Chapter 4
State-Space Planning

Dana S. Nau
University of Maryland

4:56 PM February 1, 2012
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
Outline

- State-space planning
  - Forward search
  - Backward search
  - Lifting
  - STRIPS
  - Block-stacking
Forward-search\((O, s_0, g)\)
\[
\begin{align*}
s & \leftarrow s_0 \\
\pi & \leftarrow \text{the empty plan} \\
\text{loop} & \\
\quad \text{if } s \text{ satisfies } g \text{ then return } \pi \\
\quad E & \leftarrow \{ a \mid a \text{ is a ground instance an operator in } O, \\
\quad & \text{ and } \text{precond}(a) \text{ is true in } s \} \\
\quad \text{if } E = \emptyset & \text{ then return failure} \\
\quad \text{nondeterministically choose an action } a \in E \\
\quad s & \leftarrow \gamma(s, a) \\
\pi & \leftarrow \pi.a
\end{align*}
\]
Properties

- Forward-search is *sound*
  - for any plan returned by any of its nondeterministic 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 nondeterministic traces will return a solution.
Deterministic Implementations

- 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
    » But classical planning has only finitely many states
    » Thus, can make depth-first search complete by doing loop-checking
Branching Factor of Forward Search

- 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 Part III (Heuristics and Control Strategies)
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 subgoals = $\gamma^{-1}(g,a)$

- 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
      - $g \cap \text{effects}(a) \neq \emptyset$
    - $a$ does not make any of $g$’s literals false
      - $g^+ \cap \text{effects}^-(a) = \emptyset$ 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
- $g = \{\text{on}(b1,b2), \text{on}(b2,b3)\}$
- $a = \text{stack}(b1,b2)$

What is $\gamma^{-1}(g,a)$?
Backward-search\((O, s_0, g)\)
\[\pi \leftarrow \text{the empty plan}\]
loop
\[\text{if } s_0 \text{ satisfies } g \text{ then return } \pi\]
\[A \leftarrow \{a | a \text{ is a ground instance of an operator in } O \]
\[\text{and } \gamma^{-1}(g, a) \text{ is defined}\}\]
\[\text{if } A = \emptyset \text{ then return failure}\]
nondeterministically choose an action \(a \in A\)
\[\pi \leftarrow a.\pi\]
\[g \leftarrow \gamma^{-1}(g, a)\]
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
Lifting

- Can reduce the branching factor of backward search if we partially instantiate the operators
  - this is called lifting

```
ontable(b_1)  
/          
pickup(b_1) /           holding(b_1)
on(b_1, b_1) /              unstack(b_1, b_1)
|  /     
on(b_1, b_2) /           unstack(b_1, b_2)
|    /     unstack(b_1, b_{50})
on(b_1, b_{50})
```
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

```plaintext
Lifted-backward-search(O, s₀, g)
    π ← the empty plan
    loop
        if s₀ satisfies g then return π
        A ← \{(o, θ) | o is a standardization of an operator in O,
                 θ is an mgu for an atom of g and an atom of \text{effects}^+(o),
                 and γ⁻¹(θ(g), θ(o)) is defined\}
        if A = ə then return failure
        nondeterministically choose a pair (o, θ) ∈ A
        π ← the concatenation of θ(o) and θ(π)
        g ← γ⁻¹(θ(g), θ(o))
```
The Search Space is Still Too Large

- 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)
Pruning the Search Space

- I’ll say a lot about this later, in Part III of the book
- For now, just two examples:
  - STRIPS
  - Block stacking
STRIPS

- Basic idea: given a compound goal \( g = \{g_1, g_2, \ldots\} \), try to solve each \( g_i \) separately
  - Works if the goals are *serializable* (can be solved in some linear order)

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

do a modified backward search from \( g \):

- instead of \( \gamma^{-1}(s,a) \), each new set of subgoals is just \( \text{precond}(a) \)
- whenever you find an action that’s executable in the current state, go forward on the current search path as far as possible, executing actions and appending them to \( \pi \)

repeat until all goals are satisfied
The Sussman Anomaly

On this problem, STRIPS can’t produce an irredundant solution

Try it and see
The Register Assignment Problem

- Interchange the values stored in two registers
  - State-variable formulation:
    » registers r1, r2, r3

  \[ s_0: \{ \text{value}(r1)=3, \text{value}(r2)=5, \text{value}(r3)=0 \} \]
  \[ g: \{ \text{value}(r1)=5, \text{value}(r2)=3 \} \]

  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
How to Handle Problems like These?

Several ways:

- Use a planning algorithm other than state-space search
  » e.g., Chapters 5–8

- Write a domain-specific algorithm
  » Example: the blocks world
Domain-Specific Knowledge

- A blocks-world planning problem $P = (O, s_0, g)$ is solvable iff $s_0$ and $g$ satisfy some simple consistency conditions
  - no block can be on two other blocks at once, every block in $g$ must also be in $s_0$, etc.
    - Can check these in time $O(n \log n)$
- If $P$ is solvable, can easily construct a solution of length $O(2^m)$, where $m$ is the number of blocks
  - Move all blocks to the table, then build up stacks from the bottom
    - Can do this in time $O(n)$
- With additional domain-specific knowledge, can do even better *(next slide)*
Block-Stacking Algorithm

- All of the possible situations in which a block \( x \) needs to be moved:
  - \( s \) contains \( \text{ontable}(x) \) and \( g \) contains \( \text{on}(x,y) \) - e.g., \( a \)
  - \( s \) contains \( \text{on}(x,y) \) and \( g \) contains \( \text{ontable}(x) \) - e.g., \( d \)
  - \( s \) contains \( \text{on}(x,y) \) and \( g \) contains \( \text{on}(x,z) \) for some \( y \neq z \) - e.g., \( c \)
  - \( s \) contains \( \text{on}(x,y) \) and \( y \) needs to be moved - e.g., \( e \)

loop
- if there is a clear block \( x \) that needs to be moved
  - and \( x \) can be moved to a place where it won’t need to be moved
    - then move \( x \) to that place
- else if there’s a clear block \( x \) that needs to be moved
  - then move \( x \) to the table
- else if the goal is satisfied then return the plan
- else return failure

repeat

initial state

goal
Properties of the Block-Stacking Algorithm

- Sound, complete, guaranteed to terminate
- Easily solves problems like the Sussman anomaly
- 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)
  - Recall that PLAN LENGTH for the blocks world is NP-complete