Hierarchical Task Network Planning with Answer Set Solving

A Translation Software from SHOP2's Lisp code to Gringo and Clasp

Arkadi Schelling*

April 4, 2015

Contents

*Student of the M. Sc. "Cognitive Systems" at the Univ. of Potsdam, Stud.-ID 779135, arkadi.schelling@gmail.com

1 Introduction

Hierarchical Task Network planning – or HTN Planning – is an approach in Artificial Intelligence to automated planning and scheduling. Chapter 2 explains the basics of HTN planning. There are several domain-independent HTN-planning softwares. Section 3.1 deals with one of the most performant such systems – SHOP2. A more general domain-independent planning software is Answer Set Solving, like the Potassco system explained in section 3.2. This article tries to use the more general approach of Potassco to solve HTN problems. For comparison of the two approaches a translation script has been written and is explained in chapter 4. Due to the complex optimization of SHOP2 to solve HTN planning, this translation script could not be completely finished. The current incomplete state of planning HTN problems with Answer Set Solving does not keep up with the performance of SHOP2. Still, the rough theoretical treatment of the two systems' differences in chapter 5 suggests that even a complete translation will not change this inferiority of Answer Set Solving to SHOP2 in the realm of HTN planning.

2 Hierarchical Task Network Planning

HTN planning is a branch of AI that tries to realize a sequence of actions to reach given goals. For this the world is regarded as a collection of states, where the states can be changed by actions. A problem description needs to describe the rules how an action changes the states of the world. An instance of the problem consists of initial states and goal states, for which the planner has to find a solution, that is a sequence of actions to change the initial states into the goal states. For HTN the given restrictions to this general description are subject to further simplifications of the following form:

- The initial state is unique and known
- The actions are deterministic
- Only one action at a time is possible
- There is only one agent

This set of simplifications is known as the Classical Planning Problem.[1] In this paper we are still imposing further simplifications:

- There are only finitely many initial and goal states
- The state values are discrete

These simplifications are necessary, since neither the SHOP2 System nor the Automatic Promblem Solving with Gringo and Clasp can deal with infinite sets. Further on, another feature of HTN problems is the addition of methods to the problem description. Such methods are compound tasks, consisting of a sequence of actions or simpler methods, i. e. methods that consist of less actions. These methods help in two ways – First, they can be used to decrease

[1] `http://en.wikipedia.org/wiki/Automated_planning_and_scheduling`

the size of the search space. Second, the are closer to the way humans think how to solve a task, and can thus easilier be described.

Since each action, also called primitive task, changes certain states to other states, it has exactly the deleted states as a prerequisite for its application. The compositionality of methods lets them inherit these prerequisites from its primitive tasks. Due to these constraints on the application of methods we are thus having a network of tasks – a Hierarchical Task Network.

To represent an HTN planning problem we need to be able to represent the following:

1. Atoms that represent our world
2. Constants and variables
3. Arithmetic functions
4. Initial states, states at different times and goal states
5. Actions, including prerequisites, deleted states and added states
6. Sequences of actions and compound actions

Further on we have to be able to check which actions are not constrained and choose one of these as the next action as long as the goal states are not yet reached completely.

3 Design of the System

This section describes how an HTN problem can be solved by two different systems. First, the SHOP2 system is explained, second, the Gringo and Clasp framework for ASS.

3.1 SHOP2

The SHOP2 system is an HTN planner from the University of Maryland written in Lisp around 2000 and winning important rewards on the International Planning Competition 2002. The basic SHOP system is described in [NCLMA99] and its improved successor SHOP2 in [N+03]. It plans for tasks in the same order as they shall later be executed. This is called a total-order forward search. An explanation of the algorithm is found at [N+03, p. 386ff], which basically describes it as non-deterministically choosing next steps that are without constraints. The main upside of this approach is that the system always knows the exact state of the world. Therefore, at each decision point the system knows all constraints and the space of possible tasks is as small as possible. The downside of this approach is that the search space for possible solutions can only be searched by forward propagation. Other algorithms try to tackle the big size of the search space by parallel solution of sub tasks.

