AbTweakUser's Manual

Qiang Yang Steve Woods

Computer Science Department
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1

Acknowledgement

Support for the research is made available by NSERC operating grant #OGP0089686 to Qiang Yang.

Inquiries

Please send to :
Steven Woods woods@jupiter.drev.dnd.ca
Defense Research Establishment Valcartier
Courcelette, Quebec, Canada
G0A 1R0.
or
Qiang Yang qyang@logos.waterloo.edu
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1.

1 Introduction

AbTweak is a nonlinear, hierarchical planner. The planning language is the first order predicate logic without quantifiers and logical connectives. Given an initial state description and a goal state description, represented as sets of literals, it finds a partially ordered, least-commmtted set of operators to achieve the goal. It is correct in that every solution plan found necessarily solves the goal from the initial state, and complete in that it will terminate with a solution if one exists.

AbTweak is built on top of the planner Tweak (designed by Chapman). For reference of both planners, please see the papers by Chapman [1], by Qiang Yang and Josh Tenenberg [2], and the Master Thesis by Steve Woods [7].

The planner is designed by Qiang Yang and Josh Tenenberg, and implemented by Steve Woods and Qiang Yang at University of Waterloo. It runs on either Kyoto or Allegro Common Lisp.

An additional function has been added to Abtweak for testing and comparison. In Proceedings of AAAI91, McAllester and Rosenblitt proposed a version of nonlinear planning, called *Systematic Nonlinear Planning*. We have implemented a version of their planner, as well as its abstract version. The central theme of the new planner is to use establishment relation (or causal link) to protect everything achieved so far.

Below, we give a brief introduction to the basic components of the planner. Then we will describe how the planner is used in detail.

2 Components of Tweak

Tweakis implemented in four main independent sections: plan represetation, search, constraint management/plan maintenance, and modal truth criterion (MTC).

2.1 Plan and Operator Representation

Each plan is a *lisp structure* with the following slots:

1. plan-a: a set of operator structures.
2. plan-b: a set of pairs of operators, each pair represents a temporal order.
3. plan-nc: a set of pairs of variables and constants, each pair being a non-codesignation constraint.
4. plan-cr: a set of establishment relations to be protected.

It is assumed that a variable is anything of the form $*, where * can be any character or string. Any symbols not preceded by "$" is a constant. Unique name assumption applies to constants.

Each operator template is also a lisp structure. It has the following slots:

1. operator-opid: identifier for the operator;
2. operator-name: a s-expression as the name of the operator;
3. operator-cost: a cost value for the operator. This is static;
4. operator-preconditions: a list of precondition literals for the operator;
5. operator-effects: a list of effect literals for the operator;
6. (Optional): operator-primary-effects: a list of primary effects of the operator, used for selecting the operator to achieve a goal.

2.2 Search

The search routine is an implementation of the common A* graphsearch found in Nilsson's book [5]. Each node in the search space is a list, which consists of the following information:

1. the total cost (cost-so-far + heuristic) of the node;
2. the plan-structure;
3. the cost of the plan by summing up the costs of the operators;
4. the parent node of the current node.

The open list is a list of sublists, each containing a list of plans with the same costs. Functions for accessing this list is provided, which will be discussed below.

No attempt is made to check cycles in the search space, as this is still a difficult open problem. The user can specify various kinds of heuristics for selecting the next node in the open list. This is described in detail later.

2.3 Constraint Management

The constraint management portion of Tweak handles the insertion of operators, ordering constraints between pairs of operators as well as the codesignation and non-codesignation of variables and constants within the operators present in a plan.

Codesignations of two variables and of a variable and a constant are handled by simply replacing all variable occurences in the plan with a single value as codesignation occurs. For example, if a plan contained the variable $var, and a constraint was added such that $var was to codesignate with the constant 'block', then all occurences of $var in any plan operator, or other plan structure would be replaced by 'block'. Similarly, if two variables are made to codesignate, all occurences of the second are replaced by the first. Noncodesignation of $x and $y is represented as a pair $(\$x, \$y)$. The plan has a slot, plan-nc, which is a list of such pairs,

2.4 Modal Truth Criterion

The modal truth criterion (mtc) is a function, designed by Chapman, for checking whether or not a plan is correct. To do this, it takes $O(n^4)$, where n is the total number of operators in the plan.

