Agent's actions planning

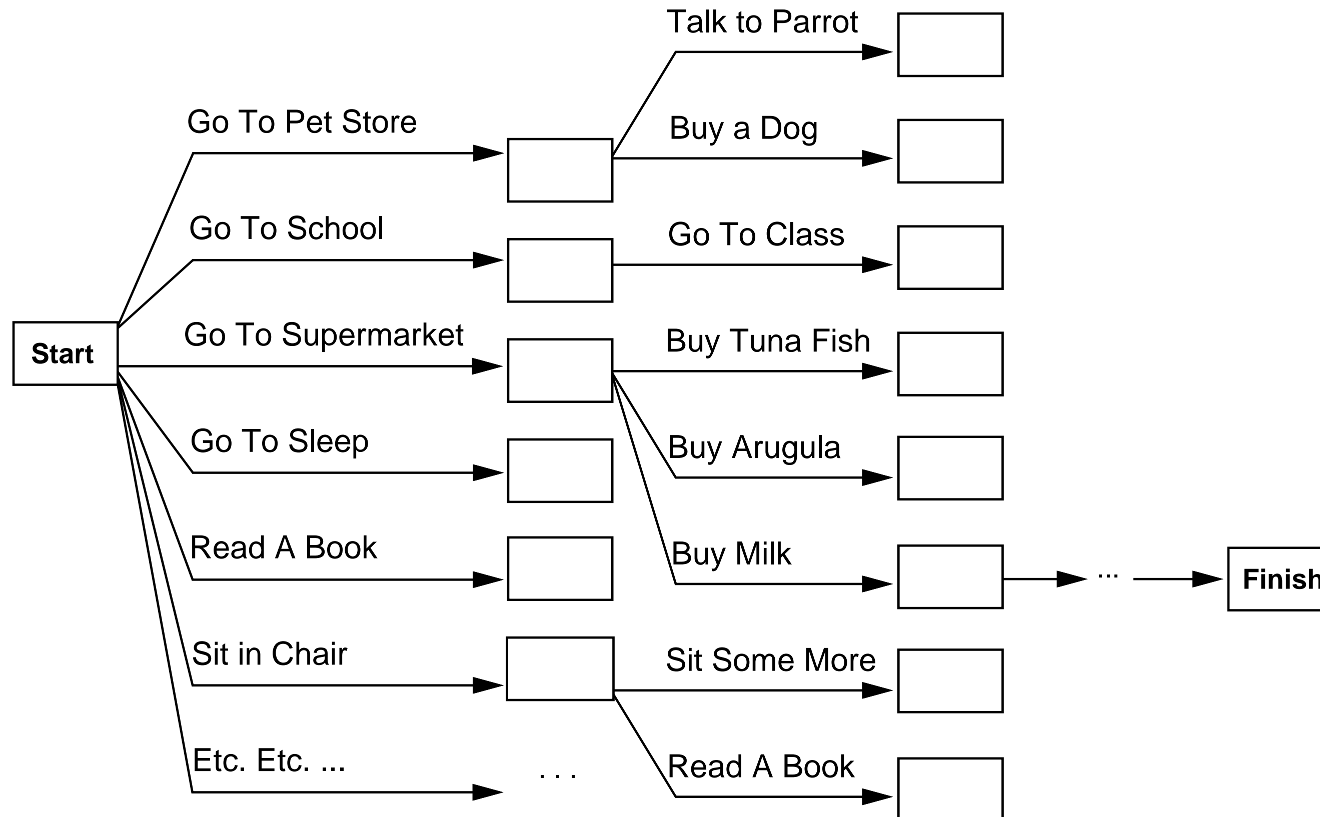
The term **classical action planning** is taken in artificial intelligence to mean the task of determining the sequence of agent's actions, given the information about its initial and final states, and knowledge about actions available to the agent (state transition operators).

In classical planning it is assumed that the agent has full knowledge about its environment, which is **deterministic, finite**, and **static**, which means it does not change except for the effects of the agent's actions. Furthermore the changes occurring in the environment are **discrete** both in the facts and time domain.

Note this is a similar statement to the task of searching in the state space.

Planning as searching

Planning task example: we need to make shopping — buy milk, bananas, etc.



Note:

- state descriptions in the real world are complex,
- the agent has many possible actions to choose from.

Planning as searching — cntd.

