Plan-Space Search

Searching for a Solution Plan in a Graph of Partial Plans

Literature

State-Space vs. Plan-Space Search

- state-space search: search through graph of nodes representing world states
- plan-space search: search through graph of partial plans
  - nodes: partially specified plans
  - arcs: plan refinement operations
  - solutions: partial-order plans

Overview

- The Search Space of Partial Plans
- Plan-Space Search Algorithms
- Extensions of the STRIPS Representation
Partial Plans

- plan: set of actions organized into some structure
- partial plan:
  - subset of the actions
  - subset of the organizational structure
    - temporal ordering of actions
    - rationale: what the action achieves in the plan
  - subset of variable bindings

Adding Actions

- partial plan contains actions
  - initial state
  - goal conditions
  - set of operators with different variables

- reason for adding new actions
  - to achieve unsatisfied preconditions
  - to achieve unsatisfied goal conditions
Adding Actions: Example

initial state
- attached(pile,loc1)
- in(cont,pile)
- top(cont,pile)
- on(cont,pallet)
- belong(crane,loc1)
- empty(crane)
- adjacent(loc1,loc2)
- adjacent(loc2,loc1)
- at(robot,loc2)
- occupied(loc2)
- unloaded(robot)

1: move(r1,l1,m1)

preconditions
- all(r1,m1)
- occupied(m1)
- adjacent(m1)
- occupied(l1)

effects
- at(r1,l1)
- ¬occupied(m1)
- ¬occupied(l1)
- ¬at(r1,l1)

goal
- at(robot,loc2)
- unloaded(robot)

2: load(k2,l2,c2,r2)

preconditions
- belong(k2,l2)
- holding(k2,c2)
- at(robot,loc2)
- occupied(loc2)

effects
- empty(k2)
- loaded(r2,c2)
- ¬holding(k2,c2)
- ¬unloaded(r2)
- unloaded(r2)

Adding Causal Links

- partial plan contains causal links
  - links from the provider
    - an effect of an action or
    - an atom that holds in the initial state
  - to the consumer
    - a precondition of an action or
    - a goal condition
- reasons for adding causal links
  - prevent interference with other actions
Adding Causal Links: Example

**Initial State**
- attached(pile,loc)
- in(cont,pile)
- top(cont,pile)
- on(cont,pallet)
- belong(crane,loc1)
- empty(crane)
- adjacent(loc1,loc2)
- at(robot,loc2)
- occupied(loc2)
- unloaded(robot)

**Goals**
- at(robot,loc2)
- unloaded(robot)

**1: move(r1,l1,m1)**

**Preconditions**
- all(r1)
- occupied(m1)
- adjacent(l1,m1)

**Effects**
- all(r1, m1)
- occupied(m1)
- at(r1, l1)
- empty(m1)
- loaded(r1, cont)

**2: load(k2,l2,c2,r2)**

**Preconditions**
- belong(k2, l1)
- holding(k2, c2)
- at(r2, l2)

**Effects**
- empty(k2)
- loaded(r2, c2)
- holding(k2, c2)
- unloaded(r2)

**Causal Link:**

---

Adding Variable Bindings

- partial plan contains variable bindings
  - new operators introduce new (copies of) variables into the plan
  - solution plan must contain actions
  - variable binding constraints keep track of possible values for variables and co-designation
- reasons for adding variable bindings
  - to turn operators into actions
  - to unify and effect with the precondition it supports
Adding Variable Bindings: Example

```
initial state
- attached(pile,loc1)
- in(cont,pile)
- top(cont,pile)
- on(cont,pallet)
- belong(crane,loc1)
- empty(crane)
- adjacent(loc1,loc2)
- adjacent(loc2,loc1)
- at(robot,loc2)
- occupied(loc2)
- unloaded(robot)
```

1: move(r₁,l₁,m₁)

```
preconditions
- at(r₁,loc1)
- occupied(m₁)
- adjacent(l₁,m₁)

effects
- at(r₁,loc1)
- ¬occupied(m₁)
- at(r₁,m₁)
- occupied(m₁)
- ¬occupied(l₁)
- ¬at(r₁,loc1)
```

```
goal
- at(robot,loc2)
- unloaded(robot)
```

variable bindings:
```
<table>
<thead>
<tr>
<th>variable</th>
<th>=</th>
<th>≠</th>
</tr>
</thead>
<tbody>
<tr>
<td>r₁</td>
<td>robot</td>
<td></td>
</tr>
<tr>
<td>l₁</td>
<td>loc1</td>
<td>loc2</td>
</tr>
<tr>
<td>m₁</td>
<td>loc2</td>
<td></td>
</tr>
</tbody>
</table>
```