In the given example problem descriptions of the SHOP2 System this downside is alleviated by three basic techniques of "hand-tailoring" a problem:

- Constructing higher order tasks, so called methods, that can be broken down into first order tasks. For all example problem descriptions this goes as far as creating a method to solve a whole problem instance, consisting of smaller and smaller sub-methods. This decreases the search space immensely.
- Flagging items according to their goal state, e. g. boxes that have to be moved. This is essentially imposing a search logic that is not forward-propagating.
- Imposing axioms onto the states. An axiom is a relation between states, that has to hold during all steps. These axioms can be used additional to the prerequisites of methods and tasks to reduce the space of non-constrained next steps.

According to [N+03, p. 382] the domain knowledge encoded in the methods can reduce the complexity of the problem from exponential to polynomial time.

3.2 Using Gringo and Clasp for HTN Planning

Answer Set Solving is a declarative programming technique, that tries to solve big search problems. A search problem is expressed as a set of logical rules. A stable model for this set of rules is a solution to the problem.

The programs Gringo and Clasp are part of the Potsdam Answer Set Solving Collection.[2] Current Answer Set Solvers like Clasp are working without variables. To solve a problem with variables, a grounder is needed to create an equivalent variable-free program. Gringo is such a grounder. Clasp is following the paradigm of conlict-driven Answer Set Programming. The classical paradigm of ASP has been the DPLL approach with unit propagation and backtracking to solve conflicts. This approach has been improved with conflict-driven constraint learning (CDCL). In contrast to DPLL the CDCL approach is learning new constraints from the derived conflicts. It also performs backjumping instead of backtracking, thus finding solutions in very few steps.

We can use Answer Set Solving to solve HTN problems. For this approach we compare the list of required descriptivity on page 3 with the syntax of ASP. The first three points can directly be translated as atoms, constants, variables and the implemented arithmetic functions. The major problem we are running into when formularizing a sequence of actions is the formularization of time. We cannot set the maximum time to infinity, since this would lead to an infinite amount of possible solutions. Nevertheless, we do not know the actual size of the solution, which can in fact be arbitrary large. As a simple solution we set a maximum number of steps, not allowing bigger solutions.

In this setting with time steps, a state will be a function on the time and possibly more than one variable. An action will change the deleted and added states, all other states shall not change – the law of inertia. To solve a problem we simply have to choose an action at each step from the set of non constrained actions until the goal states are all reached.

This way of finding a solution is neither quick, nor will it find optimal solutions. To find optimal solution each action should be assigned a cost. We can then use the optimization methods of Clasp to minimize these costs.

[2] `http://potassco.sourceforge.net/`

It is a bigger question to improve the runtime. For this, it is possible to include further optimization as in the SHOP systems. One easy implementation is also including axioms, i. e. Horn Clauses, to reduce the number of non constrained actions at each step. Further, methods can be created by combining single actions into a sequence, that has special prerequisites to be called. These methods again allow for optimization techniques like auxiliary states and the implementation of domain specific knowledge.

3.3 Implementation of HTN planning in ASP

This section is explaining the basic constructs to solve HTN planning with ASP as explained in the previous section 3.2.

Since functions cannot be applied to arbitrary atoms, we need to save a domain with them. Initial states are simply states at time 0. A typical instance of a problem is given by this small block problem. Note that the description of the initial state is a complete and explicit description, i. e. there is no indirect conclusion that a block is clear, since no other block is on it:

```
% An atom is of the form domain(atom). %
block(b1).
block(b2).

% An initial state is of the form state(state_name(atoms), 0). %
state(ontable(b1), 0).
state(clear(b1), 0).
state(ontable(b2), 0).
state(clear(b2), 0).

% Goals are of the form goal(state_name(atoms)). %
goal(on(b2, b1)).
```

