Planning involves:

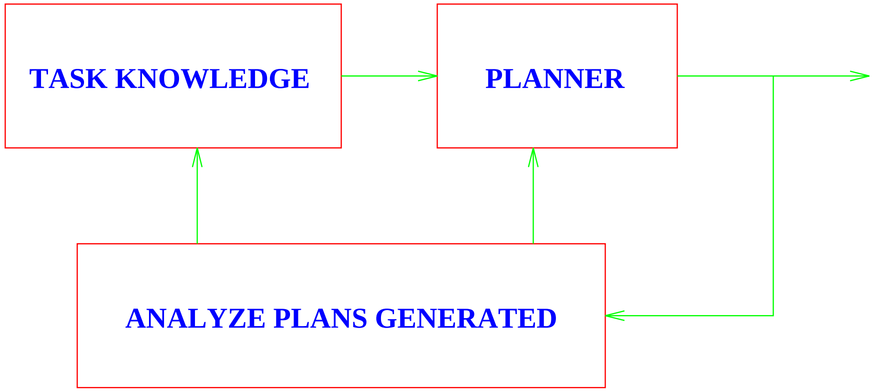
- Given knowledge about a task domain
- Given a problem specified as:
  - ▷ an initial configuration of the state of the "world"
  - ▷ a set of goals to be achieved
- Find a **solution** to the problem, i.e., a *way* to transform the initial configuration into a new state of the world where the goal statement is true.

Many issues to resolve...

A few are:

- What knowledge defines the task domain?
- How to represent the planning action model?
- What is the (sufficient) initial state of the world?
- What are the (prioritized) goals?
- How to acquire domain knowledge efficiently from expert human planners?
- Which algorithm to use to generate the solution plan?
- How to generate plans in a computationally tractable way?
- How to create plans of *good* quality?
- How to scale up to real-world problems?

Reaching Planning Expertise



- Knowledge engineering approaches:
  - ▷ Handcode and refine domain knowledge.
  - ▷ Specify control strategies.
  - ▷ Define knowledge to produce quality plans.
- Machine learning approaches:
  - ▷ **Automate** the interpretation of the planning experience into reusable task knowledge: domain, control, and quality.
  - ▷ Most recent trend: Combine with user's input.

Example: Process Planning Task

[**Pérez 95**, Gil, Hayes]

The essence of AI planning: forming plans to achieve goals.

How can we achieve a desired final configuration given some initial given one?

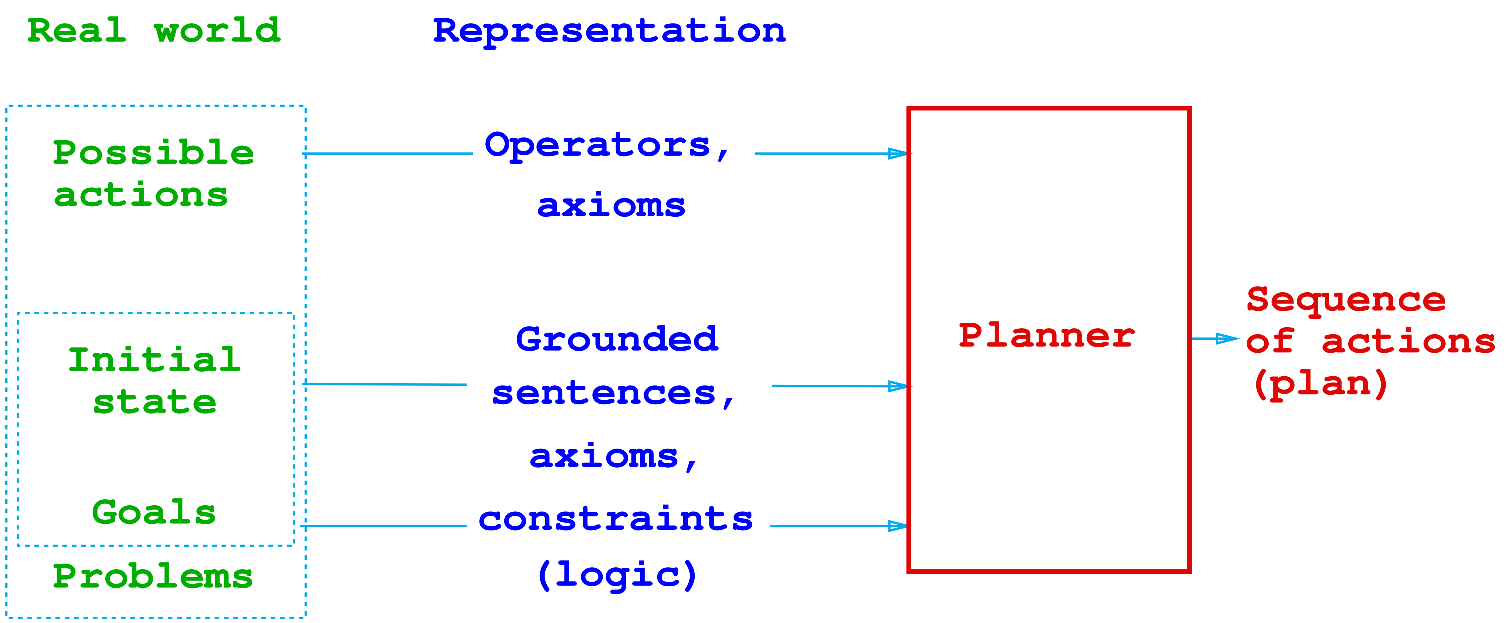
- Different processes (actions).
- Different machines.
- Different tools.
- Parts have orientations.
- Interaction among processes.
- Efficiency, quality, accuracy.
- ...

