A.I. Planning

- We bring together the ideas of state space search and richer ways to represent and reason with knowledge
  - So diachronic reasoning (between states) again;
  - deliberation before action;
  - a process of search (actions are only being tried out on mental representations so they are 'undo-able');
  - assume we develop a complete plan of action before any execution of the plan
  - however, this makes unrealistic assumptions (e.g. ignores timeliness of action).
- We look only at *domain-independent planners* — there are also domain-dependent ones, e.g. tailored to route planning

The STRIPS language: a restricted language for planners

Typically, for efficiency, a language of restricted expressiveness is used to encode actions, states and goals.

- **States:** conjunctions of function-free ground literals (i.e. predicates applied to constants, possibly negated). (In many systems, a state description need not be a complete description of the state. Hence, it would in fact denote a set of states.)
- **Goals:** conjunctions of function-free literals (i.e. predicates applied to constants or variables, the variables taken as existentially quantified). (Will not generally be complete descriptions, so similarly denote set of goal states.)

The STRIPS language, cont'd

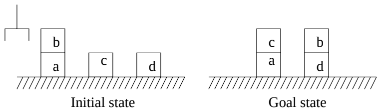
- **Operators:** 3 components
  - *action*: a symbol that identifies the action.
  - *precondition*: a conjunction of function-free atoms (i.e. non-negative predicates applied to constants or variables).
  - *effect*: a conjunction of function-free literals (i.e. positive or negative predicates applied to constants or variables). Sometimes the *effect* is divided into two wffs: the *add list* (the +ve effects) and the *delete list* (the -ve effects).

  Any variables in operators are taken to be universally quantified. An operator with variables in it denotes a family of actions: when inserting an action into a plan, the variables must be instantiated, so we know what object the action is being done to.

Applicability of an operator

- An operator o is applicable in state s if we can instantiate o's variables in such a way as to make every precondition of o true in s.
- The new state that results from applying (suitably instantiated) o to s contains all the +ve literals in o's effect (the add-list) and all of the literals of s, except for those that are -ve literals in o's effect (the delete-list).

Example: the blocks world



Initial state:

$$\text{on(b, a)} \wedge \text{ontable(a)} \wedge \text{ontable(c)} \wedge \text{ontable(d)} \wedge \text{armempty} \wedge \text{clear(b)} \wedge \text{clear(c)} \wedge \text{clear(d)}$$

Goal state:

$$\text{on(c, a)} \wedge \text{on(b, d)} \wedge \text{ontable(a)} \wedge \text{ontable(d)} \wedge \text{clear(c)} \wedge \text{clear(b)}$$

The blocks world

Operators:

Op(	ACTION:	stack(x, y),
	PRECOND:	clear(y) $\wedge$ holding(x),
	EFFECT:	$\neg$clear(y) $\wedge$ $\neg$holding(x) $\wedge$ armempty $\wedge$ on(x, y) $\wedge$ clear(x))

Op(	ACTION:	unstack(x, y),
	PRECOND:	on(x, y) $\wedge$ clear(x) $\wedge$ armempty,
	EFFECT:	$\neg$on(x, y) $\wedge$ $\neg$armempty $\wedge$ $\neg$clear(x) $\wedge$ holding(x) $\wedge$ clear(y))

Op(	ACTION:	pickup(x),
	PRECOND:	clear(x) $\wedge$ ontable(x) $\wedge$ armempty,
	EFFECT:	$\neg$ontable(x) $\wedge$ $\neg$armempty $\wedge$ $\neg$clear(x) $\wedge$ holding(x))

Op(	ACTION:	putdown(x),
	PRECOND:	holding(x),
	EFFECT:	$\neg$holding(x) $\wedge$ ontable(x) $\wedge$ clear(x) $\wedge$ armempty)

Knowledge representation problems

- The **qualification problem**: It is hard to define preconditions: the circumstances under which an action is *guaranteed* to work, e.g. to pick up an object requires that it not be too heavy. Leaving things out gives danger of execution failure.
- The **ramification problem**: It is hard to define effects: especially implicit consequences of actions, e.g. moving a container moves its contents.
- The **frame problem**: To give a logically correct specification of an operator (one that allows the right inferences), we would have to specify not only what changes (effects) but that everything else doesn't change. In STRIPS operators, we only say what changes. It is implicit in the specialised 'theorem-provers' (planning algorithms) we use that all else remains unchanged.