Adding Ordering Constraints

- partial plan contains ordering constraints
  - binary relation specifying the temporal order between actions in the plan
- reasons for adding ordering constraints
  - all actions after initial state
  - all actions before goal
  - causal link implies ordering constraint
  - to avoid possible interference
Adding Ordering Constraints: Example

Definition of Partial Plans

- A partial plan is a tuple \( \pi = (A, \prec, B, L) \), where:
  - \( A = \{a_1, \ldots, a_k\} \) is a set of partially instantiated planning operators;
  - \( \prec \) is a set of ordering constraints on \( A \) of the form \( (a_i \prec a_j) \);
  - \( B \) is a set of binding constraints on the variables of actions in \( A \) of the form \( x = y \), \( x \neq y \), or \( x \in D_x \);
  - \( L \) is a set of causal links of the form \( \langle a_i \leftarrow [p] \rightarrow a_j \rangle \) such that:
    - \( a_i \) and \( a_j \) are actions in \( A \);
    - the constraint \( (a_i \prec a_j) \) is in \( \prec \);
    - proposition \( p \) is an effect of \( a_i \) and a precondition of \( a_j \); and
    - the binding constraints for variables in \( a_i \) and \( a_j \) appearing in \( p \) are in \( B \).
Plan-Space Search: Initial Search State

- represent initial state and goal as dummy actions
  - init: no preconditions, initial state as effects
  - goal: goal conditions as preconditions, no effects
- empty plan $\pi_0 = (\{\text{init}, \text{goal}\}, \{\text{init}<\text{goal}\}, \{\}, \{\})$:
  - two dummy actions init and goal;
  - one ordering constraint: init before goal;
  - no variable bindings; and
  - no causal links.

Plan-Space Search: Initial Search State Example

![Diagram showing initial and goal states with variables and their conditions]
Plan-Space Search: Successor Function

- states are partial plans
- generate successor through plan refinement operators (one or more):
  - adding an action to $A$
  - adding an ordering constraint to $\prec$
  - adding a binding constraint to $B$
  - adding a causal link to $L$

Total vs. Partial Order

- Let $\mathcal{P}=(\Sigma, s_0, g)$ be a planning problem. A plan $\pi$ is a solution for $\mathcal{P}$ if $\gamma(s_i, \pi)$ satisfies $g$.
- problem: $\gamma(s_i, \pi)$ only defined for sequence of ground actions
  - partial order corresponds to total order in which all partial order constraints are respected
  - partial instantiation corresponds to grounding in which variables are assigned values consistent with binding constraints
Partial Order Solutions

- Let $\mathcal{P}=(\Sigma, s, g)$ be a planning problem. A plan $\pi = (A, \prec, B, L)$ is a (partial order) solution for $\mathcal{P}$ if:
  - its ordering constraints $\prec$ and binding constraints $B$ are consistent; and
  - for every sequence $\langle a_1, \ldots, a_k \rangle$ of all the actions in $A \setminus \{\text{init}, \text{goal}\}$ that is
    - totally ordered and grounded and respects $\prec$ and $B$
    - $\gamma(s_{\pi}, \langle a_1, \ldots, a_k \rangle)$ must satisfy $g$.

Threat: Example

1. \texttt{move(robot,loc1,loc2)}
   - preconditions
     - at(robot,loc1)
     - \neg occupied(loc2)
     - adjacent(loc1,loc2)
   - effects
     - at(robot,loc2)
     - occupied(loc2)

2. \texttt{load(crane,loc1,cont,robot)}
   - preconditions
     - belong(crane,loc1)
     - holding(crane,cont)
     - at(robot,loc1)
     - empty(crane)
   - effects
     - at(robot,loc1)
     - loaded(robot,cont)
     - loaded(crane,cont)

3. \texttt{move(robot,loc2,loc1)}
   - preconditions
     - at(robot,loc2)
     - occupied(loc1)
     - at(robot,loc1)
   - effects
     - at(robot,loc1)
     - \neg unloaded(robot)

0. \texttt{goal}
   - preconditions
     - at(robot,loc2)
   - effects
     - at(robot,loc2)
     - \neg unloaded(robot)
**Threats**

- An action $a_k$ in a partial plan $\pi = (A, <, B, L)$ is a threat to a causal link $\langle a_i - [p] \rightarrow a_j \rangle$ iff:
  - $a_k$ has an effect $\neg q$ that is possibly inconsistent with $p$, i.e. $q$ and $p$ are unifiable;
  - the ordering constraints $(a_i < a_k)$ and $(a_k < a_j)$ are consistent with $<$; and
  - the binding constraints for the unification of $q$ and $p$ are consistent with $B$.

