Intelligent Agents
State-Space Planning

Ute Schmid
Cognitive Systems, Applied Computer Science, Bamberg University

last change: 28. Mai 2014
Motivation

- Nearly all planning procedures are search procedures
- Different planning procedures have different search spaces
  - Two examples:
    - State-space planning
      - Each node represents a state of the world
      - A plan is a path through the space
    - Plan-space planning
      - Each node is a set of partially-instantiated operators, plus some constraints
      - Impose more and more constraints, until we get a plan
Motivation

State-Space-Planning

\[ s_0 \rightarrow a_1 \rightarrow s_1 \rightarrow a_2 \rightarrow s_2 \rightarrow a_3 \rightarrow s_3 \rightarrow a_4 \rightarrow s_4 \rightarrow a_5 \rightarrow s_5 \rightarrow \ldots \rightarrow s_g \]

Plan-Space-Planning

U. Schmid (CogSys)

Intelligent Agents

last change: 28. Mai 2014
Outlook

- Forward Search
- Backward Search
  - Inverse State Transition
  - Lifting
- Soundness, Completeness, Efficiency
- Strips
- Incompleteness of Linear Planning
  - Sussman Anomaly
- Domain Specific Knowledge
**Algorithm 1 Forward-search**($O, s_0, g$)

1. $s \leftarrow s_0$
2. $\pi \leftarrow$ the empty plan
3. **loop**
   1. if $s$ satisfies $g$ then return $\pi$
   2. $E \leftarrow \{a|a$ is a ground instance of an operator in $O$, and precond($a$) is true in $s\}$
   3. if $E = \emptyset$ then return failure
   4. non-deterministically choose any action $a \in E$
   5. $s \leftarrow \gamma(s, a)$
   6. $\pi \leftarrow \pi.a$
4. **end loop**
Properties

- Forward-search is *sound*
  - for any plan returned by any of its non-deterministic traces, this plan is guaranteed to be a solution

- Forward-search also is *complete*
  - if a solution exists then at least one of Forward-search’s non-deterministic traces will return a solution.

Dana Nau: Lecture slides for *Automated Planning*
Licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License:
http://creativecommons.org/licenses/by-nc-sa/2.0/
Some deterministic implementations of forward search:
- breadth-first search
- depth-first search
- best-first search (e.g., A*)
- greedy search

Breadth-first and best-first search are sound and complete
- But they usually aren’t practical because they require too much memory
- Memory requirement is exponential in the length of the solution

In practice, more likely to use depth-first search or greedy search
- Worst-case memory requirement is linear in the length of the solution
- In general, sound but not complete
  \[ \Rightarrow \] But classical planning has only finitely many states
  \[ \Rightarrow \] Thus, can make depth-first search complete by doing loop-checking
Forward search can have a very large branching factor
  E.g., many applicable actions that don’t progress toward goal

Why this is bad:
  Deterministic implementations can waste time trying lots of irrelevant actions

Need a good heuristic function and/or pruning procedure
  See Section 4.5 (Domain-Specific State-Space Planning)


and Lecture on Heuristic Search Planning
Backward Search

- For forward search, we started at the initial state and computed state transitions:
  - new state $= \gamma(s, a)$

- For backward search, we start at the goal and compute inverse state transitions:
  - new set of sub-goals $= \gamma^{-1}(g, a)$

- To define $\gamma^{-1}(g, a)$, must first define *relevance*:
  - An action $a$ is relevant for a goal $g$ if
    - $a$ makes at least one of $g$'s literals true
      $\iff g \cap \text{effects}(a) \neq \emptyset$
    - $a$ does not make any of $g$'s literals false
      $\iff g^+ \cap \text{effects}^-(a) \neq \emptyset$ and $g^- \cap \text{effects}^+(a) = \emptyset$
Inverse State Transitions

- If \( a \) is relevant for \( g \), then
  \[
  \gamma^{-1}(g, a) = (g - \text{effects}(a)) \cup \text{precond}(a)
  \]
- Otherwise \( \gamma^{-1}(g, a) \) is undefined

