SHOP: Simple Hierarchical Ordered Planner

Dana Nau  Yue Cao  Amnon Lotem  Héctor Muñoz-Avila

Department of Computer Science, and Institute for Systems Research
University of Maryland, College Park, MD 20742
U.S.A.

Abstract

SHOP (Simple Hierarchical Ordered Planner) is a domain-independent HTN planning system with the following characteristics.

- SHOP plans for tasks in the same order that they will later be executed. This avoids some goal-interaction issues that arise in other HTN planners, so that the planning algorithm is relatively simple.
- Since SHOP knows the complete world-state at each step of the planning process, it can use highly expressive domain representations. For example, it can do planning problems that require complex numeric computations.
- In our tests, SHOP was several orders of magnitude faster than Blackbox and several times faster than TLplan, even though SHOP is coded in Lisp and the other planners are coded in C.

1 Introduction

“Conventional wisdom” in AI planning holds that total-order forward search is a bad idea because it causes excessive backtracking. However, several groups of researchers have begun to argue that the opposite is true: that total-order forward-search allows planners to use a more expressive domain representations, which can be used to encode domain knowledge to make the planners highly efficient. More specifically:

- Prodigy [Veloso and Blythe, 1994; Fink and Veloso, 1995] does a forward state-space search that is guided by a means-end analysis made by backward chaining on the goals. Veloso and Blythe [1994] showed that causal link commitments can affect the performance of partial-order planners when the goals have a property called linkability. In their experiments, Prodigy ran many times faster than SNLP [McAllester et al., 1991].
- Smith et al. [1997, 1998] developed an approach that combines HTN-style problem reduction with left-to-right backtracking to produce a search strategy similar to Prolog’s. They used this approach successfully in domain-specific planners for several practical applications, including manufacturing planning [Smith et al., 1997] and the game of bridge [Smith et al., 1998]. They argued for the advantages of their approach by analyzing the reasons for its success in real-world applications [Nau et al. 1998]. However, they could not compare their approach head-to-head against domain-independent planning algorithms, because their implementations were domain-specific.

In order to test the performance of Smith et al.’s approach in a domain-independent setting, we have created a domain-independent formalization of the approach, and have implemented it in a planner called SHOP (Simple Hierarchical Ordered Planner). SHOP is available at <http://www.cs.umd.edu/projects/shop>, under the terms of the GNU General Public License. SHOP has the following characteristics:

1. SHOP plans for tasks in the same order that they will be executed. By avoiding some task-interaction issues, this makes SHOP simpler than HTN planners such as such as NONLIN [Tate, 1977], SIPE-2 [Wilkins, 1990], O-PLAN [Currie and Tate, 1991], and UMCP [Erol et al., 1994]. It also makes it easier to prove soundness and completeness results.

2. Since SHOP always knows the complete world-state at each step of the planning process, it can use considerably more expressivity in its domain representations than most AI planners. For example, SHOP has the ability to do Horn-clause inferencing, numeric computations, and interactions with external agents and external information sources.

3. SHOP’s expressive power can be used to create highly efficient domain representations. In our tests on blocks-world and logistics problems, SHOP was several orders of magnitude faster than Blackbox and several times faster than TLplan, even though SHOP
is coded in Lisp and the other planners are in C.

2 Formal Definitions

This section defines the syntax and semantics used in SHOP, as well as the SHOP planning algorithm. For brevity, the definitions below are for a somewhat simplified version of SHOP’s syntax and semantics. Section 3 gives an informal overview of the additional features that appear in the full syntax and semantics. For a formal description of those features, see <www.cs.umd.edu/projects/shop/documentation.html>.

2.1 Syntax

We use the usual first-order-logic definitions of variable and constant symbols, function and predicate symbols, terms, atoms, conjunctions, most-general unifiers (mgu’s), and Horn clauses; with the notation adapted for Lisp. For example, here are two Horn clauses, first in Prolog notation and then in our notation:

\[
p(f(X)) :- q(X,c), r(Y,d), s(d)
\]

\[
q(b,c).
\]

\[
(\neg (p(f ?x)) \land (q ?x c) \land (r ?y d) \land (s d)))
\]

\[
(\neg (q b c nil))
\]

A state is a set of ground atoms, and an axiom set is a set of Horn clauses. If S and is a state and X is an axiom set, then \(\text{S} \cup \text{X}\) satisfies a conjunct C if there is a substitution \(u\) (called a satisfier) such that \(\text{S} \cup \text{X}\) entails \(C\). \(u\) is a most general satisfier (or mgu) if there is no other satisfier \(v\) more general than \(u\). In contrast to mgu’s (which are unique modulo lexical renaming), there may be several distinct mgu’s for \(C\) from \(S\) and \(X\).