Planning Domains

Many AI planning domains with different degrees of *realism*:

- Process planning
- Image processing
- Logistics transportation
- Crisis management
- *Generating collection procedures*
- *Bank risk management*
- *Credit card fraud detection*
- Robot navigation
- Machine shop scheduling
- Blocks world
- Puzzles
- Matrix algebra
- Artificial domains
- ...

Example: Task Definition



- State-based representation of the world:
  - ▷ Operators: F : $\textit{states} \rightarrow \textit{states}$ (generalized)
  - ▷ Goal: set of sentences which must be true in the final state.

Example: Problem Representation

Objects

;;machines

(mm2 milling-machine)

(drill7 drill-press) ...

;;tools

(spot-drill3 spot-drill)

(twist-drill5 twist-drill)

(end-mill6 end-mill)

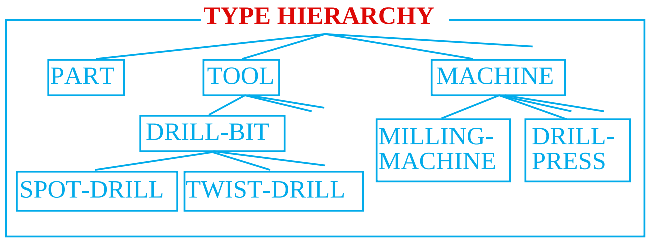
(soluble-oil soluble-oil) ...

;;holding devices

(vise12 vise) ...

;;parts

(part17 part)...



State

(diameter-of-drill-bit twist-drill5 9/64)

(material-of part17 aluminum)

(size-of part17 length 5)

(size-of part17 height 3)

(size-of part17 width 3)...

Goal ((<part> part))

(and (size-of <part> height 2)

(has-spot <part> hole1 side1 1.375 0.25)))

Plan

1. **put-tool-drill** drill7 spot-drill3
2. **put-holding-device-drill** drill7 vise12
3. **clean** part17
4. **put-on-machine-table** drill7 part17
5. **hold** drill7 vise12 part17 side1 side2-side5
6. **drill-in-drill-press** drill7 spot-drill3 part17

...

Example: Action Representation

drill-in-drill-press
<mach>: type drill-press
<drill-bit>: type spot-drill
<device>: type (or vise chuck)
<part>: type part
<hole>: type hole
<side>: type side
Pre: (holding-tool <mach> <drill-bit>)
(holding <mach> <device> <part>)
Add: (has-spot <part> <hole> <side>)
Del: (is-clean <part>)

put-tool-drill
<mach>: type drill-press
<tool>: type drill-bit
Pre: (avail-tool-holder <mach>)
(avail-tool <tool>)
Add: (holding-tool <mach> <tool>)
Del: (avail-tool-holder <mach>)
(avail-tool <tool>)

Many other actions (In Prodigy: more than 100):

- face-mill, remove-tool-from-drill, hold-with-vise,...

Domain Representation

- Operators – rules – with:
  - ▷ Precondition expression – must be satisfied before the operator is applied.
  - ▷ Set of effects – describe how the application of the operator changes the state.
- Precondition expression: propositional, typed first-order predicate logic, negation, conjunction, disjunction, existential and universal quantification, and functions.
- Effects: *add* and *delete* lists.
- Universally quantified effects.
- Conditional effects – dependent on conditions on the state.