Using straightforward state space search

- Current state := initial state; use operators to compute successor states, place onto an agenda, pick one as current state, etc., etc., until a state that satisfies goal is reached.
- But, we have horrendous state space
  - Large number of nodes: many possible sets of literals
  - High branching factor: many operators will be applicable
- And, search tree will be huge
  - Heuristics choose among states, not among operators.
  - The goal state description is treated only as a test; it is not used to help select operators.

Exploiting the logical representations

- Compare:
  - **Progression planning**: plan from initial state to goal — Many literals in initial state, so many applicable operators
  - **Regression planning**: plan from goal to initial state — fewer literals in goals, so fewer applicable operators
- Use regression planning:
  - Logical goal state representations can direct deliberations: only select operators whose effects include one of the literals to be achieved.

Exploiting the logical representations

- May also be more efficient if we can use **problem decomposition**/ **divide-and-conquer**: work on bits of the problem, then combine partial solutions. The fact that planning is goal-driven (see regression planning above), and that goals (and subgoals from preconditions) are conjunctions facilitates this.
- Will be more efficient if problem is perfectly decomposable into manageable independent subproblems. Planning for real-world problems is *nearly decomposable*: there is limited interaction between subproblems. So we solve as if perfectly decomposable, and then fix. (Puzzles, on the other hand, tend to be hard precisely because designed to be barely decomposable.)

STRIPS: an early regression planner

(Fine details, e.g. of variable instantiation, omitted for brevity.)

Uses the initial state and a goal stack, which initially contains only the user-supplied goal.

- If top of stack is a conjunction, decompose it, i.e. push each conjunct that is not true in current state onto stack (in some order). (If all conjuncts are true, not only will nothing be pushed, but conjunct can be popped.)
  — Each subproblem (subgoal) will be solved in the order they appear on the stack. Also, since conjuncts are tested repeatedly until all true, this is STRIPS' naive way of dealing with goal interaction: if an already solved subproblem has been undone, the undone goal is solved again.

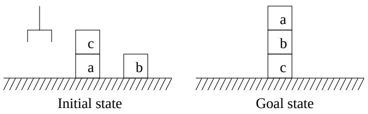
STRIPS algorithm, cont'd

- When top of stack is a single-literal goal:
  - If top of stack is true in current state, pop stack.
  - Else, find an operator whose effects achieve the goal. Replace goal by action name and push the preconditions.
- When top of stack is an action name (its preconditions must have been solved and popped):
  - Apply its effects to the current state

STRIPS: problems

Planners like STRIPS were non-optimal and incomplete

E.g. the Sussman anomaly — unsolvable by simpler planners; solved non-optimally by STRIPS:



Irrespective of which conjunct is solved first, work will be done, undone and done again.

E.g. solving on(a, b) first gives a solution:

unstack(c, a), putdown(c), pickup(a), stack(a, b), unstack(a, b), putdown(a), pickup(b), stack(b, c), pickup(a), stack(a, b)

Alternative planners

- A major cause of the problems is the assumption that the order in which actions are found during planning should be their ordering in the plan, i.e. the order in which they will be executed.

- So current day planners exploit three ideas. Two we have discussed (use of goals in action selection; and the divide-and-conquer strategy).

- The third: we break the connection between the order in which actions are found during planning and the order in which they will be executed: allow ourselves to add actions to the plan wherever they are needed, rather than in an unbroken sequence from initial state (or in reverse from goal).