Task planning is a perfectly regular case of searching in the state space. It is therefore possible to use all algorithms applicable to such search.

But in the real world the state descriptions are typically complex, and an intelligent agent has many possible actions in her repertoire. The agent trying to build a plan by searching must deal with a combinatorial explosion. At the same time it is difficult to formulate an efficient heuristic, which would allow it to direct the search by choosing the proper actions.

Applying the general search methods does not consider — and cannot make efficient use of — the specific aspect of task planning:

- the actions and their effects typically concern a small number of the world properties,
- the planning goals are also expressed with a small number of features.

It could be advantageous to have a specialized representations and algorithms for planning.

Planning using situation calculus

The monkey and bananas example illustrates a specific, but general, approach to the action planning, according to the following scheme:

situations — are explicitly stated and introduced in formulas as values of the situation variable (**reification** — treating as objects)

result functions
$At(Monkey, P_1, S_0) \wedge At(Monkey, P_2, S_1) \wedge S_1 = Result(Goto(P_1, P_2, S_0))$

effect axioms
$\forall p, s \; At(Monkey, p, Goto(p, s))$
$\forall x, s \; Present(x, s) \wedge Portable(x) \Rightarrow Holding(x, Result(Grab, s))$

frame axioms
$\forall a, x, s \; Holding(x, s) \wedge (a \neq Release) \Rightarrow Holding(x, Result(a, s))$
$\forall p, p_1, p_2, s \; At(Bananas, p, s) \Rightarrow At(Bananas, p, Move(Box, p_1, p_2, s))$

theorem — stated in the form:
$\exists s \; HaveBananas(s)$

wynik — może być uzyskany w postaci:
$Havebananas(Grab(Bananas, Climb(Box, Move(Box, P_2, P_3, Goto(P_2, S_0)))))$

Planning using situation calculus — cntd.

Action planning using situation calculus works. But just as is the case with planning by searching it is inefficient — limited by the efficiency of the theorem proving procedure in the first order predicate calculus. Also note, that the initial state is typically not fully specific, with a complete description of the state of all objects. Only some initial conditions are given — describing the agent and the objects to operate on — in effect defining a large set of states.

What we are dealing with here is the application of a general solution to a specific situation. It does not take advantage of the specific properties of the planning domain. The theorem proving algorithm must compute the solution, but its efficiency in the real world situations is usually inadequate.

Dealing with a specific problem, given its details, we can usually use these details to focus on the appropriate actions. The theorem proving engine does this, but in a way which is proper for theorem proving. The question is whether there exists a representation specialized for action planning.

Special representations for planning

It seems that efficient task planning requires specialized representations, which allow the agent to focus on choosing the appropriate plan steps. So what are the requirements of such a representation:

- the action descriptions should be connected with the elements of the state descriptions,
- selecting the steps needed in the plan should proceed in an arbitrary order, not necessarily in the order of their application, (think: trying to plan a vacation trip, you do not need first to figure out which bus, or a taxi, to take to go to the airport),
- if there are several independent planning subgoals, it should be possible to make independent plans, and then merge them into one consistent plan,
- a specialized language for planning problems would be useful, significantly restricting the statement of the states, goals, and actions.

The STRIPS representation scheme

STRIPS — STanford Research Institute Problem Solver (1970). It is the name of an action planning system for a mobile robot operating in the world of boxes moved around several rooms. The system ran on a minicomputer from its time and was very minimalist. Its representation, however, outlived its time, and is still the basis for the construction of knowledge reprezentation for task planning systems.

The STRIPS representation uses simple logical formulas consisting of ground literals (atomic formulas with no variables, and negations thereof), with no function terms. The only logical connective permitted in the formulas is the conjunction. (Since the conjunction is implied, the formulas can be written as lists of literals.) This makes it possible to process state descriptions by algebraic manipulations, without theorem proving.

- The description of the initial state: a conjunction of positive ground literals.
- The planning goal is also written as a conjunction of literals.
- The representation of actions is more elaborate.

STRIPS — reprezentacja akcji

The representation of actions in STRIPS is an **operator schema**, which is a parametrized operator description, with variables, which in each application are substituted with specific values.

The operator schema consists of:

preconditions — conditions to the applicability as a conjunction of positive literals,

effects of the application of an operator, as a conjunction of literals, both positive and negative, with the positive literals added to, and the negative literals deleted from, the state description.
(The original STRIPS in place of the conjunctions of positive and negative literals had two lists of literals: the *Add list* and the *Delete list*.)

