variables*. Primitive task variables are always bound with its superior non-primitive task (with respect to reduction schemas). Primitive task variables will be denoted as $a_i(p)$, where p is the primitive task and i is an index used to distinguish its superior non-primitive task.

Non-primitive tasks are encoded using the variables of the second type. Their domains correspond to sets of possible task reductions, i.e., the domain contains an element for each ground task network that solves the non-primitive task. Variables of this type will be referred as *non-primitive task variables*. Non-primitive task variables are bound with its superior task as well. $t_i(n)$ will be used to denote compound task variables and $g(n)$ will denote goal task variables, where n is the non-primitive task and i is the index with the same meaning as in the case of primitive tasks.

States of the world are expressed through posting sets of ground atoms true in that state. For each predicate symbol and step in the plan we define a variable. We will refer to these variables as *state variables*. The state variable for predicate p of arity a at step s will be denoted as p_s/a. The domain of the variable consists of all ground instantiations of the corresponding predicate. We can restrict ground instantiations only to those that make sense for the predicate symbol. Since the language of HTN planning does not contain function symbols and the number of constant symbols is finite, the number of all ground atoms is also finite. This fact allows the state variables to have finite domains. An example of variable construction for an HTN problem from example 1 is shown in example 2.

Example 2. Variables modeling the HTN problem from example 1 for four steps. The example illustrates types of variables and their domains used to model the problem. It does not show conditions under which they appear in the model.

A **goal task** variable: g(achieve[packageAtDestination]) $\in \{d_5\}$
A **non-primitive task variable** corresponding to choosing d_5 to satisfy the goal task:
t_1(TransportPackage) $\in \{d_1, d_2\}$
A **non-primitive task variable** corresponding to choosing d_1 to satisfy the task
TransportPackage: t_2(TransportByTruck) $\in \{d_3\}$
Example of **primitive task variable** corresponding to d_3 that satisfies the task
TransportPackageByTruck: a_1(moveTruck(L_3,L_1)) $\in \{1,...,4\}$
Example of **world states** variables: packageAt$_i$/1 $\in \{(L_1);(L_2);(L_3)\}$, for all i $\in \{1,...,4\}$
A set variables of the CSP: **X** = {g(achieve[packageAtDestination]),
t_1(TransportPackage), ..., a_1(moveTruck(L_3,L_1)), ..., packageAt$_i$/1, ...}.

Now it is necessary to specify constraints that define relations between suggested variables. A set of preconditions and a set of effects are given for each primitive task. The preconditions are specified as a set of literals that must hold before the action can be executed. Similarly, the effects are specified as a set of positive and negative literals. We can easily incorporate action's preconditions and effects into the model by adding a constraint for each primitive task that puts into relation every matching primitive task variable with the relevant world state variables.

HTN formalism also allows the user to post additional constraints in the form of a Boolean formula for every task network (reduction schema). The formula usually expresses temporal ordering of the individual tasks. Even though there are virtually no restrictions on the constructs used in the formula, it is possible to incorporate the additional constraints into the encoding in a straightforward way.

Example 3. An example of constraint model for the HTN problem from example 1. Constraints are denoted as *C(d)*, *E(d)*, *B(n)*, *G* and *I*, where *d* is the task network (selected reduction) and *n* is the non-primitive task. *C* constraints represent reduction sub-problems corresponding to the selected reduction schema, *B* constraints bind satisfiability of those reduction sub-problems with the values in the domains of non-primitive task variables. *E* constraints encode primitive task's preconditions and effects, *I* constraint defines the initial state, and finally *G* is a constraint saying that the goal must be satisfied to solve the problem. For simplicity reasons, only a cutout of the complete set of constraints is presented.

A constraint representing the **goal**: **G** = (C(d_5) **and** B(achieve[packageAtDestination]))
A constraint **binding** non-primitive task variable's domain with reduction sub-problems:
B(achieve[packageAtDestination]) = (**if** g(achieve[packageAtDestination])=d_5 **then** C(d_5))
A constraint representing **reduction sub-problem** corresponding to choosing d_5:
C(d_5) = ((C(d_1) **or** C(d_2)) **and** B(TransportPackage))
Another example of **binding** constraint:
B(TransportPackage) = ((**if** t_1(TransportPackage)=d_1 **then** C(d_1)) **and** (**if** t_1(TransportPackage)=d_2 **then** C(d_2)))
...
An example of the **reduction sub-problem** constraint at the **bottom** of the hierarchy:
C(d_3) = (a_1(moveTruck(L_3,L_1)) < a_1(loadTruck(L_1))) **and** (a_1(loadTruck(L_1)) < < a_1(moveTruck(L_1,L_2))) **and** (a_1(moveTruck(L_1,L_2)) < a_1(unloadTruck(L_2))) **and** E(d_3)

A constraint encoding primitive task's **preconditions and effects**:
E(d_3) = (**if** a_1(moveTruck(L_3,L_1))=1 **then** (($truckAt_1$/1= (L_3) **&** $truckAt_2$/1=(L_1) **&** **&** $truckAt_2$/1)≠(L_3)) **and** (**if** a_1(loadTruck(L_1))=2 **then** ($truckAt_2$/1=(L_1) **&** **&** $packageAt_2$/1=(L_1) **&** $packageInTruck_3$/1=**true** **&** $packageAt_3$/1≠ (L_1)) **and** ...