Alternative planners, cont'd

- More specifically, modern planners:
  - use **regression** planning rather than **progression** planning
  - search **plan space** rather than **state space**
  - build **partial-order** or **non-linear** plans rather than **total order** or **linear** plans
  - use the principle of **least-commitment**.

Plan space versus state space

- So far we've looked at **state space** planners: they search through the state space. We repeatedly apply an operator to a current world state to get a new world state, looking for a path of actions (and states) from initial to goal states.

- Alternative is **plan space** planners: they search through the plan space. Each node is a (possibly) partial plan. Start with a very simple partial plan. Repeatedly expand the partial plan (add things to it) until it is no longer partial and it solves the problem.

- In this search space, the nodes are plans, so the operators will not be the STRIPS operators; they will be **plan refinement operators**, e.g. adding a step to the plan, imposing an ordering constraint on steps within the plan, i.e. they further constrain the plan. (Some people talk of **constraint-posting**.)

Partial-order versus total order planning

- A planner that can represent plans in which some steps are ordered w.r.t. each other but others are unordered is a **partial-order** or **non-linear** planner.
- The planners we have looked at so far cannot represent plans in this way: even when steps do not need to be ordered, they are still ordered (the ordering coming from the order in which they were determined during planning): **total-order** or **linear** planners.
- Plan space planning is amenable to partial-order planning in a way that state space planning is generally not.

Least commitment

- Partial-order planning is then usually subject to the principle of **least-commitment**: don't make any choices that you don't have to, e.g. don't order actions unless they must be ordered. Postponement of commitment means, if search needs to backtrack, then fewer decisions to be undone.

We look at POP, an example of such a planner, neglecting fine details again (e.g. variable instantiation).

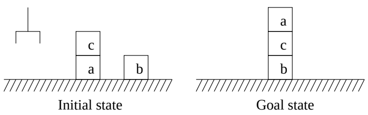
POP plans

A (partial) plan comprises:

- A set of plan *steps*. Each is one of the instantiated (STRIPS) operators.
- A set of *ordering constraints*: $S_i \prec S_j$ means step S_i must occur sometime before S_j (not necessarily immediately before).
- A set of *causal links*: $S_i \xrightarrow{c} S_j$ means step S_i achieves precondition c of step S_j.

POP initial plan

Initial plan: comprises two dummy steps, Start and Finish. Start is a step with no preconditions, only effects: the effects are the initial state of the world. Finish is a step with no effects, only preconditions: the preconditions are the goal.



Plan(STEPS:	$\{$S1: Op(	ACTION:	Start,
		EFFECT:	clear(b) $\wedge$ clear(c) $\wedge$ on(c, a) $\wedge$ ontable(a) $\wedge$ ontable(b) $\wedge$ armempt
	S2: Op(	ACTION:	Finish,
		PRECOND:	on(c, b) $\wedge$ on(a, c))$\}$,
ORDERINGS:	$\{$S1 $\prec$ S2$\}$,		
LINKS:	$\{\}$)		

POP refinement operators: take us from a partial plan to a less partial plan.

- **Goal achievement**
  - **Step addition**: Add a new step S_i which has an effect c that can achieve an as yet unachieved precondition of existing step S_j. Also add $S_i \prec S_j$ and $S_i \xrightarrow{c} S_j$ and Start $\prec S_i \prec$ Finish.
  - Use an effect c of an existing step S_i to achieve an as yet unachieved precondition of another existing step S_j. Add just $S_i \prec S_j$ and $S_i \xrightarrow{c} S_j$.
- Causal links must be **protected** from **threats**, i.e. steps that delete (or negate or **clobber**) the protected condition. If S threatens link $S_i \xrightarrow{c} S_j$:
  - **Promote**: add $S \prec S_i$; **or**
  - **Demote**: add $S_j \prec S$

POP solution plans

A solution plan is one in which:

- every precondition of every step is achieved by the effect of some other step and all possible clobberers have been suitably demoted or promoted; and
- there are no contradictions in the ordering constraints, e.g. disallowed is $S_i \prec S_j$ and $S_j \prec S_i$; also disallowed is $S_i \prec S_j$, $S_j \prec S_k$ and $S_k \prec S_i$.