Example: suppose that
\[
\begin{align*}
g &= \{\text{on}(b1,b2), \text{on}(b2,b3)\} \\
a &= \text{stack}(b1,b2)
\end{align*}
\]
What is \( \gamma^{-1}(g, a) \)?
**Algorithm 2 Backward-search** $(O, s_0, g)$

\[
\pi \leftarrow \text{the empty plan}
\]

\[
\text{loop}
\]

- if $s_0$ satisfies $g$ then return $\pi$
- $A \leftarrow \{ a | a \text{ is a ground instance of an operator in } O \text{ and } \gamma^{-1}(g, a) \text{ is defined} \}$
- if $A = \emptyset$ then return failure
- non-deterministically choose any action $a \in A$
  \[
  \pi \leftarrow a . \pi
  \]
  \[
  g \leftarrow \gamma^{-1}(g, a)
  \]

end loop
Efficiency of Backward Search

- Backward search can also have a very large branching factor.
  - E.g., an operator $o$ that is relevant for $g$ may have many ground instances $a_1, a_2, \ldots, a_n$ such that each $a_i$’s input state might be unreachable from the initial state.

- As before, deterministic implementations can waste lots of time trying all of them.
Remarks on Backward Planning

- Forward search also called \textit{progression} planning
- Backwards search also called \textit{regression} planning
- Problem with backwards planning: inconsistent states can be produced (see blocksworld example)
- Compare Graphplan strategy: build a Planning Graph by forwards search (polynomial effort) and extract the plan from the graph backwards (exponential effort, as usual for planning)
Backward Planning cont.

Axiom: $\forall x, y \ on(x, y) \rightarrow \neg clear(y)$
Lifting

Can reduce the branching factor of backward search if we partially instantiate the operators

- this is called lifting

foo(x, y)
precond: p(x, y)
effects: q(x)
Lifted Backward Search

- More complicated than Backward-search
  - Have to keep track of what substitutions were performed
- But it has a much smaller branching factor
- \( \text{mgu} = \text{most general unifier} \) (see later), e.g. for \( \text{foo}(x, y) \), substitution \( \theta = \{ x \leftarrow a_1 \} \) results in equality between all effects of \( \text{foo}(a_1, y) \) and goal \( q(a_1) \)

**Algorithm 3** Lifted-backward-search\((O, s_0, g)\)

\[
\pi \leftarrow \text{the empty plan}
\]

\[
\text{loop}
\]

\[
\text{if } s_0 \text{ satisfies } g \text{ then return } \pi
\]

\[
A \leftarrow \{(o, \theta) | o \text{ is a standardization of an operator in } O, \theta \text{ is an mgu for an atom of } g \text{ and an atom of } \text{effects}^+(o), \text{ and } \gamma^{-1}(\theta(g), \theta(o)) \text{ is defined } \}
\]

\[
\text{if } A = \emptyset \text{ then return failure}
\]

\[
\text{non-deterministically choose a pair } (o, \theta) \in A
\]

\[
\pi \leftarrow \text{the concatenation of } \theta(o) \text{ and } \theta(\pi)
\]

\[
g \leftarrow \gamma^{-1}(\theta(g), \theta(o))
\]

\[
\text{end loop}
\]
Lifted-backward-search generates a smaller search space than Backward-search, but it still can be quite large.

- Suppose actions $a$, $b$, and $c$ are independent, action $d$ must precede all of them, and there’s no path from $s_0$ to $d$’s input state.
- We’ll try all possible orderings of $a$, $b$, and $c$ before realizing there is no solution.
- More about this in Chapter 5 (Plan-Space Planning)

STRIPS

- $\pi \leftarrow$ the empty plan
- do a modified backward search from $g$
  - instead of $\gamma^{-1}(s, a)$, each new set of sub-goals is just precond($a$)
  - whenever you find an action that’s executable in the current state, then go forward on the current search path as far as possible, executing actions and appending them to $\pi$
  - repeat until all goals are satisfied

\[
\pi = \langle a_6, a_4 \rangle \\
\gamma^{-1}(s, a) = \gamma(\gamma(s_0, a_6), a_4)
\]
STRIPS

by Fikes & Nilsson (1971),

“Stanford Research Institute Problem Solver”

classical example:
moving boxes between rooms ("Strips World")

Originally:
representation formalism (relying on CWA) and planning algorithm
today:
“STRIPS planning” refers to classical representation without extensions and not to a specific algorithm