Generating a Solution Plan

Several planning **algorithms**:

- Linear planning – Planning with a **stack** of goals.
- Nonlinear planning – Interleaving of goals
  - ▷ State-space search
  - ▷ Plan-space search
- Hierarchical planning
  - ▷ Emphasis on action decomposition/refinement
  - ▷ Very little search

A complex process:

- **Alternative operators to achieve a goal.**
- **Multiple goals that interact.**
- **Efficiency, quality, and accuracy – hard.**

Search Strategies

Using a set of operators:

- Forward chaining – progression
  - ▷ From the initial state,
  - ▷ Apply operators with preconditions true in the state,
  - ▷ Get new states.
- Backward-chaining – regression
  - ▷ From the goal state,
  - ▷ Find operators that can add goal,
  - ▷ Set its preconditions as new goals.
- Partial order – network of constraints among plan steps – no direct reasoning about an explicit state.
- Total order – plan steps are ordered during search – use of a uniquely specified state.

Planning Issues

- Representation of plans
  - ▷ Plans as sequence of state changes
  - ▷ Plans as total orders of steps
  - ▷ Plans as successive levels of refinement
  - ▷ Plans as partial orders of steps
- Conditional actions/effects
- Arbitrary functions computing side-effects
- Temporal reasoning – actions take time – explicit representation of time
- Interleave of planning and execution
- Nondeterministic outcome of actions
- Probabilistic occurrence of external events

▷ Planning as search, i.e., a decision-making process – learn search heuristics

▷ Planning representation – learn efficient, complete, correct domain specifications

Means-ends Analysis

[**Newell and Simon 60s**] [**Ernst and Newell 69**]

GPS Algorithm (*initial-state, goals*)

- If *goals* $\subseteq$ *initial-state*, then return *True*
- Choose a difference d between *initial-state* and *goals*
- Choose an operator o to reduce the difference d
- If no more operators, then return *False*
- *State*=**GPS**(*initial-state*, preconditions(o))
- If *State*, then return **GPS**(apply(o,*State*), *goals*)

STRIPS reduced Algorithm (*initial-state, goals*)
[**Fikes and Nilsson 71**]
Stack= *goals*
State=*initial-state*
Repeat until *Stack*=*empty*
    Case top of *Stack* of
        operator:
            Unmet-preconditions=set of preconditions of o not true in *State*
            If *Unmet-preconditions*= *empty*,
            Then *State*=apply(o,*State*)
            Else Introduce *Unmet-preconditions* into *Stack*
        set of goals:
            If *goals*$\subseteq$*State*, Then remove *goals* from *Stack*
            (*) Introduce goal $g \in goals \mid g \notin initial\text{-}state$ into *Stack*
        single goal:
            If *goal*$\subseteq$*State*, Then remove *goal* from *Stack*
            Else If goal loop, Then backtrack
                Else (*) Select operator $o \mid g \in effects(o)$
                    Introduce o in *Stack*

Linear Planning: Discussion

Advantages:

- Linear planning assumes that goals are independent.
- Reduced search space, because goals are solved one at a time.
- Clearly an advantage if goals are independent.

Disadvantages:

- Linear planning may produce *unoptimal* solutions.
- Linear planning is **incomplete.**

Strict Completeness: A planning algorithm is *strictly complete* if all the solutions to a given problem are included in its search space.

Completeness: A planning algorithm is *complete* if at least one solution to a given problem, when one exists, is included in its search space.

Example: Irreversible Actions


```
(OPERATOR LOAD-ROCKET
 (preconds
  ((<roc> ROCKET)
   (<obj> OBJECT)
   (<loc> LOCATION))
  (and (at <obj> <loc>)
       (at <roc> <loc>)))
 (effects ()
  (add (inside <obj> <roc>))
  (del (at <obj> <loc>))))
```

```
(OPERATOR UNLOAD-ROCKET
 (preconds
  ((<roc> ROCKET)
   (<obj> OBJECT)
   (<loc> LOCATION)))
  (and (inside <obj> <roc>)
       (at <roc> <loc>)))
 (effects ()
  (add (at <obj> <loc>))
  (del (inside <obj> <roc>))))
```

```
(OPERATOR MOVE-ROCKET
 (preconds
  ((<roc> ROCKET)
   (<from-l> LOCATION)
   (<to-l> LOCATION))
  (and (at <roc> <from-l>)
       (has-fuel <roc>)))
 (effects ()
  (add (at <roc> <to-l>))
  (del (at <roc> <from-l>))
  (del (has-fuel <roc>))))
```

Incompleteness of Linear Planning

Initial state:
(at obj1 locA)
(at obj2 locA)
(at ROCKET locA)
(has-fuel ROCKET)

Goal statement:
(and
(at obj1 locB)
(at obj2 locB))

Goal	*Plan*
(at obj1 locB)	(LOAD-ROCKET obj1 locA) (MOVE-ROCKET) (UNLOAD-ROCKET obj1 locB)
(at obj2 locB)	*failure*

Goal	*Plan*
(at obj2 locB)	(LOAD-ROCKET obj2 locA) (MOVE-ROCKET) (UNLOAD-ROCKET obj2 locB)
(at obj1 locB)	*failure*

State-Space Nonlinear Planning

Extend linear planning:

- From **stack** to **set** of goals.
- Include in the search space all possible interleaving of goals.

