Planning Problem

• What is the planning problem?

• Find a sequence of actions that achieves a goal when executed from an initial state.

• That is, given
  • A set of operators (possible actions)
  • An initial state description
  • A goal (description or conjunction of predicates)

• Compute a sequence of operations: a plan.
Typical Assumptions

- **Atomic time**: Each action is indivisible
- **No concurrent actions** allowed
- **Deterministic actions**
  - The result of actions are completely known – no uncertainty
- Agent is the **sole cause of change** in the world
- Agent is **omniscient**:
  - Has complete knowledge of the state of the world
- **Closed world assumption**:
  - Everything known-true about the world is in the *state description*
  - Anything not known-true is known-false

Blocks World

The **blocks world** consists of a table, set of blocks, and a robot gripper

Some domain constraints:
- Only one block on another block
- Any number of blocks on table
- Hand can only hold one block

Typical representation:

- `ontable(a)`  `handempty`
- `ontable(c)`  `on(b,a)`
- `clear(b)`  `clear(c)`

![Diagram of blocks world]
Major Approaches

- GPS / STRIPS
- Situation calculus
- Partial order planning
  - Hierarchical decomposition (HTN planning)
  - Planning with constraints (SATplan, Graphplan)
- Reactive planning

General Problem Solver

- The General Problem Solver (GPS) system
  - An early planner (Newell, Shaw, and Simon)
  - Generate actions that reduce difference between state and goal state
  - Uses Means-Ends Analysis
    - Compare what is given or known with what is desired
    - Select a reasonable thing to do next
    - Use a table of differences to identify procedures to reduce differences
- GPS is a state space planner
  - Operates on state space problems specified by an initial state, some goal states, and a set of operations
Situation Calculus Planning

- Idea: Represent the **planning problem** using first-order logic
  - Situation calculus lets us reason about **changes** in the world
  - Use theorem proving to show (“prove”) that a sequence of actions will lead to a desired result
  - When applied to a world state / situation

Situation Calculus Planning, cont.

- **Initial state**: a logical sentence about (situation) $S_0$
- **Goal state**: usually a conjunction of logical sentences
- **Operators**: descriptions of how the world changes as a result of the agent’s actions:
  - Result($a,s$) names the situation resulting from executing action $a$ in situation $s$.
- Action sequences are also useful:
  - Result’($l,s$): result of executing list of actions ($l$) starting in $s$
Situation Calculus Planning, cont.

- **Initial state:**
  \[ \text{At(Home, } S_0) \land \neg \text{Have(Milk, } S_0) \land \neg \text{Have(Bananas, } S_0) \land \neg \text{Have(Drill, } S_0) \]

- **Goal state:**
  \[ (\exists s) \text{At(Home, } s) \land \text{Have(Milk, } s) \land \text{Have(Bananas, } s) \land \text{Have(Drill, } s) \]

- **Operators:**
  \[ \forall (a, s) \text{Have(Milk, Result(a, s)) } \iff ((a=\text{Buy(Milk)} \land \text{At(Grocery, } s)) \lor (\text{Have(Milk, } s) \land a \neq \text{Drop(Milk)}) \]

- **Action sequence Result'(l, s):**
  \[ (\forall s) \text{Result'([ ]. } s) = s \]
  \[ (\forall a, p, s) \text{Result'([a|p]s) = Result'(p, Result(a, s))} \]

Situation Calculus, cont.

- **Solution:** a **plan** that when applied to the **initial state** gives a situation satisfying the **goal query:**
  \[ \text{At(Home, Result'(p, } S_0)) \]
  \[ \land \text{Have(Milk, Result'(p, } S_0)) \]
  \[ \land \text{Have(Bananas, Result'(p, } S_0)) \]
  \[ \land \text{Have(Drill, Result'(p, } S_0)) \]

- **Thus we would expect a plan (i.e., variable assignment through unification) such as:**
  \[ p = [\text{Go(Grocery)}, \text{Buy(Milk)}, \text{Buy(Bananas)}, \text{Go(HardwareStore)}, \text{Buy(Drill)}, \text{Go(Home)}] \]
Situation Calculus: Blocks World