These initial states can be changed by operators. An operator has a name, variables that it can apply to and prerequisites when it can be applied. When the planner decides to use an operator as its current task, this is changing the state of the world.

```
% A typical operator %
operator(pickup).
operator_var(pickup, B) :- block(B).
operator_prerequisite(pickup, B, T) :- state(arm(empty), T), state(
    clear(B), T), state(ontable(B), T), operator_var(pickup, B).
deleted_state(arm(empty), T) :- currentTask(pickup, B, T).
deleted_state(clear(B), T) :- currentTask(pickup, B, T).
deleted_state(ontable(B), T) :- currentTask(pickup, B, T).
added_state(arm(B), T) :- currentTask(pickup, B, T).
```

The next part of code is used to describe the state of the world. They are changed by the applied operators. Even though there is no maximum number of steps that can solve any problem, we are setting it to 100, so we do not run into infinite solutions.

```
% Setting the maximum number of steps %
maxtime(100).

% Law of inertia %
time(0..T) :- maxtime(T).
state(X, T+1) :- time(T), state(X, T), not deleted_state(X, T).
```

```
state(X, T+1) :- time(T), added_state(X, T).
:- deleted_state(X, T), not state(X,T).

% Checking whether we are finished within the maximum time %
not_finished(T) :- time(T), goal(State), not state(State, T).
:- goal(State), not state(State, T), maxtime(T).
```

As long as the problem is not solved the following code guesses next steps. The second line minimizes the found solutions for their costs.

```
% Stupid Guessing Solver %
1 {currentTask(Op, Var, T) : operator_prerequisite(Op, Var, T),
    operator(Op) } 1 :- time(T), not_finished(T).
#minimize{C, T : action(Op, Var, T, TT, C)}.
```

These pieces of code are in principal enough to solve any given instance of a described problem. Still, without further optimization the solving will take a long time. One way to optimize further are additional states, that are derived by axioms, as in the SHOP system.

```
% A typical axiom (or Horn Clause) %
state(needtomove(B), T) :- state(on(B, C), T), goal(on(B, D)), C !=
    D.
```

Another way to optimize the runtime of the program is the implementation of domain specific knowledge with methods. The prerequisites are inherited from the operator. For perfomance reasons it is better to write them out, then to calculate the inheritance again each step.

```
% A typical method %
method(move).
method_var(move, (B, C)) :- block(B), block(C).
method_prerequisites(move, (B, C), T) :- state(needtomove(B), T),
    state(clear(B), T), state(clear(C), T).
currentTask(pickup, B, T) :- currentTask(move, (B, C), T).
currentTask(stack, (B, C), T+1) :- currentTask(move, (B, C), T).
```

4 Translation from SHOP2 to ASP

The main work of this project is a translation script from SHOP2 to Answer Set Programming. This translation should support comparability between the two systems. This chapter will show several problems in the translation of a SHOP2 problem into ASP and if they were solved in this project. Some of these problems are due to the SHOP2 syntax being optimized for HTN and their complexity complicates the translation. Other problems are based on structural differences between SHOP and ASP that cannot be translated at all, or at least not fully automatic.[3]

4.1 The Translation Script

The translation script works semi-automatically. This allows to partially solve the structural differences between HTN planning with SHOP and ASP, explained in the next section 4.2.

[3] A description of the SHOP2 syntax can be found in [NCLMA99, ch. 2.1] and on `http://www.cs.umd.edu/projects/shop/documentation.html`.

The code can be downloaded with a readme from `https://github.com/arksch/Shop2ASS`. The basic algorithm works as follows:

1. Parse atoms, initial states and goals from the input file according to the line numbers passed by the user
2. Clean the Lisp code: Strip off comments, strip search keywords, delete ASP operators like - and !, write out Lisp shorthands like *nil*, convert variables to capitals.
3. Add the general ASP solver and some documentary to the output file.
4. Search for axioms and operators by their keywords *:-* and *:operator*, parse them, translate them into ASP and write to the output file. For each operator a feedback from the user is asked, to specify the domain and implicit prerequisites.