STRIPS algorithm:
a linear (and therefore incomplete) approach

compare to:
General Problem Solver (GPS), a cognitively motivated problem solving algorithm which is also linear and therefore incomplete
STRIPS Algorithm

- Backward-search with a kind of hill climbing strategy
- In each recursive call only such sub-goals are relevant which are preconditions of the last operator added
- Consequence:
  considerable reduction of branching, but resulting in incompleteness
- Linear planning:
  organizing sub-goals in a stack
- Non-linear planning:
  organizing sub-goals in a set, interleaving of goals
Algorithm 4 STRIPS (O, s, g)

\[ \pi \leftarrow \text{empty plan} \]

\[ \text{loop} \]

\[ \text{if } s \text{ satisfies } g \text{ then} \]

\[ \text{return } \pi \]

\[ \text{end if} \]

\[ A \leftarrow \{ a | a \text{ is a ground instance of an operator in } O, \text{ and } a \text{ is relevant for } g \} \]

\[ \text{if } A = \emptyset \text{ then} \]

\[ \text{return failure} \]

\[ \text{end if} \]

\[ \text{non-deterministically choose any action } a \in A \]

\[ \pi' \leftarrow \text{STRIPS}(O, s, \text{precond}(a)) \]

\[ \text{if } \pi' = \text{failure} \text{ then} \]

\[ \text{return failure} \]

\[ \text{else} \]

\[ s \leftarrow \gamma(s, \pi') \]

\[ s \leftarrow \gamma(s, a) \]

\[ \pi \leftarrow \pi.\pi'.a \]

\[ \text{end loop} \]
Incompleteness of Linear Planning

The Sussman Anomaly

Initial State

Goal:

on(A, B) and on(B, C)

on(B, C)

on(A, B)

on(A, B)
Linear planning corresponds to dealing with goals organized in a stack:

\[ \text{on}(A, B), \text{on}(B, C) \]

try to satisfy goal \( \text{on}(A, B) \)

solve sub-goals \[ \text{clear}(A), \text{clear}(B) \] \(^1\)

all sub-goals hold after \( \text{puttable}(C) \)

apply \( \text{put}(A, B) \)

goal \( \text{on}(A, B) \) is reached

try to satisfy goal \( \text{on}(B, C) \).
Interleaving of Goals

- Non-linear planning allows that a sequence of planning steps dealing with one goal is interrupted to deal with another goal.
- For the Sussman Anomaly, that means that after block $C$ is put on the table pursuing goal $on(A, B)$, the planner switches to the goal $on(B, C)$.
- Non-linear planning corresponds to dealing with goals organized in a set.
- The correct sequence of goals might not be found immediately without backtracking.
Interleaving of Goals cont.

\[
\{on(A, B), on(B, C)\}
\]

try to satisfy goal \(on(A, B)\)

\[
\{clear(A), clear(B), on(A, B), on(B, C)\}
\]

\(clear(A)\) and \(clear(B)\) hold after \(puttable(C)\)

try to satisfy goal \(on(B, C)\)

apply \(put(B, C)\)

try to satisfy goal \(on(A, B)\)

apply \(put(A, B)\).
Rocket Domain

(Veloso)

- Objects:
  - $n$ boxes, Positions (Earth, Moon), one Rocket

- Operators:
  - load/unload a box, move the Rocket
  - (oneway: only from earth to moon, no way back!)

- Linear planning:
  - to reach the goal, that Box1 is on the Moon, load it, shoot the Rocket, unload it, now no other Box can be transported!
The Register Assignment Problem

- State-variable formulation:

  Initial state: \{\text{value}(r1)=3, \text{value}(r2)=5, \text{value}(r3)=0\}

  Goal: \{\text{value}(r1)=5, \text{value}(r2)=3\}

  Operator: assign\(r,v,r',v'\)

    precond: \text{value}(r)=v, \text{value}(r')=v' \\
    effects: \text{value}(r)=v'

- STRIPS cannot solve this problem at all
The Sussman Anomaly can also be handled by the usage of domain-specific knowledge