- Example situation calculus rule for blocks world:
  - clear(X, Result(A,S)) ↔
    - clear(X, S) ∧
      - (¬(A=Stack(Y,X) ∨ A=Pickup(X)))
      ∧
      - (A=Stack(Y,X) ∧ ¬(holding(Y,S)))
      ∨
      - (A=Stack(Y,X) ∧ ¬(handempty(S) ∧ ontable(X,S) ∧ clear(X,S))))
    ∨
    - (A=Stack(X,Y) ∧ holding(X,S) ∧ clear(Y,S))
    ∨
    - (A=Unstack(Y,X) ∧ on(X,Y,S) ∧ clear(Y,S) ∧ handempty(S))
    ∨
    - (A=Putdown(X) ∧ holding(X,S))

- English translation: a block is clear if
  - (a) in the previous state it was clear AND we didn’t pick it up or stack something on it successfully, or
  - (b) we stacked it on something else successfully, or
  - (c) something was on it that we unstacked successfully, or
  - (d) we were holding it and we put it down.

Situation Calculus Planning: Analysis

- Fine in theory, but:
  - Problem solving (search) is exponential in the worst case
  - Resolution theorem proving only finds a proof (plan), not necessarily a good plan

- So what can we do?
  - Restrict the language
    - Blocks world is already pretty small…
  - Use a special-purpose algorithm (a planner) rather than general theorem prover
Basic Representations for Planning

- Classic approach first used in the STRIPS planner circa 1970
- **States** represented as conjunction of ground literals
  - \( \text{at(Home)} \land \neg \text{have(Milk)} \land \neg \text{have(bananas)} \ldots \)
- Goals are conjunctions of literals, but may have variables*
  - \( \text{at(?x)} \land \text{have(Milk)} \land \text{have(bananas)} \ldots \)
- Don’t need to fully specify state
  - Un-specified: either don’t-care or assumed-false
  - Represent many cases in small storage
  - Often only represent **changes in state** rather than entire situation
- Unlike theorem prover, not finding whether the goal is true, but is there a sequence of actions to attain it

*generally assume **∃**

Operator/Action Representation

- **Operators** contain three components:
  - **Action description**
  - **Precondition** - conjunction of positive literals
  - **Effect** - conjunction of positive or negative literals which describe how situation changes when operator is applied
- Example:
  - \( \text{Op[Action: Go(there), Precond: At(here) \land Path(here,there), Effect: At(there) \land \neg At(here)] } \)
  - All variables are **universally** quantified
  - Situation variables are implicit
    - **Preconditions** must be true in the state immediately before operator is applied
    - **Effects** are true immediately after
Blocks World Operators

- Classic basic operations for the blocks world:
  - stack(X,Y): put block X on block Y
  - unstack(X,Y): remove block X from block Y
  - pickup(X): pickup block X
  - putdown(X): put block X on the table

- Each will be represented by
  - Preconditions
  - New facts to be added (add-effects)
  - Facts to be removed (delete-effects)
  - A set of (simple) variable constraints (optional!)

Example: stack

preconditions(stack(X,Y), [holding(X), clear(Y)])
deletes(stack(X,Y), [holding(X), clear(Y)]).
adds(stack(X,Y), [handempty, on(X,Y), clear(X)])
constraints(stack(X,Y), [X≠Y, Y≠table, X≠table])
Blocks World Operators II

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precond</th>
<th>Add</th>
<th>Delete</th>
<th>Constr</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack(X,Y)</td>
<td>[holding(X), clear(Y)]</td>
<td>[handempty, on(X,Y), clear(X)]</td>
<td>[holding(X), clear(Y)]</td>
<td>[X≠Y, Y≠table, X≠table]</td>
</tr>
<tr>
<td>unstack(X,Y)</td>
<td>[on(X,Y), clear(X), handempty]</td>
<td>[holding(X), clear(Y)]</td>
<td>[handempty, clear(X), on(X,Y)]</td>
<td>[X≠Y, Y≠table, X≠table]</td>
</tr>
<tr>
<td>pickup(X)</td>
<td>[ontable(X), clear(X), handempty]</td>
<td>[holding(X)]</td>
<td>[ontable(X), handempty, clear(X)]</td>
<td>[holding(X)]</td>
</tr>
<tr>
<td>putdown(X)</td>
<td>[holding(X)]</td>
<td>[ontable(X), handempty, clear(X)]</td>
<td>[holding(X)]</td>
<td>[X≠table]</td>
</tr>
</tbody>
</table>

STRIPS Planning

- STRIPS maintains two additional data structures:
  - **State List** - all currently true predicates.
  - **Goal Stack** – push-down stack of goals to be solved, current goal at top.

- If current goal is not satisfied by present state:
  - Examine **add lists** of operators
  - Push operator and preconditions list on stack (and call them subgoals)

- When current goal *is* satisfied, POP it from stack.