In the search routine, (mtc plan) is a function for determining whether the current plan is a necessarily correct solution.

Note: white knights are *not* used in the Tweak implementation, as we don't believe that it offers computational advantage. It is trivial to prove the completeness of the planner without white knights.

3 How to Use the Planner

3.1 Defining your Domain

The first step in using the planner is to define a domain definition. For example, if we consider the 2 ring tower of hanoi domain, there can be two operators, defined as follows:

```
(setq op1 (create-operator-instance
:opid 'moveb
:name '(moveb $x $y)
:preconditions '((not ons $x) (onb $x) (not ons $y)
(ispeg $x) (ispeg $y) )
:effects '((not onb $x) (onb $y))))

(setq op2 (create-operator-instance
:opid 'moves
:name '(moves $x $y)
:preconditions '((ons $x) (ispeg $x) (ispeg $y) )
:effects '((not ons $x) (ons $y))))

(setq *operators* (list op1 op2))
```

Each operator has a name, including a list of parameters that the operator accepts, a list of preconditions specifying exactly the state the world must be in so as to allow this operator to be applied, and a list of effects specifying exactly what the results of applying this operator are. Negations are allowed, and are specified simply as (not predicate) where the positive would be (predicate). These operators are stored in a global variable *operators* so the planner can find and use them.

An initial state specification might look like:

```
(setq initial
      '((ispeg peg1) (ispeg peg2) (ispeg peg3) (onb peg1) (ons
      peg1) (not onb peg2) (not onb peg3) (not ons peg2) (not ons peg3)))
```

A goal specification for hanoi might look like:

```
(setq goal '((onb peg3) (ons peg3)))
```

3.2 Directory Structures

The file, plan.lsp, contains the source of the main routine. Through the key words set in this routine, one can invoke either AbTweak, or Tweak. AbTweak related routines are in the directory Ab-routines/. Tweakrelated routines are in Tw-routines/. A* search related routines are stored in Search-routines/. In addition, the sub-directory Plan-routines/ contains general routines connecting the planners.

3.3 How to Plan

The main command for planning appears below:

```
(defun plan (initial goal &key
                                (planner-mode 'tweak)
                                (continue-p nil)
                                (debug-mode nil)
                                (heuristic-mode 'num-of-unsat-goals)
                                (expand-bound 500)
                                (generate-bound 1000)
                                (open-bound 1000)
                                (output-file nil)
                                (cpu-sec-limit 60) ; 1 min
  ;AbTweak related keywords
                                (use-primary-effect-p nil)
                                (mp-mode t)
                                (abstract-goal-mode t)
                                (drp-mode nil)
                                (left-wedge-mode t)
                                (subgoal-determine-mode 'stack)
                                (solution-limit 100)
                                )
```

To run the planner, be sure one is located in the directory Abtweak/. Invoke lisp interpreter, and load "init.lsp".

If *initial* contains the initial state literals, and *goal* the goal state literals, then the following call invokes the planner tweak.

(plan initial goal :planner-mode 'tweak)

Upon termination, the total number of nodes expanded and generated are returned. Also, the solution plan is stored in a global variable *solution*.

Some important key words that are related to Tweak are briefly described below:

planner-mode Can be 'tweak, 'abtweak, 'mr, or 'mr-crit.

If 'abtweak is used, then be sure that a list of criticality assignments are given in *critical-list*. Also, if left-wedge-mode is chosen, then a list of level-preference should be given through *k-list*.