**Flaws**

- A flaw in a plan $\pi = (A, <, B, L)$ is either:
  - an unsatisfied sub-goal, i.e. a precondition of an action in $A$ without a causal link that supports it; or
  - a threat, i.e. an action that may interfere with a causal link.
**Flawless Plans and Solutions**

- **Proposition**: A partial plan \( \pi = (A, \prec, B, L) \) is a solution to the planning problem \( P = (\Sigma, s_i, g) \) if:
  - \( \pi \) has no flaw;
  - the ordering constraints \( \prec \) are not circular; and
  - the variable bindings \( B \) are consistent.

- **Proof**: by induction on number of actions in \( A \)
  - base case: empty plan
  - induction step: totally ordered plan minus first step is solution implies plan including first step is a solution:
    \[
    \gamma(s_i, \langle a_1, \ldots, a_k \rangle) = \gamma(\gamma(s_i, a_1), \langle a_2, \ldots, a_k \rangle)
    \]

**Overview**

- The Search Space of Partial Plans
- **Plan-Space Search Algorithms**
- Extensions of the STRIPS Representation
Plan-Space Planning as a Search Problem

- given: statement of a planning problem \( P = (O, s_i, g) \)
- define the search problem as follows:
  - initial state: \( \pi_0 = (\{\text{init, goal}\}, \{\text{init} \prec \text{goal}\}, \emptyset, \emptyset) \)
  - goal test for plan state \( p \): \( p \) has no flaws
  - path cost function for plan \( \pi \): \( |\pi| \)
  - successor function for plan state \( p \): refinements of \( p \) that maintain \( \prec \) and \( B \)

PSP Procedure: Basic Operations

- PSP: Plan-Space Planner
- main principle: refine partial \( \pi \) plan while maintaining \( \prec \) and \( B \) consistent until \( \pi \) has no more flaws
- basic operations:
  - find the flaws of \( \pi \), i.e. its sub-goals and its threats
  - select one of the flaws
  - find ways to resolve the chosen flaw
  - choose one of the resolvers for the flaw
  - refine \( \pi \) according to the chosen resolver
**PSP: Pseudo Code**

```plaintext
function PSP(plan)
    allFlaws ← plan.openGoals() + plan.threats()
    if allFlaws.empty() then return plan
    flaw ← allFlaws.selectOne()
    allResolvers ← flaw.getResolvers(plan)
    if allResolvers.empty() then return failure
    resolver ← allResolvers.chooseOne()
    newPlan ← plan.refine(resolver)
    return PSP(newPlan)
```

**PSP: Choice Points**

- `resolver ← allResolvers.chooseOne()`
  - non-deterministic choice
- `flaw ← allFlaws.selectOne()`
  - deterministic selection
  - all flaws need to be resolved before a plan becomes a solution
  - order not important for completeness
  - order is important for efficiency
**Implementing plan.openGoals()**

- finding unachieved sub-goals (incrementally):
  - in $\pi_0$: goal conditions
  - when adding an action: all preconditions are unachieved sub-goals
  - when adding a causal link: protected proposition is no longer unachieved

**Implementing plan.threats()**

- finding threats (incrementally):
  - in $\pi_0$: no threats
  - when adding an action $a_{\text{new}}$ to $\pi = (A, <, B, L)$:
    - for every causal link $\langle a_i, [-p] \Rightarrow a_j \rangle \in L$
      - if $(a_{\text{new}} < a_i)$ or $(a_i < a_{\text{new}})$ then next link
      - else for every effect $q$ of $a_{\text{new}}$
        - if $(\exists \sigma: \sigma(p) = \sigma(\neg q))$ then $q$ of $a_{\text{new}}$ threatens $\langle a_i, [-p] \Rightarrow a_j \rangle$
  - when adding a causal link $\langle a_i, [-p] \Rightarrow a_j \rangle$ to $\pi = (A, <, B, L)$:
    - for every action $a_{\text{old}} \in A$
      - if $(a_{\text{old}} < a_i)$ or $(a_i = a_{\text{old}})$ or $(a_j < a_{\text{old}})$ then next action
      - else for every effect $q$ of $a_{\text{old}}$
        - if $(\exists \sigma: \sigma(p) = \sigma(\neg q))$ then $q$ of $a_{\text{old}}$ threatens $\langle a_i, [-p] \Rightarrow a_j \rangle$
Implementing flaw.getResolvers(plan)