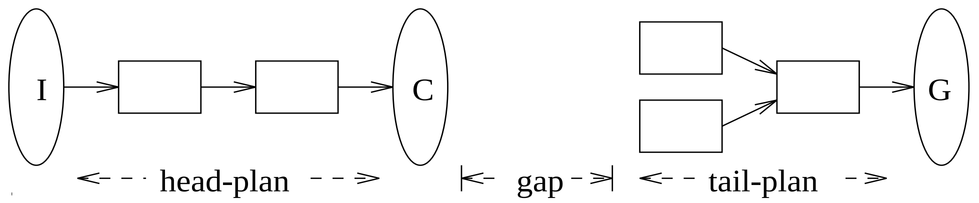
State-space nonlinear planning is **complete**.

Goal	*Plan*
(at obj1 locB)	(LOAD-ROCKET obj1 locA)
(at obj2 locB)	(LOAD-ROCKET obj2 locA)
(at obj1 locB)	(MOVE-ROCKET) (UNLOAD-ROCKET obj1 locB)
(at obj2 locB)	(UNLOAD-ROCKET obj2 locB)

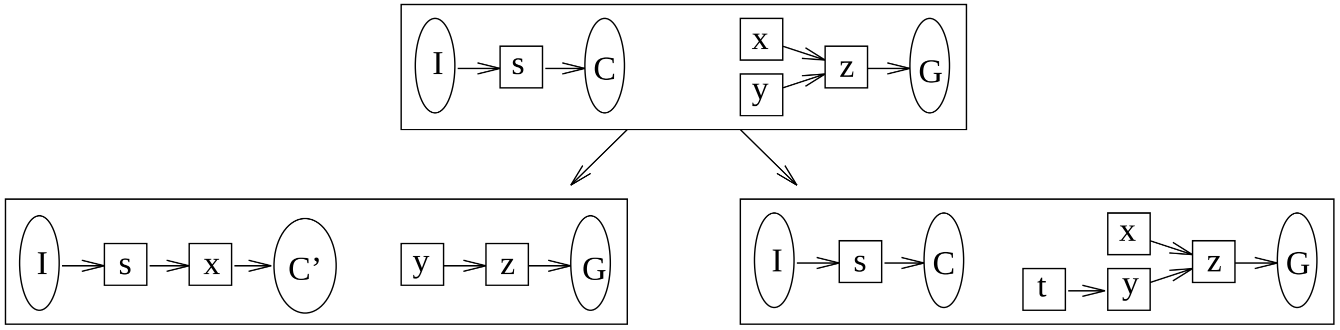
PRODIGY4.0 Planning Algorithm

1. Terminate if the goal statement is satisfied in the current state.
2. Compute the **SET** of *pending goals* $\mathcal{G}$, and the **set** of *applicable operators* $\mathcal{A}$.
   - ▷ A goal is pending if it is a precondition, not satisfied in the current state, of an operator selected to be in the plan to achieve a particular goal.
   - ▷ An operator is applicable when all its preconditions are satisfied in the state.
3. Choose a goal G from $\mathcal{G}$ or select an operator A from $\mathcal{A}$.
4. If G has been chosen, then
   - ▷ *Expand goal* G, i.e., get the set $\mathcal{O}$ of *relevant instantiated operators* that could achieve the goal G,
   - ▷ Choose an operator O from $\mathcal{O}$,
   - ▷ Go to step 1.
5. If an operator A has been selected as directly applicable, then
   - ▷ *Apply* A,
   - ▷ Go to step 1.

Representation of an incomplete plan during search:



Modifying the current plan – children of a search node:



Applying an operator (moving it to the head)

Adding an operator to the tail-plan

Hierarchical Planning

- General-purpose search heuristics do not solve reasonably complex representations of domains
- A well chosen simplification of the representation can improve the performance
- Need to simplify search and representation
- Key idea: Identify levels of abstraction, details.