A task is a list of the form \((s \ t_1 \ t_2 \ \ldots \ t_n)\), where \(s\) (the task’s name) is a task symbol, and \(t_1, t_2, \ldots, t_n\) (the task’s arguments) are terms. The task is primitive if \(s\) is a primitive task symbol (a symbol whose first character is an exclamation point) and it is compound if \(s\) is a compound task symbol (a symbol whose first character is not a special character). A task list is a list of tasks.

An operator is a expression \((:\text{operator} \ h \ \text{D} \ \text{A})\), where \(h\) (the head) is a primitive task, and \(D\) and \(A\) (the deletions and additions) are sets of atoms containing no variable symbols other than those in \(h\). For example, here is an operator to put a block on the table:

\[
(:\text{operator} (:\text{putdown} \ ?\text{block}) (:\text{holding} \ ?\text{block})) (:\text{ontable} \ ?\text{block}) \text{ handempty}))
\]

A method is an expression that has the form \((:\text{method} \ h \ \text{C} \ \text{T})\), where \(h\) (the method’s head) is a compound task, \(C\) (the method’s precondition) is a conjunct, and \(T\) (the method’s tail) is a task list. For example, here is a pair of methods for clearing the top of a block:

\[
(:\text{method} (\text{make-clear} \ ?y)) \text{ nil}
\]

\[
(\text{clear} \ ?y)) \text{ nil})
\]

\[
(:\text{method} (\text{make-clear} \ ?y)
\]

\[
\text{ (on} \ ?x \ ?y)\text{)}
\]

\[
(\text{make-clear} \ ?x)
\]

\[
\text{ (unstack} \ ?x \ ?y) \text{ (putdown} \ ?x))
\]

The first method says that if \(y\) is already clear we should do nothing; the second says that if another block \(x\) is on \(y\), we should make \(x\) clear and then move \(x\) to the table.

2.2 Semantics

The intent of an operator \(o = (:\text{operator} \ h \ \text{D} \ \text{A} \ c)\) is to specify that \(h\) can be accomplished by modifying the current state of the world to remove every atom in \(D\) and add every atom in \(A\). More specifically, if \(t\) is a primitive task and there is an mgu \(u\) for \(t\) and \(h\) such that \(h\) is ground, then \(o\) is applicable to \(t\), and the list \((h^u)\) is a simple plan for \(t\). If we execute this plan in some state \(S\), it produces the state \(h^u(S) = o^u(S) = (S - D^u) \cup D^u\).

The intent of a method \(m = (:\text{method} \ h \ C \ T)\) is to specify that if the current state of the world satisfies \(C\), then \(h\) can be accomplished by performing the tasks in \(T\) in the order given. More specifically, let \(S\) be a state, \(X\) be an axiom set, and \(r\) be a task atom. Suppose there is an mgu \(u\) that unifies \(t\) with \(h\), and suppose \(S \cup X\) satisfies \(C^u\). Then \(m\) is applicable to \(t\) in \(S \cup X\), and the result of applying \(m\) to \(t\) is the set of task lists \(R = \{(T^v) : \text{v is an mg} \text{s for } C^u \text{ from } S\}\). Each task list \(r\) in \(R\) is a simple reduction of \(t\) by \(m\) in \(S \cup X\).

A plan is a list of heads of ground operator instances. If \(p\) is a plan and \(S\) is a state, then \(p(S)\) is the state produced by starting with \(S\) and executing the operator instances in the order that their heads appear in \(p\).

A planning problem is a tuple \(P = (S,T,D)\), where \(S\) is a state, \(T\) is a task list, and \(D\) is a set of axioms, operators, and methods. If \((S,T,D)\) is a planning problem, then \(\Pi(S,T,D)\), the set of all plans for \(T\) from \(S\) in \(D\), is defined recursively as follows.