Example: block stacking using forward search
Quick Review of Blocks World

**unstack(x,y)**
Pre: on(x,y), clear(x), handempty
Eff: \(~\text{on}(x,y), ~\text{clear}(x), ~\text{handempty}, \text{holding}(x), \text{clear}(y)\)

**stack(x,y)**
Pre: holding(x), clear(y)
Eff: \(~\text{holding}(x), ~\text{clear}(y), \text{on}(x,y), \text{clear}(x), \text{handempty}\)

**pickup(x)**
Pre: ontable(x), clear(x), handempty
Eff: \(~\text{ontable}(x), ~\text{clear}(x), ~\text{handempty}, \text{holding}(x)\)

**putdown(x)**
Pre: holding(x)
Eff: \(~\text{holding}(x), \text{ontable}(x), \text{clear}(?x), \text{handempty}\)
The Sussman Anomaly

Initial state

- On this problem, STRIPS can’t produce an irredundant solution
  - Try it and see
A blocks-world planning problem $P = (O, s_0, g)$ is solvable if $s_0$ and $g$ satisfy some simple consistency conditions:

- $g$ should not mention any blocks not mentioned in $s_0$
- A block cannot be on two other blocks at once
- etc.

$\Rightarrow$ Can check these in time $O(n \log n)$

If $P$ is solvable, can easily construct a solution of length $O(2m)$, where $m$ is the number of blocks:

- Move all blocks to the table, then build up stacks from the bottom

$\Rightarrow$ Can do this in time $O(n)$

With additional domain-specific knowledge can do even better . . .
Additional Domain-Specific Knowledge

- A block \( x \) needs to be moved if any of the following is true:
  - \( s \) contains \( \text{onTable}(x) \) and \( g \) contains \( \text{on}(x,y) \) - see a below
  - \( s \) contains \( \text{on}(x,y) \) and \( g \) contains \( \text{onTable}(x) \) - see d below
  - \( s \) contains \( \text{on}(x,y) \) and \( g \) contains \( \text{on}(x,z) \) for some \( y \neq z \)  
    \[ \Rightarrow \]  see c below
  - \( s \) contains \( \text{on}(x,y) \) and \( y \) needs to be moved - see e below
Domain-Specific Algorithm

\[\textbf{loop}\]
\[
\textbf{if} \text{ there is a clear block } x \text{ such that } \\
x \text{ needs to be moved \textbf{and} } \\
x \text{ can be moved to a place where it won’t need to be moved} \\
\textbf{then} \text{ move } x \text{ to that place} \\
\textbf{else} \text{ if there is a clear block } x \text{ such that } \\
x \text{ needs to be moved} \\
\textbf{then} \text{ move } x \text{ to the table} \\
\textbf{else} \text{ if the goal is satisfied} \\
\textbf{then} \text{return} \text{ the plan} \\
\textbf{else} \text{return} \text{ failure} \\
\] 

\[\textbf{repeat}\]
Easily Solves the Sussman Anomaly

\[\text{loop}\]

\[\text{if there is a clear block } x \text{ such that}\]
\[x \text{ needs to be moved and }\]
\[x \text{ can be moved to a place where it won’t need to be moved}\]
\[\text{then move } x \text{ to that place}\]
\[\text{else if there is a clear block } x \text{ such that}\]
\[x \text{ needs to be moved}\]
\[\text{then move } x \text{ to the table}\]
\[\text{else if the goal is satisfied}\]
\[\text{then return the plan}\]
\[\text{else return failure}\]

\[\text{repeat}\]
Properties

- The block-stacking algorithm:
  - Sound, complete, guaranteed to terminate
  - Runs in time $O(n^3)$
    - Can be modified to run in time $O(n)$
  - Often finds optimal (shortest) solutions
  - But sometimes only near-optimal (Exercise 4.22 in the book)
    - $PLAN\ LENGTH$ for the blocks world is NP-complete
Planning is search

Basic search techniques are forward (from initial state to a state fulfilling the goals) and backward (from the goals to the initial state)

For backward search an inverse state-transition operator has to be defined

Algorithms need to be sound and complete, furthermore, efficiency should be considered (branching factor during search!)

A classical planning algorithm is Strips

Strips is incomplete as demonstrated with the Sussman Anomaly

Incompleteness can be overcome by defining domain specific algorithms