- for unachieved precondition \( p \) of \( a_g \):
  - add causal links to an existing action:
    - for every action \( a_{old} \in A \)
      if \( (a_g = a_{old}) \) or \( (a_g < a_{old}) \) then next action
      else for every effect \( q \) of \( a_{old} \)
      if \( (\exists \sigma: \sigma(p) = \sigma(q)) \) then adding
      \( \langle a_{old} - [\sigma(p)] \rightarrow a_g \rangle \) is a resolver
  - add a new action and a causal link:
    - for every effect \( q \) of every operator \( o \)
      if \( (\exists \sigma: \sigma(p) = \sigma(q)) \) then adding
      \( a_{new} = o.newInstance() \) and
      \( \langle a_{new} - [\sigma(p)] \rightarrow a_g \rangle \) is a resolver

- for effect \( q \) of action \( a_i \) threatening \( \langle a_i - [p] \rightarrow a_j \rangle \):
  - order action before threatened link:
    - if \( (a_i = a_j) \) or \( (a_i < a_j) \) then not a resolver
    else adding \( (a_i < a_j) \) is a resolver
  - order threatened link before action:
    - if \( (a_i = a_j) \) or \( (a_i < a_j) \) then not a resolver
    else adding \( (a_i < a_j) \) is a resolver
  - extend variable bindings such that unification fails:
    - for every variable \( v \) in \( p \) or \( q \)
      if \( v \neq \sigma(v) \) is consistent with \( B \) then
      adding \( v \neq \sigma(v) \) is a resolver
**Implementing**

**plan.refine(resolver)**

- refines partial plan with elements in resolver by adding:
  - an ordering constraint;
  - one or more binding constraints;
  - a causal link; and/or
  - a new action.
- no testing required
- must update flaws:
  - unachieved preconditions (see: plan.openGoals())
  - threats (see: plan.threats())

**Maintaining Ordering Constraints**

- required operations:
  - query whether \(a_i < a_j\)
  - adding \(a_i < a_j\)
- possible internal representations:
  - maintain set of predecessors/successors for each action as given
  - maintain only direct predecessors/successors for each action
  - maintain transitive closure of \(<\) relation
Maintaining Variable Binding Constraints

- types of constraints:
  - unary constraints: \( x \in D_x \)
  - equality constraints: \( x = y \)
  - inequalities: \( x \neq y \)

- note: general CSP problem is NP-complete

PSP: Data Flow

\[ \pi_0 \]

Plan = \((A, \preceq, B, L)\)

- compute threats
- compute open goals

- has flaw?
  - select flaw
  - compute resolvers
  - has resolver?
    - choose resolver
    - apply resolvers
    - maintain ordering constraints
    - maintain binding constraints

- return failure
- return plan
**PSP: Sound and Complete**

- **Proposition**: The PSP procedure is sound and complete: whenever $\pi_0$ can be refined into a solution plan, $\text{PSP}(\pi_0)$ returns such a plan.

- **Proof**:
  - soundness: $\prec$ and $B$ are consistent at every stage of the refinement
  - completeness: induction on the number of actions in the solution plan

**PSP Implementation: PoP**

- extended input:
  - partial plan (as before)
  - agenda: set of pairs $(a,p)$ where $a$ is an action and $p$ is one of its preconditions

- search control by flaw type
  - unachieved sub-goal (on agenda): as before
  - threats: resolved as part of the successor generation process
**PoP: Pseudo Code (1)**

function PoP(plan, agenda)

  if agenda.empty() then return plan

  (a_g,p_g) ← agenda.selectOne()
  agenda ← agenda - (a_g,p_g)
  relevant ← plan.getProviders(p_g)

  if relevant.empty() then return failure

  (a_p,p_p,σ) ← relevant.chooseOne()
  plan.L ← plan.L ∪ \langle a_p -[p] \rightarrow a_g \rangle
  plan.B ← plan.B ∪ σ