If \(T\) is empty, then \(\Pi(S,T,D)\) contains exactly one plan, namely the empty plan. Otherwise, let \(t\) be the first task atom in \(T\), and \(R\) be the remaining task atoms. There are three cases. (1) If \(t\) is primitive and there is a simple plan \(p\) for \(t\), then \(\Pi(S,T,D) = \{\text{append}(p,q) : q \in \Pi(p(S),R,D)\}\). (2) If \(t\) is primitive and there is no simple plan for \(t\), then \(\Pi(S,T,D) = \emptyset\). (3) If \(t\) is compound, then \(\Pi(S,T,D) = \cup\{\Pi(S,\text{append}(r,R),D) : r \text{ is a simple reduction of } t\}\).

2.3 Soundness and Completeness

The SHOP planning procedure is as follows:

**procedure** find-plan(S,T,D)

end find-plan

**procedure** seek-plan(S,T,D,p)

end seek-plan

**procedure** seek-plan(S,T,D,p)

if \(T = \text{nil}\) return \(\text{the list}(p)\)

\(t\) = the first task in \(T\); \(R\) = the remaining tasks

if \(t\) is primitive then

if there is a simple plan \(q\) for \(t\) then

return \(\text{seek-plan}(q(S),R,D,\text{append}(p,q))\)

else return \text{FAIL}

else

for every simple reduction \(r\) for \(t\) in \(S\)

ans = \text{seek-plan}(S, \text{append}(r,R),D,p)

if \(ans \neq \text{FAIL}\) then return \(ans\)
Since \textit{find-plan} is a straightforward implementation of the definition of \Pi S(T,D), it is easy to show it is sound. For finite search spaces, \textit{find-plan} is also complete. For infinite search spaces, it is incomplete for the same reason Prolog is incomplete: if the leftmost unexplored path is infinite, it will never return from that path. It is straightforward to make \textit{find-plan} complete for infinite search spaces, by doing an iterative-deepening search of \textit{find-plan}'s search space. Our implementation can do iterative deepening (at the user’s option), but in practice we have found it more efficient not to use it.

3 Example Planning Domain

To illustrate how SHOP works, we now describe a simple transportation-planning domain. Table 1 defines the domain, Table 2 shows a specific problem in that domain, and Table 3 shows plans found by SHOP on several problems in that domain.

The scenario for the domain is that we want to travel from one location to another in a city. There are three possible modes of transportation: taxi, bus, and foot. Taxi travel involves hailing the taxi, riding to the destination, and paying the driver $1.50 plus $1.00 for each mile traveled. Bus travel involves hailing the bus, paying the driver $1.00, and riding to the destination. Foot travel just involves walking, but the maximum feasible walking distance depends on the weather. Thus, different plans are possible depending on what the layout of the city is, where we start, where we want to go, how much money we have, and what the weather is like.

As mentioned earlier, SHOP incorporates several extensions to the syntax and semantics described in this paper. To illustrate those extensions, the transportation-planning domain uses most of them. In particular:

1. Axioms’ tails and methods’ preconditions can include negations (which are evaluated using the closed-world assumption) and calls to the Lisp evaluator. For example, Axiom A1 of Table 1 says that the taxi fare is $1.50 plus $1 if more than $1, and Method M1’s precondition says that to pay the driver, we need sufficient money for the fare.

2. If a method’s precondition is satisfied, then its entire tail is passed to the Lisp evaluator. Lisp’s \texttt{quote}, \texttt{backquote}, and \texttt{comma} constructs can be used to prevent evaluation (see Method M2) or to do conditional evaluation (see Method M1, which does subtraction to create \texttt{set-cash}’s second argument).