Initial state constraint: **I** = (packageAt/1=(L_1) **and** truckAt/1=(L_3) **and** trainIn/1=(S_3))
The **final CSP** problem **(X,C)** = (X, {G**,** I**}**) (X is the set from example 2).

The constraint model (example 3) is constructed hierarchically with goal task constraints on the top and primitive task constraints at the bottom. Each value in the domain of a non-primitive variable represents a decision which reduction schema is selected to achieve the task. If a reduction schema is selected, a set of constraints corresponding to this selection must hold. Those constraints form some kind of a sub-problem, we will refer to these sub-problems as *reduction sub-problems*. Having the

non-primitive task at least one of its reduction sub-problems must have solution, thus disjunction of the sub-problems must be satisfied. Each reduction sub-problem further consists of reduction sub-problems according to the task hierarchy. The lower levels of the model hierarchy consist of constraints binding primitive task variables.

The solution of the traditional CSP model is a complete instantiation of the variables such that all the constraints are satisfied. But here it is sufficient to instantiate variables that are relevant to the selected reductions, the remaining variables can have assigned an arbitrary value from their domains. Similar approach is also known as a *conditional* CSP [6].

4 Search strategy and combined constraints

The model described in the previous section is constructed dynamically as the search proceeds. The search can be built upon any backtracking based algorithm for constrain satisfaction. When non-primitive task variables are instantiated the model is extended with the corresponding reduction sub-problems. The depth of the hierarchical model is always limited. There is the only condition that must be satisfied to ensure completeness of the search. When the domain of a non-primitive task becomes singleton, i.e., it contains only one element, the model must be extended with the corresponding reduction sub-problem. The extension of the particular branch of the model stops, when primitive tasks are reached on the branch or the number of allowed steps is exceeded. Figure 1 shows a simple framework of the backtracking based search algorithm for solving suggested DCSP encoding.

Algorithm 1. A basic framework of the search algorithm for solving DCSP encoding.

```
function Find_k-Plan((X,C))
1  v←variable to instantiate suggested by variable
   ordering heuristic
2  dv←value from the domain of v suggested by value
   ordering heuristic
3  C←C ∪(v=dv) and propagate the constraint (v=dv)
4  if (X,C) is solved then return true
5  (X,C)←extension of (X,C) according to the
   encoding extension criterion for k-step plan
6  return Find_k-Plan(X,C)
```

To build an algorithm it is necessary to have a *variable* and a *value ordering heuristic* that are aware of the used encoding. The preferred order of variable instantiation is top-down. Variables that represent tasks higher in the task hierarchy should be instantiated earlier. The second necessary ingredient is an *encoding extension criterion*. It tells how to extend the CSP with the reduction sub-problems if some values are ruled out from the variable domains (non-primitive task reductions become fixed). The model is always extended to a limited depth with respect to the task hierarchy. The deeper the extension is the more the problem is constrained and the search space

can be pruned more. On the other hand the memory consumption may be higher. We expect that extension of a small number of layers of the task hierarchy would be the best compromise.

The crucial technique in constraint programming is constraint propagation. In our CSP model we often use constraint combination via logical conjunctions. Although *reification* [7] is the most frequently used technique for constraint combination, it is unsuitable for our model. The disadvantage of this approach is that it can result in weaker constraint propagation. Fortunately there exists a different technique for combination of disjunctive constraints. *Constructive disjunction* [8] provides much stronger constraint propagation which is much more suitable for our model.

5 Conclusion and future work

In this paper we propose a new approach for solving HTN planning problems based on constraint satisfaction. An encoding of the HTN planning problem as the DCSP is suggested. We also propose the framework of the backtracking based algorithm for solving such DCSP problem. We hope that our encoding would help to reduce the size of the search space by early pruning of parts not containing any solution.

We are starting to implement the proposed framework for working with HTN problems in order to make necessary experiments and to precise the details of encoding. Especially the dynamic change of the model, i.e., the encoding extension criterion needs to be specified more exactly. This is impossible without both algorithmic and experimental studies. Next we need to explore how the constraint propagation works for our model. In particular the benefits of constructive disjunction must be verified.

References

1. J. Allen, J. Hendler, A. Tate: Readings in Planning. Morgan Kaufmann Publishers, 1990.
2. R. Dechter, A. Dechter: Belief Maintenance in Dynamic Constraint Networks. In Proceedings the 7th National Conference on Artificial Intelligence (AAAI-88), 37-42, 1988.
3. K. Erol, J. Hendler, D. S. Nau: UMCP: A Sound and Complete Procedure for Hierarchical Task Network Planning. In Proceedings of the 2nd International Conference on AI Planning Systems (AIPS-94), 249-254, 1994.
4. K. Erol, J. Hendler, D. S. Nau: HTN Planning: Complexity and Expressivity. In Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94), 1123-1128, 1994.
5. R. E. Fikes, N. J. Nilsson: STRIPS: a new approach to the application of theorem proving to problem solving. Artificial Intelligence, 2 (3/4):189-208, 1971.
6. S. Mittal, and B. Falkenhainer: Dynamic Constraint Satisfaction Problems. In Proceedings of the 8th National Conference on Artificial Intelligence (AAAI-90), 25–32, 1990.
7. Ch. Schulte: Programming Deep Concurrent Constraint Combinators. In Proceedings of the 2nd International Workshop on Practical Aspects of Declarative Languages (PADL-2000), 215-229, 2000.
8. J. Würtz, T. Müller: Constructive Disjunction Revised. In Proceedings of the 20th Annual German Conference on Artificial Intelligence (KI-96), 377-386, 1996.