- When an operator is on top stack
  - Record the application of that operator on the plan sequence
  - Use the operator’s add and delete lists to update current state.
Typical Blocks World Planning Problem

Initial state:
- clear(a)
- clear(b)
- clear(c)
- ontable(a)
- ontable(b)
- ontable(c)
- handempty

Goal:
- on(b,c)
- on(a,b)
- ontable(c)

A plan:
- pickup(b)
- stack(b,c)
- pickup(a)
- stack(a,b)

Another Blocks World Planning Problem

Initial state:
- clear(a)
- clear(b)
- clear(c)
- ontable(a)
- ontable(b)
- ontable(c)
- handempty

Goal:
- on(a,b)
- on(b,c)
- ontable(c)

A plan:
- pickup(a)
- stack(a,b)
- unstack(a,b)
- putdown(a)
- pickup(b)
- stack(b,c)
- pickup(a)
- stack(a,b)
- ...

A
B
C
A
B
C
Goal Interactions

- Simple planning assumes that goals are **independent**
  - Each can be solved separately and then the solutions concatenated
- Let’s look at when that fails

![Initial state](image1)

![Goal state](image2)

Goal Interactions

- The “Sussman Anomaly”: classic **goal interaction problem**
  - Solving on(A,B) first (by doing unstack(C,A), stack(A,B))
  - Solve on(B,C) second (by doing unstack(A,B), stack(B,C))
- Solving on(B,C) first will be undone when solving on(A,B)
- Classic STRIPS can’t handle this (minor modifications can do simple cases)

![Initial state](image3)

![Goal state](image4)
State-Space Planning

• We initially have a space of situations or world states
  • Where you are, what you have, ...

• Find plan by searching situations to reach goal

• **Progression planner**: searches forward
  • From initial state to goal state

• **Regression planner**: searches backward from goal
  • Works iff operators have enough information to go both ways
  • Ideally leads to reduced branching: planner is only considering things that are relevant to the goal

Planning Heuristics

• Need an **admissible** heuristic to apply to planning states
  • Estimate of the distance (number of actions) to the goal

• Planning typically uses **relaxation** to create heuristics
  • Ignore all or some selected preconditions
  • Ignore delete lists: Movement towards goal is never undone
  • Use state abstraction (group together “similar” states and treat them as though they are identical) – e.g., ignore fluents*
  • Assume subgoal independence (use max cost; or, if subgoals actually are independent, sum the costs)
  • Use pattern databases to store exact solution costs of recurring subproblems

*an aspect of the world that changes - R&N 266
Plan-Space Planning

- Alternative: **search through space of plans**, not situations
- Start from a **partial plan**; expand and refine until a complete plan that solves the problem is generated
- **Refinement operators** add constraints to the partial plan and modification operators for other changes
- We can still use STRIPS-style operators:
  - \text{Op(ACTION: RightShoe, PRECOND: RightSockOn, EFFECT: RightShoeOn)}
  - \text{Op(ACTION: RightSock, EFFECT: RightSockOn)}
  - \text{Op(ACTION: LeftShoe, PRECOND: LeftSockOn, EFFECT: LeftShoeOn)}
  - \text{Op(ACTION: LeftSock, EFFECT: LeftSockOn)}

  can result in a partial plan of \([\text{RightShoe, LeftShoe}]\) 😃

Partial-Order Planning
Partial-Order Planning

- A **linear planner** builds a plan as a **totally ordered sequence** of plan steps.

- A **non-linear planner (aka partial-order planner)** builds up a plan as a set of steps with some temporal constraints:
  - E.g., S1 < S2 (step S1 must come before S2).

- Partially ordered plan (POP) **refined** by either:
  - adding a new **plan step**, or
  - adding a new **constraint** to the steps already in the plan.

- A POP can be **linearized** (converted to a totally ordered plan) by topological sorting*
  *

```
* from search - R&N 223
```

Least Commitment

- Non-linear planners embody the principle of **least commitment**:
  - Only choose actions, orderings, and variable bindings that are absolutely necessary.
  - Leave non-absolutely-necessary decisions till later.
  - Avoids early commitment to decisions that don’t really matter.

- A **linear** planner always chooses to add a plan step in a particular place in the sequence.