Example: ABSTRIPS [Sacerdoti, 74]

- Each precondition has a *criticality value*
- Planning algorithm: *incremental refinement*
  - ▷ For *cv* from maximum-criticality-value down to minimum
    - Plan using only preconditions of criticality+*cv* refining previous abstract plan

Other examples:

- NOAH [Sacerdoti, 75] – Nets of action hierarchies
- O-PLAN [Tate 80] – elaborated abstract levels, no search

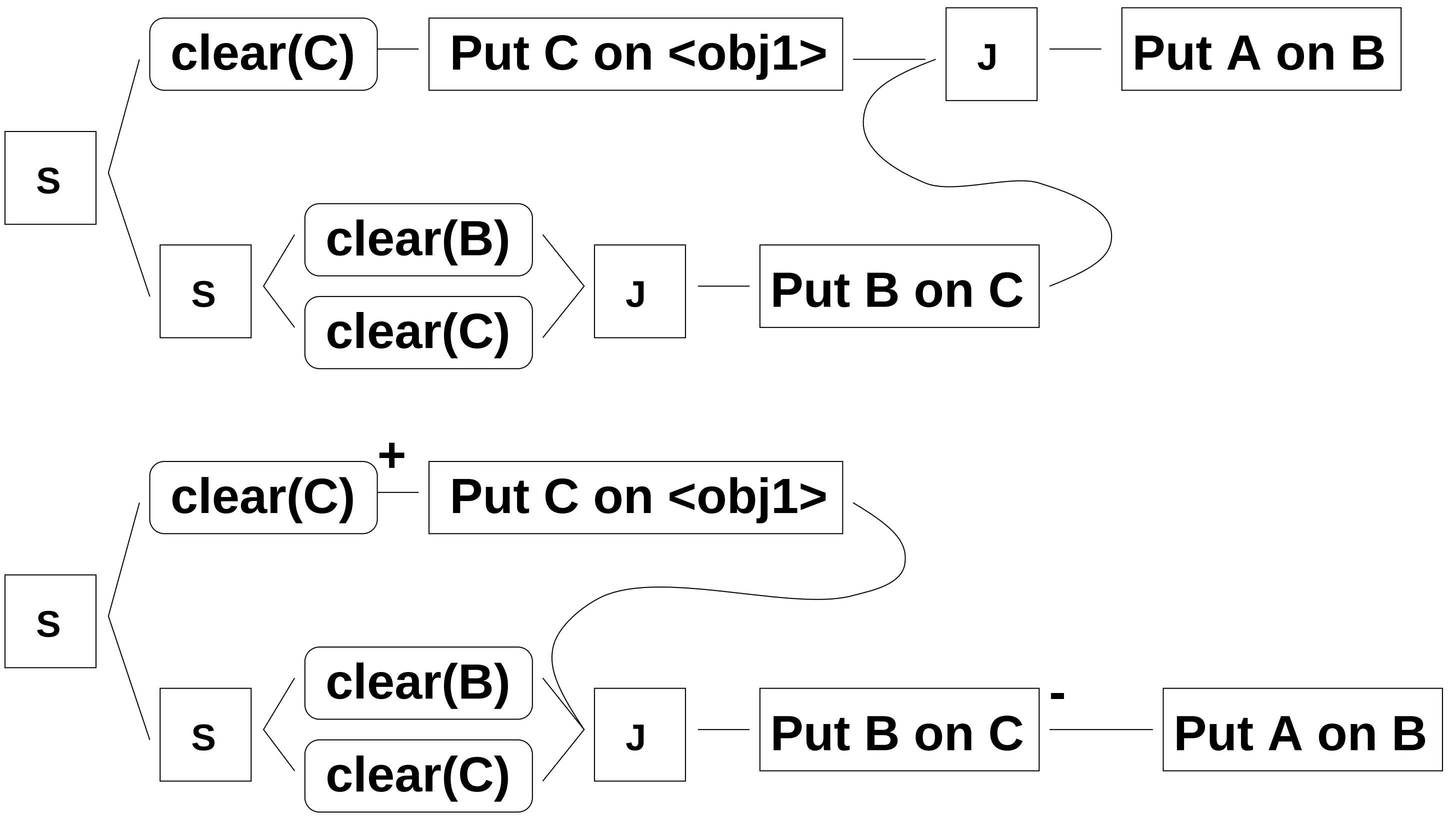
PUSH-THRU-DOOR

Preconditions:
$\{6\}$pushable(box1)$\wedge\{6\}$type(door1,DOOR)$\wedge$ $\{6\}$type(room1,ROOM)$\wedge$
$\{2\}$status(door1,OPEN)$\wedge\{1\}$next-to(box1,door1)$\wedge$ $\{1\}$next-to(ROBOT,box)$\wedge$
$\exists$ room2 [$\{5\}$in-room(box,room2)$\wedge\{5\}$in-room(ROBOT,room2)$\wedge$
$\{6\}$connects(door1,room1,room2)]