Due to their complexity, the translation of SHOP-methods into ASP was not implemented.

4.2 Syntactic and Structural Translation Issues

A not complete list of syntactical sugar to SHOP that makes translation a nuisance:

- SHOP's if-then-else fashion for axioms and methods is complicating the translation. This means, that the head of an axiom of the form *(:- head tail1 tail2)* will be true if tail1 is true or tail1 is false and tail2 is true. For methods of the form *(:method head pre1 tail1 pre2 tail2)* it means, that tail1 will be applied if pre1 is true and tail2 will be applied if pre1 is false and pre2 is true.
- SHOP's omission of brackets in cases, when a collection of atoms has size 1. E. g. *(:- (place ?x) (depot ?x))* instead of *(:- (place ?x) ((depot ?x)))*
- SHOP's additional convenience methods: *forall*, *!!*, *:protection*, *:ordered*, *:unordered*, *:task*

Due to time reasons only the if-then-else fashion for axioms can currently be correctly translated into ASP.

The following list of structural differences cannot be automatically translated and each is solved individually by the translation script.

- The design of HTN planning with ASP explained in section 3.3 specifies domains of all atoms. Apparantely SHOP2 does specify the domain of atoms, but never uses them. The operators seem to infers their domain only from the added and deleted states. Appart from the superfluous information in the atoms' domains, another downside of the SHOP approach is the coupling of operators' and states' definitions. The translation script fixes this issue by requiring the user to add the domains of the operators.
- A SHOP problem description does not make a syntacic difference between domains and states. Problems deal with goals differently, e. g. with an *achieve-goals* method in the *Blocks Problem* or the *:task* keyword in the

Logistics Problem. This is solved by requiring the user to input the lines that describe atoms, initial states and goals.

- Logically, some operators should have prerequisites. E. g. the arm of the block problem robot being empty, when it wants to pick up a block. SHOP's syntax allows to include these prerequisites into the methods that call these operators. This seems to be a bad practice, but unless methods can be translated, these prerequisites have to be manually added while translation.

- SHOP's search keywords to optimize the search within single methods improve the speed, but cannot be translated to ASP. Precisely this would require to search a specific part of the search space with depth first search, complete search, depth first search for shallowest plan or iterative deepening search. The translation script deletes these keywords.

5 Evaluation of Correctness and Performance

An important task of the translation script was the comparison of correctness and performance between SHOP and ASP. Since the translation script does not work completely, this task cannot be tackled. As stated in section 3.1 the hand-tailoring of methods is crucial to the performance of the SHOP system. Section 4.2 explained the main differences between SHOP and ASP that cannot be translated. Especially the search commands will have a crucial impact on the performance of SHOP.

Without the translation of methods into ASP, its performance is out of scope. Solving a block problem with only 10 blocks takes a few seconds, whereas SHOP can solve a block problem of 100 blocks in a split second. Due to this immense difference in performance, this section focusses on theoretical aspects of the runtime behaviour.

As stated in section 3.2, the main advantage of ASP is its well developed search algorithm and conflict solution with CDCL. In contrast to this smartly implemented search, the optimization works by simply setting upper bounds. This is opposite to the SHOP approach and its problem description. Since SHOP always knows the complete world state, there is no action that can cause a conflict. The question is how SHOP is calculating the space of unconstrained possible next steps. Taking these steps, there should never be a conflict, since the system is only stupidly chosing non-conflicting methods to solve the given problem. It might run in circles, so the main point is the optimization by the design of special methods, encoding domain knowledge and special search commands. In the light of this architecture, it seems that the main advantage of ASP's CDCL approach does not hold. The SHOP2 system will never have to solve a conflict, but its main advantages are the fast forward-propagation through a tuned search space and a rather concise syntax to write down typical HTN problems.

References

[N+03] Dana Nau et al. Shop2: An htn planning system. *J. of AI Research*, 2003.

[NCLMA99] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munez-Avila. Shop: Simple hierarchical ordered planner. *IJCAI*, 1999.