- A **non-linear** planner chooses to add a step and possibly some temporal constraints.
Non-Linear Plan: Steps

• A non-linear plan consists of
  (1) A set of steps \( \{S_1, S_2, S_3, S_4, \ldots \} \)

  Each step has an operator description, preconditions and post-conditions

  (2) A set of causal links \( \{ \ldots (S_i, C, S_j) \ldots \} \)

  (One) goal of step \( S_i \) is to achieve precondition \( C \) of step \( S_j \)

  (3) A set of ordering constraints \( \{ \ldots S_i < S_j \ldots \} \)

  if step \( S_i \) must come before step \( S_j \)

Non-Linear Plan: Completeness

• A non-linear plan consists of
  (1) A set of steps \( \{S_1, S_2, S_3, S_4, \ldots \} \)

  (2) A set of causal links \( \{ \ldots (S_i, C, S_j) \ldots \} \)

  (3) A set of ordering constraints \( \{ \ldots S_i < S_j \ldots \} \)

• A non-linear plan is complete iff
  • Every step mentioned in (2) and (3) is in (1)
  • If \( S_j \) has prerequisite \( C \), then there exists a causal link in (2) of the form \( (S_i, C, S_j) \) for some \( S_i \)
  • If \( (S_i, C, S_j) \) is in (2) and step \( S_k \) is in (1), and \( S_k \) threatens \( (S_i, C, S_j) \)
    (makes \( C \) false), then (3) contains either \( S_k < S_i \) or \( S_j < S_k \)
The Initial Plan

Every plan starts the same way

Initial State

S1: Start

Goal State

S2: Finish

Trivial Example

Operators:

- Op(ACTION: RightShoe, PRECOND: RightSockOn, EFFECT: RightShoeOn)
- Op(ACTION: RightSock, EFFECT: RightSockOn)
- Op(ACTION: LeftShoe, PRECOND: LeftSockOn, EFFECT: LeftShoeOn)
- Op(ACTION: LeftSock, EFFECT: leftSockOn)

Steps: {S1:[Op(Action:Start)],

S2:[Op(Action:Finish,

Pre: RightShoeOn^LeftShoeOn)]}

Links: {}

Orderings: {S1<S2}
Solution

POP Constraints and Search Heuristics

- Only add steps that reach a not-yet-achieved precondition

- Use a least-commitment approach:
  - Don’t order steps unless they need to be ordered

- Honor causal links $S_1 \rightarrow S_2$ that protect a condition $c$:
  - Never add an intervening step $S_3$ that violates $c$
  - If a parallel action threatens $c$ (i.e., has the effect of negating or clobbering $c$), resolve that threat by adding ordering links:
    - Order $S_3$ before $S_1$ (demotion)
    - Order $S_3$ after $S_2$ (promotion)
Partial-Order Planning Example

- Goal: Have milk, bananas, and a drill
• Add three actions to achieve basic goals
• Use initial state to achieve the "Sells" preconditions
• Bold links are causal, regular are just ordering constraints
Resolving Threats

- Cannot resolve threat to At(Home) preconditions of both Go(HWS) and Go(SM).
- Must backtrack to supporting At(x) precondition of Go(SM) from initial state At(Home) and support it instead from the At(HWS) effect of Go(HWS).
- Since Go(SM) still threatens At(HWS) of Buy(Drill), must promote Go(SM) to come after Buy(Drill). Demotion is not possible due to causal link supporting At(HWS) precondition of Go(SM)
- Add Go(Home) action to achieve At(Home)
- Use At(SM) to achieve its precondition, and
- Order it after Buy(Milk) and Buy(Banana) to resolve threats to At(SM)

Partial Planning Exercise

```
START
\~Tilled  Weeds
Swimmable  Playable  Walkable
FINISH
```

ordering link
causal link

```
\~Weeds  Tilled  Weeds
Lay Sod  Spray  \~Tilled
Playable  \~Weeds  Tilled
Lay Walkway  Rototill
Walkable  Tilled  \~Walkable  Swimmable
```

Swimmable  Playable  Walkable
Weeds ~Tilled ~Tilled ~Walkable Swimmable
Rototill Tilled Tilled Tilled
Dig Pool ~Walkable Swimmable
Swimmable Playable Walkable
FINISH

Your Turn...