In addition, two other planners can be chosen with either *'mr* or 'mr-crit as the keyword value. Option 'mr invokes McAllester and Rosenblitt's systematic nonlinear planner, and mr-crit is an abstract version of mr.

continue-p once set, the next lowest cost solution is planned for, taking the existing open list as the current state space.

heuristic-mode With 'num-of-unsat-goals, the total number of unsatisfied goals are taken as a heuristic. However, if set to be 'user-defined, then the user defined heuristic function (explained below) is used to evaluate the heuristic.

expand-bound If the total number of nodes expanded exceeds this number, then the planner terminates.

generate-bound If the total number of nodes generated exceeds this number, then the planner terminates.

open-bound If the total number of nodes in the open list exceeds this number, then the planner terminates.

cpu-sec-limit If the total number of cpu time in seconds exceeds this number, then planner terminates.

debug-mode If t, the planner breaks every time a new node is generated. Both the parent and the new node are displayed for debugging purposes.

output-file If non-nil, the output is written into the filename specified. For example, :output-file "file1.data" specifies that the output of the planner should be written into file1.data.

solution-limit Maximum depth of the search tree, in terms of the total number of operators in a solution.

use-primary-effect-p T iff one wants to use primary effects for goal achievement. Before using this option, make sure that each operator in a domain has a set of primary effects specified, besides the set of effects. The primary effects are used for backwards chaining during goal achievement, but the effects are generally used for checking conflicts and interactions. An example where primary effects are used is given in Domains/simple-robot-1.lsp.

Other flags will be described in detail, in later versions of the manual.

3.3.1 Specifying Search Heuristic

The user can specify a user-defined heuristic function. This function value will be added to the cost-so-far, and used to select a node from the open list. To activate this, the key-word "heuristic-mode" has to be set true.

The function (user-heuristic) returns a function, which takes a plan as a parameter, and which evaluates to be a number. An example of one such function is as the following:

```
(defun user-heuristic ()
  '(lambda (plan) (num-of-unsat-goals plan)))
(defun num-of-unsat-goals (plan)
  (declare
    (type array plan) )
   (count-if-not
   #'(lambda (ith-goal)
     (hold-p plan 'g ith-goal))
   (get-preconditions-of-opid 'g plan)))
```

This function evaluates the total number of unsatisfied goals in the plan, and use that number as the heuristic value.

4 AbTweak

AbTweak combines the advantages of Tweak and Abstrips [6], by planning at multiple levels of abstraction. The abstraction hierarchy is specified via a criticality assignment to the literals in the domain. At each level i of abstraction, all preconditions of an operator which are lower than i are removed, resulting in a simpler operator set on that level. Planning is first done at the highest level of abstraction, using the operator set on the corresponding level. When a correct solution is found at level i of abstraction, it is *refined* to the $i-1$ level by adding back all preconditions of criticality $i-1$. It terminates when MTC returns True at the lowest level of abstraction.

For more information, please see [2].

4.1 Criticality Assignments

The criticality assignment to the literals is done through a global variable *critical-list*. Each element of the list is a sublist of the form:

```
(crit-value  (p $ $) (not q $))
```

That is, any literal with a predicate p or $\neg q$, with whatever argument(s), will have a criticality value = *crit-value.*

For the Tower of Hanoi domain, this is specified as follows:

```
(setq *critical-list-1* '(
   (3 (ispeg $))
   (2  (not onb $) (onb $) )
   (1  (not onm $) (onm $) )
   (0  (not ons $) (ons $) )
)
```

That is, $(ispeg\$)$ literals has the highest criticality in this assignment, while the $(ons\$)$ ones the lowest. The global variable *critical-list* is set in a domain file.

4.2 Left-Wedge Control Strategy

Our experience has been that, for most "good" abstraction hierarchies and most problems, the first abstract solution can be refined successfully to a lowest level solution. Although not always true, this observation can be taken as a heuristic that leads to a heavier invest in computation time to the first abstract solution than the rest. The result, is a complete search control method, called the *left-wedge.*

To make it possible to execute left-wedge, a list of cost-adjustment values have to be provided. Let the i^{th} value of the list be v_i, then the cost of a plan at $k-i$th level abstraction is adjusted by $cost(plan) - v_i$. This list of values is specified through a global variable *left-wedge-list*.

As an example, the left wedge list for Tower of Hanoi domain can be specified as follows:

```
(setq *left-wedge-list* '(0 1 3 7))
```

4.3 The Downward Refinement Property

Some abstraction hierarchy has the property that, every abstraction solution has a refinement down to the lowest level of abstraction. In that case, one does not have to keep all abstraction solutions in the OPEN list. As soon as one abstract solution is found to be correct, the rest can be saved on a stack, *drp-stack*. This can often lead to an exponential amount of saving in computation [3].