Example operator schema:

At(p), Sells(p,x), Have($$$)

Buy(x,p)

Have(x), ¬Have($$$)

Action: $Buy(x, p)$
Preconditions: $At(p), Sells(p, x), Have(\$\$\$)$
Effects: $Have(x), \neg Have(\$\$\$)$

The STRIPS representation scheme — an example

Let's consider a task of planning some shopping. Two operator schemas are available: $Go(p)$ (go to place p), and $Buy(x, p)$ (buy article x at place p). In the initial state the agent is at home, but also knows some other facts necessary when shopping, ie. that the desired articles (milk and bananas) can be bought at a supermarket. The objective is to purchase them.

$$\begin{aligned} S_0 &= \{At(Home), Sells(SM, Milk), Sells(SM, Bananas)\} \\ G &= \{Have(Milk), Have(Bananas)\} \end{aligned}$$

Action:	$Go(p_1, p_2)$	Action:	$Buy(x, p)$
Precond.:	$At(p_1)$	Precond.:	$At(p), Sells(p, x), Have(\$\$\$)$
Effects:	$\neg At(p_1), At(p_2)$	Effects:	$Have(x), \neg Have(\$\$\$)$

We should not miss the fact, that such representation would only allow the agent to buy one item, provided that she had $\$\$\$$ at all, which the initial state specification does not mention.

The STRIPS representation scheme — limitations

Having to express all facts only with ground positive literals restricts the STRIPS language significantly. For example, to properly write the Go operator, it arguments must state where from and where to the agent is going. Only the first representation below is permitted, while the second single-argument Go operator is impossible to write in STRIPS.

Action:	$Go(p_1, p_2)$	Action:	$Go(p)$	
Precond.:	$At(p_1)$	Precond.:	$\{\}$	//none
Effects:	$\neg At(p_1), At(p_2)$	Effects:	$\neg At(*), At(p)$	

The constructs like in the single-argument Go are often useful and would not be hard to implement. That they have not been included in STRIPS is a matter of its general simplicity rather than a conscious design decision.

More planning domain representations

The original STRIPS was created with early computing technology. Nevertheless it was used in a real mobile robot, moving around a laboratory, analyzing environment images from its camera, and planning and executing actions in real time. Out of necessity its representation was very restricted. Perhaps it should now be extended?

Even though contemporary mobile processing hardware is way more powerful, the logic and theorem proving algorithms bring in a level of complexity which we still would rather avoid. However, some extensions to the original STRIPS are possible, permitting more flexible descriptions, but without full logic apparatus.

For example, the logic of STRIPS assumes the CWA (closed world assumption), where any condition not explicitly mentioned in the state description is taken to be false. Consequently, the initial world description consists only of positive literals, as there is no need to state the negative ones.

STRIPS has been extended to the **Action Description Language (ADL)**. ADL permits negative literals in the state descriptions, formulas with conjunctions and alternatives in the goal statement, and drops the CWA.

The PDDL language

The **Planning Domain Definition Language (PDDL)** is a notation language introduced as a standard for writing planning task specifications. This way, different planning programs can read and work on uniformly specified problems. PDDL is more general than either STRIPS and ADL, but contains sublanguages for both of them.

The PDDL syntax is based on Common Lisp.

The problem description in PDDL consists of two parts, typically stored in two separate files:

- a domain description, which is a dictionary of predicates and action representations,
- a problem instance description, which is: the objects, the initial state description, and the goal specification.

