Planning

Chapter 11
Outline

◊ Search vs. planning
◊ STRIPS operators
◊ Partial-order planning
◊ Planning graphs and GRAPHPLAN
Search vs. planning

Consider the task *get milk, bananas, and a cordless drill*

Standard search algorithms seem to fail miserably:

After-the-fact heuristic/goal test inadequate
Planning systems do the following:

1) open up action and goal representation to allow selection
2) divide-and-conquer by subgoaling
3) relax requirement for sequential construction of solutions

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Planning</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>Lisp data structures</td>
<td>Logical sentences</td>
</tr>
<tr>
<td>Actions</td>
<td>Lisp code</td>
<td>Preconditions/outcomes</td>
</tr>
<tr>
<td>Goal</td>
<td>Lisp code</td>
<td>Logical sentence (conjunction)</td>
</tr>
<tr>
<td>Plan</td>
<td>Sequence from $S_0$</td>
<td>Constraints on actions</td>
</tr>
</tbody>
</table>
STRIPS operators

Tidily arranged actions descriptions, restricted language

**ACTION:** $Buy(x)$

**PRECONDITION:** $At(p), Sells(p, x)$

**EFFECT:** $Have(x)$

[Note: this abstracts away many important details!]

Restricted language $\Rightarrow$ efficient algorithm

- Precondition: conjunction of positive literals
- Effect: conjunction of literals

A complete set of STRIPS operators can be translated into a set of successor-state axioms
Partially ordered plans

*Partially ordered* collection of steps with

- **Start** step has the initial state description as its effect
- **Finish** step has the goal description as its precondition
- causal links from outcome of one step to precondition of another
- temporal ordering between pairs of steps

**Open condition** = precondition of a step not yet causally linked

A plan is **complete** iff every precondition is achieved

A precondition is **achieved** iff it is the effect of an earlier step and no possibly intervening step undoes it
Example

Start

At(Home)  Sells(HWS,Drill)  Sells(SM,Milk)  Sells(SM,Ban.)

Have(Milk)  At(Home)  Have(Ban.)  Have( Drill)

Finish
Example

- **Start**
  - At(Home)
  - Sells(HWS,Drill)
  - Sells(SM,Milk)
  - Sells(SM,Ban.)

- **At(HWS)**
  - Sells(HWS,Drill)
  - Buy(Drill)

- **At(x)**
  - Go(SM)

- **At(SM)**
  - Sells(SM,Milk)
  - Buy(Milk)

- **Have(Milk)**
  - At(Home)
  - Have(Ban.)
  - Have(Drill)

- **Finish**
Example

- **Start**
  - At(Home)
  - Go(HWS)
  - At(HWS) → Sells(HWS,Drill)
  - Buy(Drill)
    - At(HWS)
  - Go(SM)
    - At(SM) → Sells(SM,Milk)
    - Buy(Milk)
      - At(SM) → Sells(SM,Ban.)
    - Buy(Ban.)
    - Go/Home
      - At(SM)

- **Finish**
  - Have(Milk) → At(Home) → Have(Ban.) → Have(Drill)
Planning process

Operators on partial plans:
- add a link from an existing action to an open condition
- add a step to fulfill an open condition
- order one step wrt another to remove possible conflicts

Gradually move from incomplete/vague plans to complete, correct plans

Backtrack if an open condition is unachievable or
if a conflict is unresolvable
POP algorithm sketch

function POP(initial, goal, operators) returns plan

    plan ← Make-Minimal-Plan(initial, goal)

    loop do
        if Solution?(plan) then return plan
        S_need, c ← Select-Subgoal(plan)
        Choose-Operator(plan, operators, S_need, c)
        Resolve-Threats(plan)
    end

function Select-Subgoal(plan) returns S_need, c

    pick a plan step S_need from Steps(plan)
    with a precondition c that has not been achieved

    return S_need, c
POP algorithm contd.

**procedure** CHOOSE-OPERATOR\((plan, operators, S_{need}, c)\)

choose a step \(S_{add}\) from \(operators\) or \(STEPS(plan)\) that has \(c\) as an effect
if there is no such step then fail
add the causal link \(S_{add} \rightarrow S_{need}\) to \(LINKS(plan)\)
add the ordering constraint \(S_{add} < S_{need}\) to \(ORDERINGS(plan)\)
if \(S_{add}\) is a newly added step from \(operators\) then
  add \(S_{add}\) to \(STEPS(plan)\)
  add \(Start < S_{add} < Finish\) to \(ORDERINGS(plan)\)

**procedure** RESOLVE-THREATS\((plan)\)

for each \(S_{threat}\) that threatens a link \(S_i \rightarrow S_j\) in \(LINKS(plan)\) do
  choose either
  \[\text{Demotion: Add } S_{threat} < S_i \text{ to } ORDERINGS(plan)\]
  \[\text{Promotion: Add } S_j < S_{threat} \text{ to } ORDERINGS(plan)\]
  if not CONSISTENT\((plan)\) then fail
end
A clobberer is a potentially intervening step that destroys the condition achieved by a causal link. E.g., $Go(Home)$ clobbers $At(Supermarket)$:

**Demotion:** put before $Go(Supermarket)$

**Promotion:** put after $Buy(Milk)$
Properties of POP

Nondeterministic algorithm: backtracks at choice points on failure:
- choice of $S_{add}$ to achieve $S_{need}$
- choice of demotion or promotion for clobberer
- selection of $S_{need}$ is irrevocable

POP is sound, complete, and systematic (no repetition)