**PoP: Pseudo Code (2)**

if a_p ∉ plan.A then

  plan.add(a_p)
  agenda ← agenda + a_p.preconditions
  newPlan ← plan

  for each threat on \langle a_p -[p] \rightarrow a_g \rangle or due to a_p do

    allResolvers ← threat.getResolvers(newPlan)

    if allResolvers.empty() then return failure

    resolver ← allResolvers.chooseOne()
    newPlan ← newPlan.refine(resolver)

  return PSP(newPlan,agenda)
State-Space vs. Plan-Space Planning

- State-space planning
  - finite search space
  - explicit representation of intermediate states
  - action ordering reflects control strategy
  - causal structure only implicit
  - search nodes relatively simple and successors easy to compute

- Plan-space planning
  - finite search space
  - no intermediate states
  - choice of actions and organization independent
  - explicit representation of rationale
  - search nodes are complex and successors expensive to compute

Using Partial-Order Plans: Main Advantages

- more flexible during execution
- using constraint managers facilitates extensions such as:
  - temporal constraints
  - resource constraints
- distributed and multi-agent planning fit naturally into the framework
Overview

- The Search Space of Partial Plans
- Plan-Space Search Algorithms
  - Extensions of the STRIPS Representation

Existential Quantification in Goals

- allow existentially quantified conjunction of literals as goal:
  - \( g = \exists x_1, \ldots, x_n : l_1 \land \ldots \land l_m \)
- rewrite into equivalent planning problem:
  - new goal \( g' = \{p\} \) where \( p \) is an unused proposition symbol
  - introduce additional operator
    \( o = (\text{op-g}(x_1, \ldots, x_n), \{l_1, \ldots, l_m\}, \{p\}) \)
- in plan-space search: no change needed
DWR Example: Existential Quantification in Goals

- goal: $\exists x, y: \text{on}(x, c1) \land \text{on}(y, c2)$

- rewritten goal: $p$
- new operator:
  $o = (\text{op-g}(x, y), \{\text{on}(x, c1), \text{on}(y, c2)\}, \{p\})$

- plan-space search goal: $\text{on}(x, c1) \land \text{on}(y, c2)$

Typed Variables

- allow typed variables in operators:
  - name($o$) = $n(x_1: t_1, \ldots, x_k: t_k)$ where $t_i$ is the type of variable $x_i$
- rewrite into equivalent planning problem:
  - add preconditions $\{t_1(x_1), \ldots, t_k(x_k)\}$ to $o$
  - if constant $c_i$ is of type $t_j$, add rigid relation $t_j(c_i)$ to the initial state
  - remove types from operator names
**DWR Example: Typed Variables**

- **operator:** `move(r:robot, l:location, m:location)`
  - **precond:** `adjacent(l, m), at(r, l), ¬occupied(m)`
  - **effects:** `at(r, m), occupied(m), ¬occupied(l), ¬at(r, l)`

- **rewritten operator:** `move(l, r, m)`
  - **precond:** `adjacent(l, m), at(r, l), ¬occupied(m), robot(r), location(l), location(m)`
  - **effects:** `at(r, m), occupied(m), ¬occupied(l), ¬at(r, l)`

- **rewritten initial state:**
  - `s_i ∪ {robot(r1), container(c1), container(c2),...}`

---

**Conditional Operators**

- **conditional planning operators:**
  - `o = (n, (precond_0, effects_0),..., (precond_n, effects_n))`
    where:
    - `n = o(x_1,...,x_n)` as before,
    - `(precond_0, effects_0)` are the unconditional preconditions and effects of the operator, and
    - `(precond_i, effects_i)` for `i ≥ 1` are the conditional preconditions and effects of the operator.
  - A ground instance `a` of `o` is applicable in state `s` if `s` satisfies `precond_0`
  - Let `I = {i ∈ [0,n] | s satisfies precond_i(a)}`; then:
    - `γ(s,a) = (s - ∪_{i∈I} effects_i(a)) ∪ (∪_{i∈I} effects_i*(a))`
DWR Example: Conditional Operators