Deletions:
at(ROBOT,\$1,\$2)$\wedge$next-to(ROBOT,\$1)$\wedge$at(box1,\$1,\$2)$\wedge$
next-to(box1,\$1)$\wedge$next-to(\$1,box1)$\wedge$in-room(ROBOT,\$1)$\wedge$in-room(box1,\$1)

Additions:
in-room(box1,room2)$\wedge$in-room(ROBOT,room2)$\wedge$next-to(ROBOT,box1)

Example of Planning in NOAH



Several Other Planning Algorithms

- TWEAK [Chapman 87], SNLP [McAllester & Rosenblitt 91] , UCPOP [Penberthy and Weld 92]
  - ▷ Emphasis on plan-space *search*
- NONLIN [Tate 76], O-PLAN [Tate], SIPE [Wilkins 88]
  - ▷ Emphasis on plan decomposition
- UNPOP, Planning and acting [McDermott 78]
- Reactive planning [Georgeff & Lansky 87], [Firby 87], [Hendler & Sanborn 87]
- Action and time [Allen 84] [Dean & McDermott 87]
- Walksat [Selman et al. 92, Kautz & Selman 92, 96]
- Flecs [Veloso & Stone 95]
- Graphplan [Blum & Furst 95]

Plan-Space Partial-Order Nonlinear Planning

SNLP Planning Algorithm [McAllester & Rosenblitt 91]

1. Terminate if the goal set is empty.
2. Select a goal g from the goal set and identify the plan step that needs it, S_{need}.
3. Let S_{add} be a step (operator) that adds g, either a new step or a step that is already in the plan. Add the causal link $S_{add} \xrightarrow{g} S_{need}$, constrain S_{add} to come before S_{need}, and enforce bindings that make S_{add} add g.
4. Update the goal set with **all** the preconditions of the step S_{add}, and delete g.
5. Identify *threats* and resolve the conflicts by adding ordering or bindings constraints.

- A step S_k threatens a causal link $S_i \xrightarrow{g} S_j$ when it occurs between S_i and S_j, and it adds or deletes p.
- Resolve threats by using *promotion*, *demotion*, or *separation*.

State-space and Plan-space

- Planning is NP-hard.
- Two different planning approaches: state-space and plan-space planning

	State-space	*Plan-space*
Commitments in plan step orderings	Yes	No
Therefore, suffer with goal orderings	Yes	No
Therefore, handle goal interactions	Poorly	Efficiently

WHY?

Use of a uniquely specified STATE of the world while planning

In PRODIGY4.0 advantages include:

- Means-ends analysis - plan for goals that reduce the differences between current and goal states.
- Informed selection of operators - select operators that need less planning work than others.
- State useful for learning, generation and match of conditions supporting informed decisions.
- Helpful for generating anytime planning - provide *valid*, executable, plans at any time.
- Probabilistic planning - may be useful to reason about states, events that affect them, and eventual transitions.

Parallel between Commitments - Example

Operator Polish
preconds: ()
adds: polished
deletes: ()

Operator Drill-Hole
preconds: ()
adds: has-hole
deletes: polished

Goal: *polished* and *has-hole* Initial state: empty PRODIGY4.0	Goal: *polished* and *has-hole* Initial state: *polished* SNLP
- plan for goal *polished* - select *Polish* • order *Polish* as first step - plan for goal *has-hole* - select *Drill-Hole* • order *Drill-Hole* $\succ$ *Polish* • *polished* deleted, backtrack - *Polish* $\succ$ *Drill-Hole*	- plan for goal *polished* - select *Initial* state • link *Initial* to *polished* - plan for goal *has-hole* - select *Drill-Hole* • link *Drill-Hole* to *has-hole* • threat - relink *polished* - select *Polish* - link *Polish* to *polished* - *Polish* $\succ$ *Drill-Hole*

Serializability and Linkability

- A set of subgoals is ***serializable*** [Korf]:
  - If there exists some ordering whereby they can be solved sequentially,
  - without ever violating a previously solved subgoal.
- Easily serializable, laboriously serializable [Barrett and Weld].

- A set of subgoals is ***easily linkable***:
  - If, independently of the order by which the planner links these subgoals to operators,
  - it never has to undo those links.
  - Otherwise it is ***laboriously linkable***.