PDDL — an example

The robot *Robby* can move between two rooms and has two grippers which it can use to pick up or drop balls. Initially, all balls and the robot are in the first room. The goal is to have the balls in the second room.

```
(define (domain gripper-strips)
  (:predicates (room ?r)
               (ball ?b)
               (gripper ?g)
               (at-robby ?r)
               (at ?b ?r)
               (free ?g)
               (carry ?o ?g))

  (:action move
   :parameters (?from ?to)
   :precondition (and (room ?from) (room ?to)
                      (at-robby ?from))
   :effect (and (at-robby ?to)
                (not (at-robby ?from))))
```

```
(:action pick
 :parameters (?obj ?room ?gripper)
 :precondition (and (ball ?obj) (room ?room)
                    (gripper ?gripper)
                    (at ?obj ?room) (at-robby ?room)
                    (free ?gripper))
 :effect (and (carry ?obj ?gripper) (not (at ?obj ?room))
              (not (free ?gripper))))

(:action drop
 :parameters (?obj ?room ?gripper)
 :precondition (and (ball ?obj) (room ?room)
                    (gripper ?gripper)
                    (carry ?obj ?gripper)
                    (at-robby ?room))
 :effect (and (at ?obj ?room) (free ?gripper)
              (not (carry ?obj ?gripper)))))
```

```
(define (problem strips-gripper2)
  (:domain gripper-strips)
  (:objects rooma roomb ball1 ball2 left right)
  (:init (room rooma)
         (room roomb)
         (ball ball1)
         (ball ball2)
         (gripper left)
         (gripper right)
         (at-robby rooma)
         (free left)
         (free right)
         (at ball1 rooma)
         (at ball2 rooma))
  (:goal (at ball1 roomb)))
```

Planning strategies

In the simplest case a planning system proceeds directly from the initial state, and generates subsequent, fully instantiated plan steps in order. Such system is termed a **total order progression planner**.

Due to the lack of appropriate heuristics for directing the search, a more common approach has traditionally been to search backward from the goal formula, selecting the agent's actions according the goal statement. This is called **regression planning**.

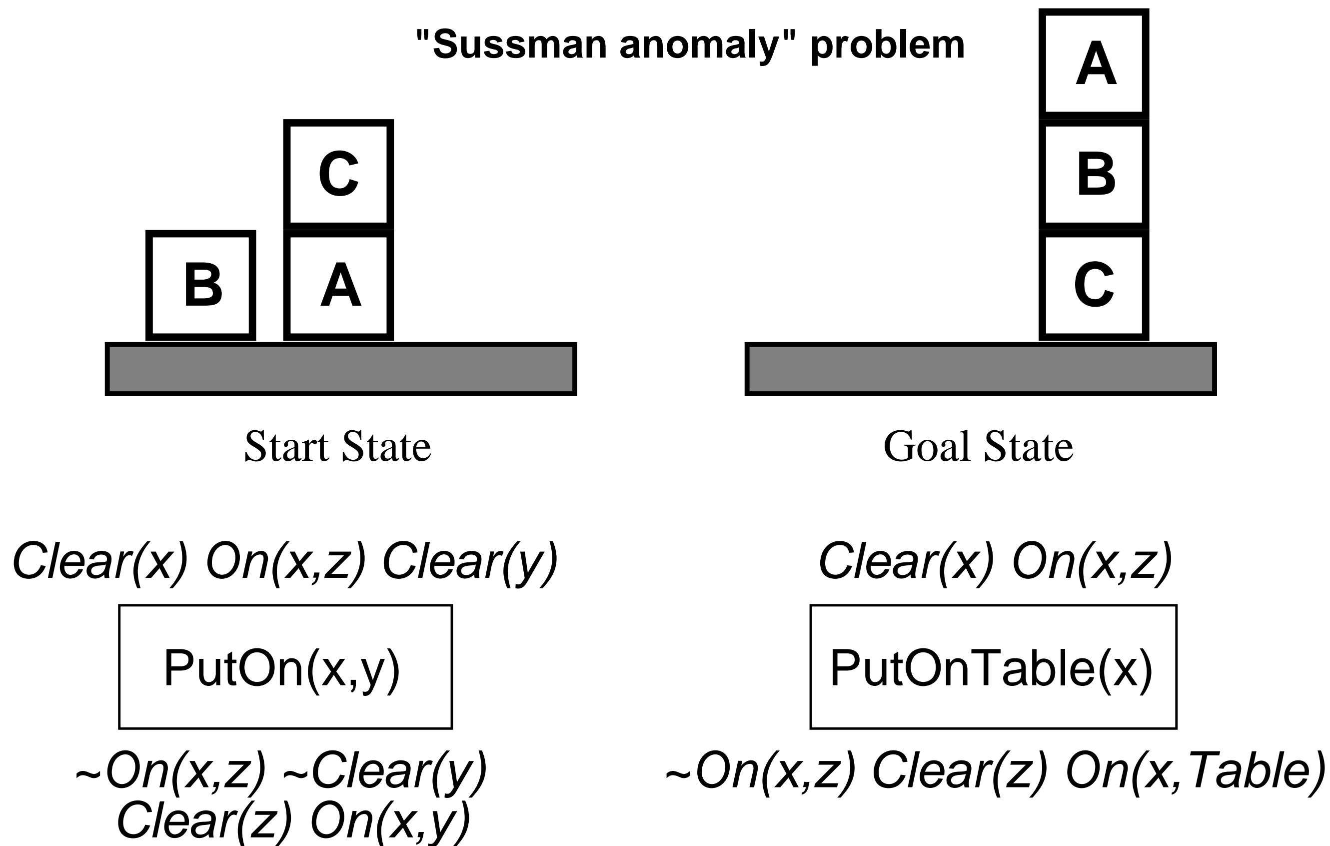
After selecting a terminal action (or one of them, since there may be several literals in the goal formula), one verifies that its preconditions are satisfied, and, if not, then achieving them is added (or replaced) as the new planning goal. Then the next (from the end) operator is selected, and so on, until arriving at a set of requirements which is satisfied by the initial state description.