- relation \( \text{at}(o,l) \): object \( o \) is at location \( l \)
- conditional move operator:
  \( \text{move}(r,l,m,c) \)
  - \( \text{precond}_0 \): \( \text{adjacent}(l,m) \), \( \text{at}(r,l) \), \( \neg \text{occupied}(m) \)
  - \( \text{effects}_0 \): \( \text{at}(r,m) \), \( \text{occupied}(m) \), \( \neg \text{occupied}(l) \), \( \neg \text{at}(r,l) \)
  - \( \text{precond}_1 \): \( \text{loaded}(r,c) \)
  - \( \text{effects}_1 \): \( \text{at}(c,m) \), \( \neg \text{at}(c,l) \)

Extending PoP to handle Conditional Operators

- modifying \( \text{plan}.\text{getProviders}(p_g) \):
  - new action with matching conditional effect
  - add precondition of conditional effect to agenda
- managing conditional threats:
  - new alternative resolver: add negated precondition of threatening conditional effect to agenda
Quantified Expressions

- allow universally quantified variables in conditional preconditions and effects:
  - for-all $x_1, \ldots, x_n$: $(\text{precond}_i, \text{effects}_i)$
- $a$ is applicable in state $s$ if $s$ satisfies $\text{precond}_0$
- Let $\sigma$ be a substitution for $x_1, \ldots, x_n$ such that $\sigma(\text{precond}_i(a))$ and $\sigma(\text{effects}_i(a))$ are ground.
  - If $s$ satisfies $\sigma(\text{precond}_i(a))$ then
  - $\sigma(\text{effects}_i(a))$ are effects of the action.

DWR Example: Quantified Expressions

- extension: robots can carry multiple containers
- extended move operator:
  - $\text{move}(r, l, m)$
    - precond$_0$: adjacent$(l, m)$, at$(r, l)$, $\neg$occupied$(m)$
    - effects$_0$: at$(r, m)$, occupied$(m)$, $\neg$occupied$(l)$, $\neg$at$(r, l)$
    - for-all $x$:
      - precond$_i$: loaded$(r, x)$
      - effects$_i$: at$(x, m)$, $\neg$at$(x, l)$
Disjunctive Preconditions

- allow alternatives (disjunctions) in preconditions:
  - precond = precond₁ ∨ … ∨ precondₙ
  - a is applicable in state s if s satisfies at least one of precond₁ … precondₙ
  - effects remain unchanged
- rewrite:
  - replace operator with n disjunctive preconditions by n operators with precondᵢ as precondition

DWR Example: Disjunctive Preconditions

- robot can move between locations if there is a road between them or the robot has all-wheel drive
- extended move operator:
  move(r,l,m)
  - precond: (road(l,m), at(r,l), ¬occupied(m)) ∨ (all-wheel-drive(r), at(r,l), ¬occupied(m))
  - effects: at(r,m), occupied(m), ¬occupied(l), ¬at(r,l)
Axiomatic Inference: Static Case

- axioms over rigid relations:
  - example:
    \[ \forall l_1, l_2: \text{adjacent}(l_1, l_2) \leftrightarrow \text{adjacent}(l_2, l_1) \]
- state-specific axioms:
  - example:
    \[ \forall c: \text{container}(c) \leftrightarrow \text{at}(c, \text{loc1}) \text{ holds in } s_i \]
- approach: pre-compute

Axiomatic Inference: Dynamic Case

- axioms over flexible relations:
  - example: \[ \forall k, x: \neg \text{holding}(k, x) \leftrightarrow \text{empty}(k) \]
  - approach:
    - divide relations into primary and secondary where secondary relations do not appear in effects
    - transform axioms into implications where primary relations must not appear in right-hand side
  - example:
    - primary: holding / secondary: empty
      \[ \forall k \neg \exists x: \text{holding}(k, x) \rightarrow \text{empty}(k) \]
      \[ \forall k \exists x: \text{holding}(k, x) \rightarrow \neg \text{empty}(k) \]
Extended Goals

- not part of classical planning formalisms
- some problems can be translated into equivalent classical problems, e.g.
  - states to be avoided: add corresponding preconditions to operators
  - states to be visited twice: introduce visited relation and maintain in operators
  - constraints on solution length: introduce count relation that is increased with each step

Other Extensions

- Function Symbols
  - infinite domains, undecidable in general
- Attached Procedures
  - evaluate relations using special code rather than general inference
    - efficiency may be necessary in real-world domains
    - variables must usually be bound to evaluate relations
    - semantics of such relations
Overview

- The Search Space of Partial Plans
- Plan-Space Search Algorithms
- Extensions of the STRIPS Representation