3. Axioms can have multiple tails, to be used in an “if-then-else” fashion. For example, the axiom \texttt{?(:- \texttt{head} tail1 \texttt{tail2} \texttt{tail3})} says \texttt{head} is true if \texttt{tail1} is true, or if \texttt{tail2} is false but \texttt{tail1} is true, or if \texttt{tail1} and \texttt{tail2} are false but \texttt{tail3} is true. This gives expressivity similar to a restricted version of Prolog’s “cut,” but in a way that is easier to understand. For example, Axiom A2 says that walking distance is ≤ 3 miles in good weather, and ≤ 1 mile otherwise.
Table 3: Plans produced by SHOP on problems in the planning domain of Table 1. In each problem, the distances and bus routes are the same as in lines 4–11 of Table 2. In each problem, SHOP found all possible plans in less than 0.01 seconds.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Plan(s) found by SHOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go to park, good weather, no cash</td>
<td>([][WALK DOWNTOWN PARK])</td>
</tr>
<tr>
<td>Go to park, bad weather, no cash</td>
<td>None (can’t afford a taxi or bus, and it’s too far to walk).</td>
</tr>
<tr>
<td>Go to park, good weather, have $12</td>
<td>1 ([][WALK DOWNTOWN PARK])</td>
</tr>
<tr>
<td></td>
<td>2 ([][HAIL TAXI DOWNTOWN])</td>
</tr>
<tr>
<td></td>
<td>([][SET-CASH 12 8.5])</td>
</tr>
<tr>
<td>Go to park, good weather, have $80</td>
<td>1 ([][WALK DOWNTOWN PARK])</td>
</tr>
<tr>
<td></td>
<td>2 ([][HAIL TAXI DOWNTOWN])</td>
</tr>
<tr>
<td></td>
<td>([][SET-CASH 80 76.5])</td>
</tr>
<tr>
<td>Go uptown, good weather, have $12</td>
<td>([][HAIL TAXI DOWNTOWN])</td>
</tr>
<tr>
<td></td>
<td>([][SET-CASH 12 2.5])</td>
</tr>
<tr>
<td>Go uptown, good weather, have $80</td>
<td>([][HAIL TAXI DOWNTOWN])</td>
</tr>
<tr>
<td></td>
<td>([][SET-CASH 80 70.5])</td>
</tr>
<tr>
<td>Go to suburb, good weather, have $12</td>
<td>([][WAIT-FOR BUS3 DOWNTOWN])</td>
</tr>
<tr>
<td></td>
<td>([][SET-CASH 12 11.0])</td>
</tr>
<tr>
<td></td>
<td>([][HAIL-TAXI DOWNTOWN SUBURB])</td>
</tr>
<tr>
<td></td>
<td>([][SET-CASH 80 66.5])</td>
</tr>
</tbody>
</table>

4. If the first element of a method’s precondition or an axiom’s tail is :first, SHOP’s theorem prover returns after finding the first satisfy (just as Prolog would do), rather than looking for all satisifiers. As an example, in Method M3 this is used to tell SHOP that it should only consider hailing the first taxi at the taxi stand, rather than hailing all of them.

5. A method can have multiple pairs of preconditions and tails, to be used in an “if-then-else” fashion. For example, “(:method head pre1 tail1 pre2 tail2)” says that the reduction of head is tail1 if pre1 is true, or tail2 if pre1 is false and pre2 is true. Method M3 uses this to specify that we won’t consider bus travel unless we don’t have enough money for taxi travel.

6. Operators have numeric costs (the default cost is 1), and the cost of a plan is the sum of its operator costs. The transportation domain does not illustrate this.

Although the transportation-planning domain is easy to represent in SHOP, we believe that most other AI planners would not have sufficient expressive power to represent it fully, because of the numeric computations that need to be done as part of the planning process.

### 4 Experiments

We have tested SHOP against two other planners: Blackbox [Kautz and Selman, 1998], which was one of the two fastest planners in the AIPS-98 planning competition [McDermott, 1998]; and TLplan [Bacchus and Kabanza, 1998], which outperformed Blackbox by several orders of magnitude in Bacchus and Kabanza’s tests.

#### 4.1 Blocks-World Planning

To run SHOP in the blocks world, we encoded the blocks-world planning algorithm of [Gupta and Nau, 1992] as a set of axioms, operators, and methods. We tested SHOP, TLplan and Blackbox on the blocks-world problems in the Blackbox software distribution. We ran SHOP and TLplan on a 167-MHz Sun Ultra, and Blackbox on a 143-MHz Sun Ultra. Both machines had 64 MB of RAM. The results are shown in Table 4.