If :drp-mode is on, then in the abstract direction (vertical), a depth first search is done. Backtracking to the previous level occur only when the current level search fails.

Flag :drp-debug-mode will enter a break loop whenver an abstract refinement of a solution is found at an abstract level. The user can then inspect an abstract solution, before it is passed down to the next level for refinement. In Allegro Common Lisp, type :cont to resume computation.

4.4 The Monotonic Property

The monotonic property is a formalization of the idea of *protection intervals* in classical hierarchical planning. Once a solution is found at level i, all causal relations for the preconditions above $i-1$ is computated, and protected during planning at subsequent levels. This heuristic has been shown to be complete [2].

4.5 The Abstract Planner

Now we describe how to invoke the abstract planner.

1. To call AbTweak, type

   (plan initial goal :planner-mode 'abtweak)

2. To use left-wedge strategy, type

   (plan initial goal :planner-mode 'abtweak :left-wedge-mode t)

3. To use the monotonic property, type

   (plan initial goal :planner-mode 'abtweak :mp-mode t)

4. To use the downward refinement property, type

   (plan initial goal :planner-mode 'abtweak :drp-mode t)

5 Planner Mode

A global variable *planner-mode* determines the planner routines that are run, Tweak, Abtweak, MR, or MR-crit.

6 Sample Domains

Several domains are listed in /Abtweak/Domains/. They include a simple blocksworld domain, a Nilsson's blocksworld domain, a Towers of Hanoi domain, a register's content exchange domain, a simple robot planning domain, a simple transportation domain, a simple computer hardware domain, a simple biology domain, a simple database query optimization domain, and a simple natural language style generation domain.

The two robot planning domains are adapted from Sacerdoti's Abstrips domain, where a robot can move a number of boxes around several rooms, and the rooms have doors that can be opened or closed. For a picture of the domain, consult Sacerdoti's paper on Abstrips, AIJ. To reduce search, we have implemented a user-defined heuristic that checks cycles in this domain. The file that contains this heuristic is in /Domains/robot-heuristic.lsp. This heuristic can be used if (a) the file is loaded into lisp interpreter, and (b) the option

(plan initial goal :heuristic-mode 'user-defined ...)

is used.

7 Other Useful Routines

Show Plan (show-ops plan-structure)

input: plan structure

output: a pretty print structure of the plan.

Statistics So Far (cur)

Outputs the current statistics including the number of nodes expanded and generated.

Outputs a list of initial states, goal states, and operator templates.

Length of the Open List (length-of-open)

outputs the length of the open list.

First and Last Elements In The Open List (first-of-open) or (last-of-open)

outputs the first or the last plan structure on the plan list.

References

[1] D. Chapman, "Planning for Conjunctive Goals", *Artificial Intelligence*, (32), pages 333-377, 1987.

[2] Q. Yang and J. Tenenberg, "ABTWEAK: Abstracting a Nonlinear, Least Commitment Planner", Proceedings of the Eighth National Conference on Artificial Intelligence, pages 199-204, 1990.

[3] F. Bacchus and Q. Yang, "The Downward Refinement Property," Proceedings of the IJCAI 91.

[4] McAllester and Rosenblitt, *Systematic Nonlinear Planning*, Proceedings of the AAAI91.

[5] N. J. Nilsson, *Principles of Artificial Intelligence*, Morgan Kaufmann Publishers, 1980.

[6] Sacerdoti, Earl 1974. *Planning in a hierarchy of abstraction spaces. Artificial Intelligence* 5:115–135.

[7] Steve Woods, Master Thesis: *An Implementation and Evaluation of A Nonlinear, Hierarchical Planner.* Department of Computer Science, University of Waterloo, 1991.

[8] Qiang Yang, Josh D. Tenenberg and Steven Woods, "Abstraction in Nonlinear Planning," Unpublished journal length paper. Available as University of Waterloo technical report CS-91-65. 52 pages. A postscript version can be obtained via anonymous ftp to "cs-archive.uwaterloo.ca"