(Note that solutions may still be partially-ordered. This retains flexibility for as long as possible. Only immediately prior to execution will the plan need **linearisation**, i.e. imposition of arbitrary ordering constraints where steps are not yet ordered — and, only then if there is only a single execution agent.)

POP algorithm

1. Make the initial plan, i.e. the one that contains only the Start and Finish steps.
2. Do until you have a solution plan
   - Take an unachieved precondition from the plan; achieve it
   - Resolve any threats using promotion or demotion

Search in the POP algorithm

- Choice points:
  - In goal achievement, a condition c might be achievable by any one of a number of new steps and/or existing steps. (Why try the latter before the former?)
  - Resolve threats by demotion or promotion.
- Unexplored alternatives must be stored on an agenda.
- With the right search strategy, the planner is sound, complete and optimal, but...
- ...since branching factor can still be high-ish, we typically need good heuristics (plus pruning to keep agenda manageable) or a search strategy that is more depth-first. These bring risks of non-optimal plans and incompleteness.

Practical planners

AI planners are made more efficient by:

- *Hierarchical decomposition*: Include *abstract operators* as well as normal STRIPS ones. I.e. operators that are like macros: precompiled plans. Revise the algorithm so that it is allowed to replace an abstract step by its decomposition (which requires you to order its earliest and latest steps and check for threats between the existing plan and the contents of the macro).

- *Hierarchical approximation*: Assign *criticality levels* to STRIPS preconditions. First, build a plan in which you achieve only preconditions of highest criticality. Then successively deal with preconditions of lower criticality levels. (But criticality might depend on context.)

- *Domain-dependence*

Adding expressiveness

STRIPS operators are quite restrictive. At the price of efficiency, can add more expressiveness, e.g.

- *Conditional effects*: within effects, make some of them conditional on state of the world.
  (Advantage over using two operators: more delayed commitment)

- *Logical expressiveness*: Adding negative goals, negative preconditions, disjunctive preconditions, universally quantified preconditions and universally quantified effects have all been done with no great performance penalty.

  Disjunctive effects are harder and make for low efficiency: see later.

Adding expressiveness, cont'd

- *Logical expressiveness*: We still haven't looked closely at introducing function symbols. So our planners have a static and finite universe (no more objects than there are constants). This is no good if there is to be dynamic creation and destruction of objects.

- *Resources*: Resources (money, fuel, time) probably break the static universe condition. But we can handle them without introducing functions in general. We can usually finesse them by giving a numeric treatment. Numbers and numeric functions would have to be allowed into the logic but would receive fixed interpretations, i.e. a special semantics and specialised proof.

Non-classical planning

Our assumptions throughout:

- The planner has complete and certain knowledge of (relevant portions of) the initial world state;

- each action will be executed infallibly;

- there are no other agents in the execution environment whose actions would interfere with execution of the plan.

These are the assumptions of *classical planners*.

Non-classical planners are being built that abandon one or more of these assumptions.

Non-classical planners: two main approaches:

- *Contingency planning* (or *conditional planning*): uses operators with disjunctive effects, hence multiple paths in the plan. At execution time, sense the actual outcome of the step and follow the appropriate path. (Can be an explosion of paths; so must use utility and likelihood to constrain.)

- *Interleaved planning and execution*: Monitor execution of plan and re-invoke planning if execution effects diverge from expected effects. Often detailed planning is deferred and done closer to execution time so more likely to be done for the world as it actually is.

To some extent, the former is for bounded indeterminacy (small, enumerable, anticipatable set of competing effects); the latter is for unbounded indeterminacy (large or infinite or changing set of competing effects). They can be combined.