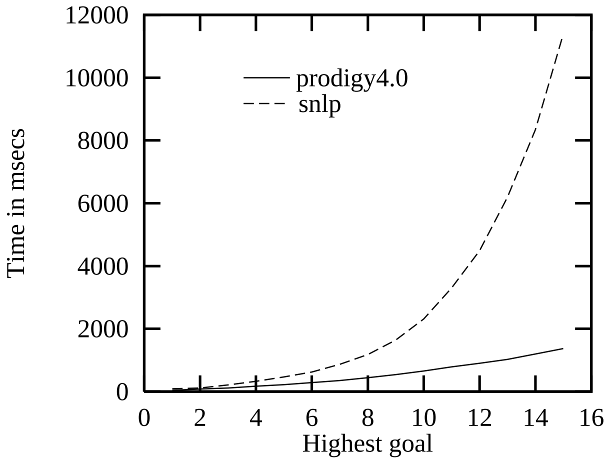
Laboriously Linkable Goals

operator A_i		operator A_*	
preconds	g_*, g_{i-1}	preconds	()
adds	g_i	adds	g_*
deletes	g_*	deletes	()

Initial state: g_*

Goal statement: g_*, g_5

Plan: $A_1, A_*, A_2, A_*, A_3, A_*, A_4, A_*, A_5$



The Importance of the Commitment Choice Comparison

- PRODIGY4.0's algorithm simplifies to the regular expression (**Subgoal Apply***)*.

Two main choices:
- Plan for a goal OR
- Change the state, i.e. simulate execution.

- PRODIGY4.0 uses **state** to determine ...
  - ▷ if the goal state has been reached.
  - ▷ which goals still need to be achieved.
  - ▷ which operators are applicable.
  - ▷ which operators to try first while planning.

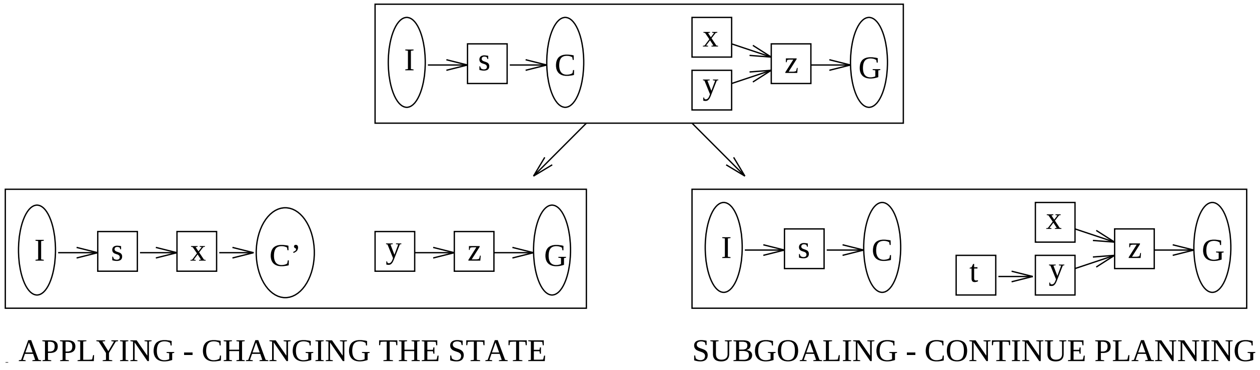
Two Heuristics: SAVTA, SABA

SAVTA: Eager application = **Eager** state changes

Subgoal **A**fter e**V**ery **T**ry to **A**pply

SABA: Eager subgoaling = **Delayed** state changes

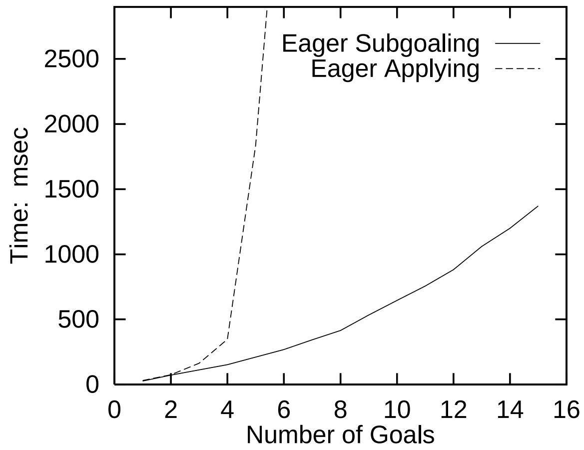
Subgoal **A**lways **B**efore **A**pplying



Eagerly Subgoaling Can Be Better

Operator:	**paint-white** <obj>	**paint-yellow** <obj>	$\cdots$	**paint-black** <obj>
preconds:	(usable white)	(usable yellow)	$\cdots$	(usable black)
adds:	(white <obj>)	(yellow <obj>)	$\cdots$	(black <obj>)
deletes:		(usable white)	$\cdots$	(usable white)
			$\cdots$	(usable yellow)
			$\vdots$	$\vdots$
				(usable brown)