START ~Tilled Weeds
Rototill Tilled Tilled
Dig Pool ~Walkable Swimmable
Swimmable Playable Walkable
FINISH

Tilled Weeds
Spray ~Weeds ~Weeds
Lay Sod Playable ~Weeds
Lay Walkway Walkable

ordering link causal link
ordering link causal link
Real-World Planning Domains

- Real-world domains are complex
- Don’t satisfy assumptions of STRIPS or partial-order planning methods
- Some of the characteristics we may need to deal with:
  - Modeling and reasoning about resources
  - Representing and reasoning about time
  - Planning at different levels of abstractions
  - Conditional outcomes of actions
  - Uncertain outcomes of actions
  - Exogenous events
  - Incremental plan development
  - Dynamic real-time replanning

\{ Scheduling \} \{ Planning under uncertainty \} \{ HTN planning \}
Hierarchical Decomposition

• Hierarchical decomposition, or hierarchical task network (HTN) planning, uses **abstract operators** to **incrementally** decompose a planning problem from a **high-level goal** statement to a **primitive plan network**

• **Primitive operators** represent actions that are **executable**, and can appear in the final plan

• **Non-primitive operators** represent **goals** (equivalently, **abstract actions**) that require further decomposition (or **operationalization**) to be executed

• There is no “right” set of primitive actions: One agent’s goals are another agent’s actions!

---

HTN Operator: Example

```
OPERATOR decompose
PURPOSE: Construction
CONSTRAINTS:
    Length (Frame) <= Length (Foundation),
    Strength (Foundation) > Wt(Frame) + Wt(Roof)
    + Wt(Walls) + Wt(Interior) + Wt(Contents)
PLOT: Build (Foundation)
    Build (Frame)
    PARALLEL
        Build (Roof)
        Build (Walls)
    END PARALLEL
    Build (Interior)
```
HTN Planning: Example

![HTN Planning Diagram]

HTN Operator Representation

- Russell & Norvig explicitly represent causal links
- Can also be computed dynamically by using a model of preconditions and effects
- Dynamically computing causal links means that actions from one operator can safely be interleaved with other operators, and subactions can safely be removed or replaced during plan repair
- R&N representation only includes variable bindings
- Can actually introduce a wide array of variable constraints
Truth Criterion

- Determining whether a formula is true at a particular point in a partially ordered plan is, in the general case, NP-hard
- Intuition: there are exponentially many ways to linearize a partially ordered plan
- In the worst case, if there are N actions unordered with respect to each other, there are N! linearizations
- Ensuring soundness of truth criterion requires checking the formula under all possible linearizations
- Use heuristic methods instead to make planning feasible
- Check later to be sure no constraints have been violated

Truth Criterion in HTN Planners

- Heuristic:
  1. Prove that there exists one possible ordering of the actions that makes the formula true
  2. But don’t insert ordering links to enforce that order
- Such a proof is efficient
  - Suppose you have an action A1 with a precondition P
  - Find an action A2 that achieves P (A2 can be initial world state)
  - Make sure there is no action necessarily between A2 and A1 that negates P
- Applying this heuristic for all preconditions in the plan can result in infeasible plans
Increasing Expressivity

- Conditional effects
  - Instead of different operators for different conditions, use a single operator with conditional effects
  - Move (block1, from, to) and MoveToTable (block1, from) collapse into one Move (block1, from, to):
    - `Op(ACTION: Move(block1, from, to),
      PRECOND: On (block1, from) ^ Clear (block1) ^ Clear (to)
      EFFECT: On (block1, to) ^ Clear (from) ^ ~On(block1, from) ^ ~Clear(to) when to<>Table`  
  - There’s a problem with this operator: can you spot what it is?

- Negated and disjunctive goals
- Universally quantified preconditions and effects

Reasoning About Resources

- Introduce numeric variables that can be used as measures
- These variables represent resource quantities, and change over the course of the plan
- Certain actions may produce (increase the quantity of) resources
- Other actions may consume (decrease the quantity of) resources
- More generally, may want different types of resources
  - Continuous vs. discrete
  - Sharable vs. nonsharable
  - Reusable vs. consumable vs. self-replenishing
Other Real-World Planning Issues

- Conditional planning
- Partial observability
- Information gathering actions
- Execution monitoring and replanning
- Continuous planning
- Multi-agent (cooperative or adversarial) planning

Planning Summary

- Planning representations
  - Situation calculus
  - STRIPS representation: Preconditions and effects
- Planning approaches
  - State-space search (STRIPS, forward chaining, …)
  - Plan-space search (partial-order planning, HTN, …)
  - Constraint-based search (GraphPlan, SATplan, …)
- Search strategies
  - Forward planning
  - Goal regression
  - Backward planning
  - Least-commitment
  - Nonlinear planning