Extensions for disjunction, universals, negation, conditionals

Can be made efficient with good heuristics derived from problem description

Particularly good for problems with many loosely related subgoals
Example: Blocks world

"Sussman anomaly" problem

Start State

\[ \begin{align*}
\text{Clear}(x) & \quad \text{On}(x,z) \quad \text{Clear}(y) \\
\text{PutOn}(x,y) \\
\sim \text{On}(x,z) & \quad \sim \text{Clear}(y) \\
\text{Clear}(z) & \quad \text{On}(x,y)
\end{align*} \]

Goal State

\[ \begin{align*}
\text{Clear}(x) & \quad \text{On}(x,z) \\
\text{PutOnTable}(x) \\
\sim \text{On}(x,z) & \quad \text{Clear}(z) \quad \text{On}(x,\text{Table})
\end{align*} \]

+ several inequality constraints
Example contd.

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

On(A,B) On(B,C)

FINISH
Example contd.

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

Cl(B) On(B,z) Cl(C)

PutOn(B,C)

On(A,B) On(B,C)

FINISH
Example contd.

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

PutOn(A,B) clobbers Cl(B) => order after PutOn(B,C)

PutOn(A,B)

PutOn(B,C)

FINISH
Example contd.

On(C,A)  On(A,Table)  Cl(B)  On(B,Table)  Cl(C)

PutOnTable(C)

PutOn(A,B)  

On(A,B)  On(B,C)

FINISH
Planning Graphs

Heuristics used for total or partial-order planning can be inaccurate
  – Number of open subgoals
  – Relaxed planning (e.g., no negative effects)

Planning graphs consist of a sequence of levels or time steps of the plan
  – Each level contains all literals true after applying all actions
  – Each level also contains all applicable actions

Planning graphs can be used to give better heuristic estimates

Plans can be extracted from planning graphs using, e.g., Graphplan
Planning problem:

Init(Have(Cake))
Goal(Have(Cake) \land Eaten(Cake))
Action: Eat(Cake)
  Precond: Have(Cake)
  Effect: \neg Have(Cake) \land Eaten(Cake)
Action: Bake(Cake)
  Precond: \neg Have(Cake)
  Effect: Have(Cake)

Planning graph: (square = persistence action, gray line = mutex)
Planning Graphs

Each level $S_i$ represents multiple states

Mutex links are constraints that define the possible states

Alternate between state level $S_i$ and action level $A_i$

Continue until leveled off - two consecutive levels are identical
- Every subsequent level will be identical
- Further expansion is unnecessary

Note: We do not choose among actions (avoids combinatorial search)

Planning graph generated in polynomial time despite exponential state space
Mutual Exclusion (Mutex) Links

A mutex relation holds between two actions at the same level if one of these three conditions holds:

1. **Inconsistent effects**: One action negates the effects of the other. E.g., *Eat(Cake)* and the persistence of *Have(Cake)*.

2. **Interference**: One of the effects of one action is the negation of a precondition of the other. E.g., *Eat(Cake)* and the persistence of *Have(Cake)*.

3. **Competing needs**: One of the preconditions of one action is mutually exclusive with a precondition of the other. E.g., *Bake(Cake)* and *Eat(Cake)*.

A mutex relation holds between two literals at the same level if
- One is the negation of the other, or
- Each possible pair of actions achieving the two literals is mutex
- E.g., *Have(Cake)* and *Eaten(Cake)*
A planning graph is a rich source of information about the problem.

A literal that does not appear in the final level of the graph cannot be achieved by any plan.

– Any state $n$ containing an unachievable literal has cost $h(n) = \infty$

In general, we can estimate the cost of achieving and goal literal as the level at which it first appears in the planning graph

– This is called the level cost
– Level cost is an admissible heuristic
– May be inaccurate due to multiple action effects per time step

A serial planning graph constrains only one action per time step

– Add mutex links between every pair of actions except persistence actions
Planning Graphs for Heuristic Estimation

Estimating cost of a conjunction of goals

Approach 1: The max-level heuristic
   – Maximum level cost of any goal
   – Admissible, but not always accurate

Approach 2: The level-sum heuristic
   – Sum of level costs of all goals
   – Not admissible, but works well in practice

Approach 3: The set-level heuristic
   – Level at which all goal literals appear without mutex links
   – Admissible, and dominates max-level
   – Works well on problems with significant interaction among subplans
The Graphplan Algorithm

Algorithm for extracting a plan directly from a planning graph

```
function Graphplan(problem) returns solution or failure
    graph ← Initial-Planning-Graph(problem)
    goals ← Goals(problem)
    loop do
        if goals all non-mutex in last level of graph then do
            solution ← Extract-Solution(graph, goals, Length(graph))
            if solution ≠ failure then return solution
        else if No-Solution-Possible(graph) then return failure
        graph ← Expand-Graph(graph, problem)
```
The **Graphplan Algorithm**

**Extract-Solution**: Looks for whether a plan can be found starting at the end and searching backwards.

**No-Solution-Possible**: Checks if planning graph has leveled off and if either one of the goals is missing, or mutex with another goal.

**Expand-Graph**: Adds the actions for the current level and the state literals for the next level.
Summary

Planning systems are problem-solvers that operate on explicit representations of states and actions.

Explicit representation allows more effective heuristics.

Partial-order planning (POP) algorithms explore the space of plans without committing to a totally-ordered sequence of actions.

Planning graphs can be used to derive useful planner heuristic or produce plans directly via GRAPHPLAN.

Planning is still hard, and no one approach is best.