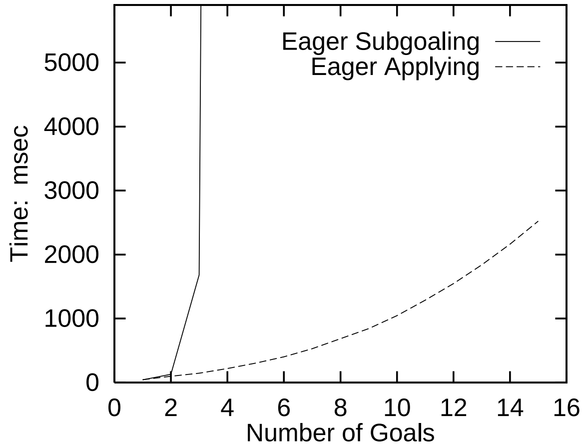
Operator: A_i
preconds: $\{I_i\}$
adds: $\{G_i\}$
deletes: $\{I_j | j < i\}$



Eagerly Applying Can Be Better

Operator:	**paint-with-brush1** <parts> <color>	...	**paint-with-brush8** <parts> <color>
preconds:	(unused brush1)	...	(unused brush8)
adds:	(painted <parts> <color>)	...	(painted <parts> <color>)
deletes:	(unused brush1)	...	(unused brush8)

Operator: A_i
preconds: $\{I_i\}$
adds: $\{< g >\}$
deletes: $\{I_i\}$



FLECS – Flexible Commitment Strategy

Operator:	**designate-roller** <wall> <roller> <color>	**fill-roller** <roller> <color>	**paint-wall** <wall> <roller> <color>
preconds:	(clean <roller>) (needs-painting <wall>)	(clean <roller>) (chosen <roller> <color>)	(ready <wall> <roller> <color>) (filled-with-paint <roller> <color>)
adds:	(ready <wall> <roller> <color>) (chosen <roller> <color>)	(filled-with-paint <roller> <color>)	(painted <wall> <color>)
deletes:		(clean <roller>)	(ready <wall> <roller> <color>) (needs-painting <wall>)

Initial State

(needs-painting wallA)
(needs-painting wallB)
(needs-painting wallC)
(needs-painting wallD)
(needs-painting wallE)
(clean roller1)
(clean roller2)

Goal Statement

(painted wallA red)
(painted wallB red)
(painted wallC red)
(painted wallD green)
(painted wallE green)

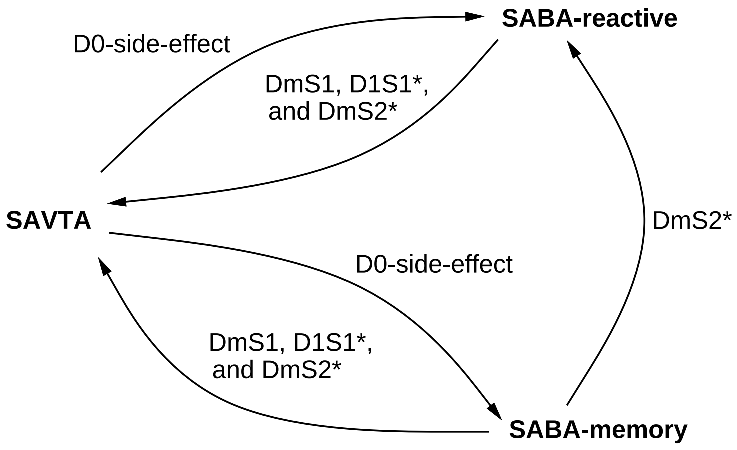
An Optimal Solution

<Designate-Roller wallA roller1 red>
<Designate-Roller wallB roller1 red>
<Designate-Roller wallC roller1 red>
<Fill-Roller roller1 red>
<Paint-Wall wallA roller1 red>
<Paint-Wall wallB roller1 red>
<Paint-Wall wallC roller1 red>
<Designate-Roller wallD roller2 green>
<Designate-Roller wallE roller2 green>
<Fill-Roller roller2 green>
<Paint-Wall wallD roller2 green>
<Paint-Wall wallE roller2 green>

	time(sec)	solution
eager applying	500	no
eager subgoaling	500	no
variable strategy	4	yes

Summary – Planning

- Several different planning algorithms.
- **There is not a planning strategy that is universally better than the others.**
- Even for a particular planning algorithm: **There is no single domain-independent search heuristic that performs more efficiently than others for all problems or in all domains.**



Learning is appropriate for **ANY** planner.