Blackbox did worst: its time requirements increased far more quickly with problem size than SHOP’s and TLplan’s. This was to be expected, because SHOP and TLplan are guaranteed to run in low-order polynomial time on blocks-world problems, whereas Blackbox does an exponential-time search. Blackbox could not solve the two largest problems at all, because it ran out of memory.

On the larger problems, TLplan took more time than SHOP, and found longer plans. We should run more tests to establish statistical significance, but the results clearly are algorithmically significant: TLplan found some non-optimal plans that the blocks-world algorithm that encoded into SHOP’s methods and operators [Gupta and Nau, 1992] had been designed to avoid.

#### 4.2 Logistics Problems

To run SHOP in the logistics domain, we encoded the following procedure into methods, operators, and axioms.

- First remove from the current world-state all “useless objects” that will not contribute to the plan. These include packages not mentioned in the goal, and empty trucks and airplanes in the same city with other trucks and airplanes. Then do the following steps repeatedly until every package is at its final destination:
  1. If there is a truck or airplane at the same location as some packages that need to be picked up or dropped off, then pick them up or drop them off.
that need to go to the airport, and load them onto the airplane. Then fly the airplane to city c.

3. Else if there is an airplane with at least one package on board, then fly it to the destination of one of the packages on board.

4. Else if there are one or more packages that need to be picked up, then drive a truck to the location of any one of them.

5. Else if there is a truck that is carrying one or more packages, then drive it to the final destination of one of the packages in the truck.

We ran SHOP and TLplan on logistics problems in the Blackbox and TLplan distributions, on a 167-MHz Sun Ultra with 64 MB of RAM. Because of Blackbox’s memory requirements, we did not run it ourselves. Instead, we used published data for Blackbox on a machine that is faster than ours and has 8 GB of RAM.5 Tables 5 and 6 show the results.

Again Blackbox did worst and SHOP did best. Blackbox was several orders of magnitude slower than both SHOP and TLplan, and it found significantly larger plans. SHOP and TLplan found plans of comparable size, but on most of the problems SHOP ran several times faster than TLplan (more than an order of magnitude faster on the more difficult problems).

5 Discussion and Conclusions

It did not surprise us that SHOP did so much better than Blackbox, for SHOP’s methods and axioms contained sophisticated domain knowledge that could not be represented in Blackbox’s operators. However, it did surprise that SHOP did so much better than TLplan. Here, we think, are the primary reasons why it did so:

1. Although TLplan’s modal-logic representation capabilities are quite sophisticated, their use (at least in the examples we have seen) has been limited to writing pruning heuristics rather than actual planning algorithms. SHOP’s use of HTN methods makes it easy to write efficient planning algorithms, as we did for both the blocks world and the logistics domain.

2. TLplan’s planning algorithm is basically a state-space search, whereas SHOP uses HTN-style problem reduction. Problem reduction can be much more efficient than state-space search (even by an exponential amount in some cases [Korf, 1987; Yang et al., 1992]).

Our results support the contention that total-order forward search, combined with HTN-style problem reduction, can “scale up” to complex planning problems better than partial-order action-based planning. Our

---

5 We got the Blackbox performance data from Table 11 of [Bacchus and Kabanza 1998]. According to Faheem Bacchus, the data came originally from the Blackbox distribution, and the machine was a Silicon Graphics with 8 GB of RAM, running at around 200 MHz.
results also illustrate the impact that planning applications can have on planning theory: SHOP is a domain-independent formalization and implementation that evolved from our previous domain-specific work on manufacturing planning and computer bridge.

Our ongoing and future work is as follows:

- We are doing additional experiments and analyses in order to get a better understanding of the efficiency issues discussed above.
- SHOP appears to be powerful enough to be of use in complex applications such as noncombatant evacuation operation planning [Muñoz-Avila et al., 1999]. To make it easier to embed SHOP in such applications, we are creating an implementation of SHOP in Java.
- It is straightforward to prove soundness and completeness using the definitions in Section 2, but it is more difficult to prove soundness and completeness in the presence of some of the extensions discussed in Section 3 (such as the calls to the Lisp evaluator). We have begun working with others who have experience in these issues, to put this aspect of SHOP on a more solid formal footing.
- We are developing a general way to handle some partial-order-planning operations while preserving SHOP’s expressivity and left-to-right control strategy. We intend to describe this in a forthcoming paper.

Acknowledgements

This work 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