Using heuristics

However, in practice both forward and backward planning turn out to be inefficient. These approaches can be used only in connection with general (problem domain independent) heuristics. For example, the approach presented earlier, of generating a simplified problem, solving it, and using the found solution cost as an estimate of the cost of the original problem solution, is one example of such heuristic.

Hard cases — interactions between subgoals

The planning algorithm should take advantage of the **divide and conquer** strategy, and build plans separately for independent subgoals, whenever possible. There are cases, however, when planning subgoals are not independent, and solving them separately leads to problems, as in the following **Sussman anomaly**:



Short review

1. Why is it desirable to use specialized representations for agent's action planning, instead of using standard state space, or logical representations?

2. Construct a STRIPS representation to describe the problem of a student, who must pass examinations in two courses, and in order to pass each one she must first study the course material, but while studying one course she loses all knowledge gained previously.

   Use your representation to describe the problem of this student.

3. Try to solve the above student's problem, first using the forward and then the backward strategy.

   If, while working on this, you finally understood the student's basic dilemma (somewhat alike to the Sussman anomaly), then try to think what kind of mechanism(s) would be necessary to solve it.

Planning in the plan space

An alternative representation for the above approach to planning is using the **plan space**. The idea is to construct and subsequently modify a **partial order plan**, which is a triple:

A set of plan steps (fully instantiated operator schemas)
The initial set of plan steps contains two technical operators: the Start operator, whose effects give the full description of the initial state, and the Finish operator, whose precondition state the planning goals. Some plan steps may not have their preconditions satisfied, such conditions are called open.

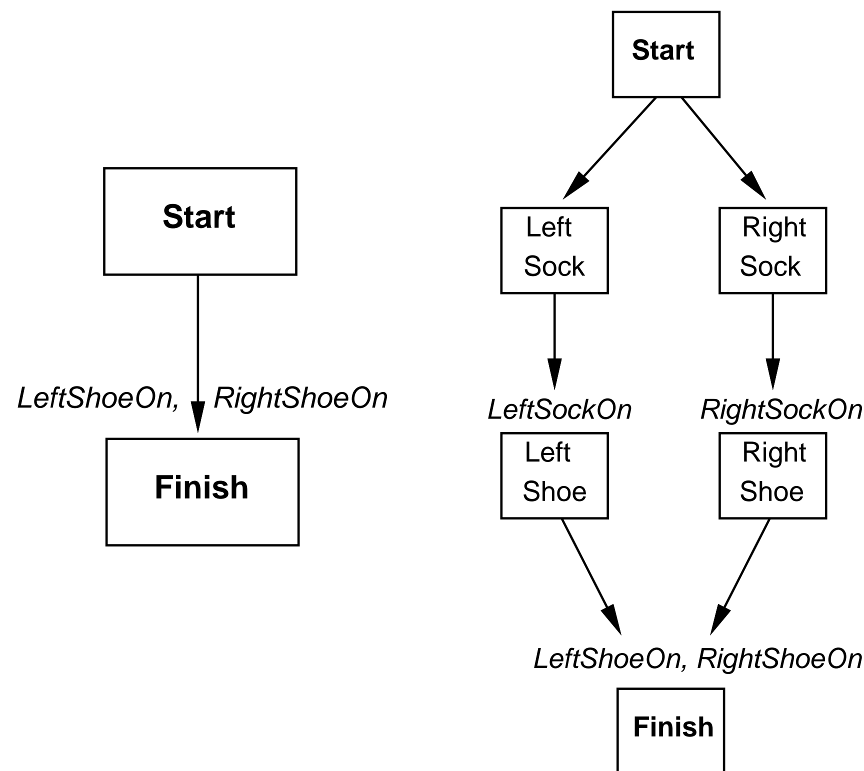
A set of causal links
The causal links connect the plan steps with specific precondition literals of other plan steps. They indicate that a plan step requires executing another plan step to have its precondition satisfied.

A set of chronological orderings
Additional necessary chronological ordering determined by the planner.

A plan is **complete** if all the preconditions of all the plan steps are satisfied by earlier plan steps, and not affected by intermediate steps.

Example: putting on shoes

Let us consider an example: we need to construct a plan for putting on shoes, where the action of putting on a shoe has a precondition of having a sock on first. We could construct a number of alternative total order plans for this task, or the following partial order plan:



The plan on the left is the initial skeletal plan, while the plan on the right is a final complete solution.

Converting partial plans

The operations of converting partial plans:

- adding a link (causal) from an existing plan step to an open condition of another step,
- adding a plan step to satisfy an open condition of another plan step (with a causal link),
- adding a chronological ordering of two steps of the plan.

Example: shopping (1)

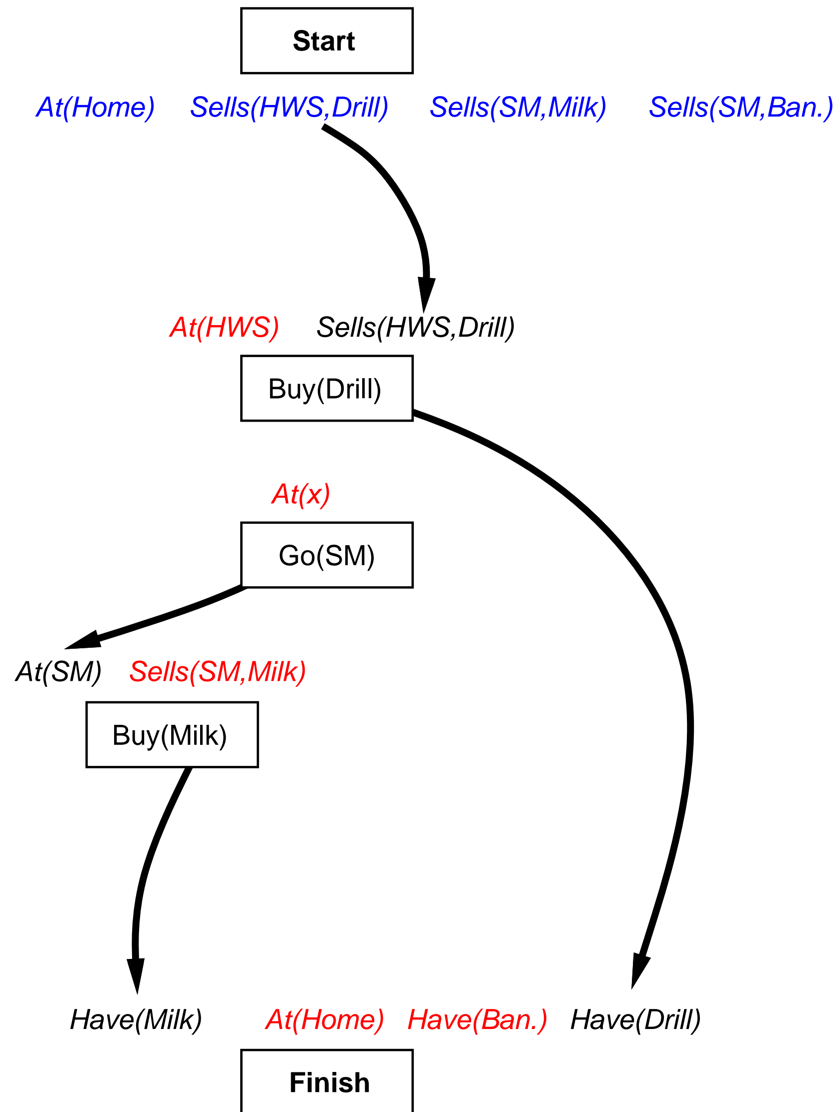
Start

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

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

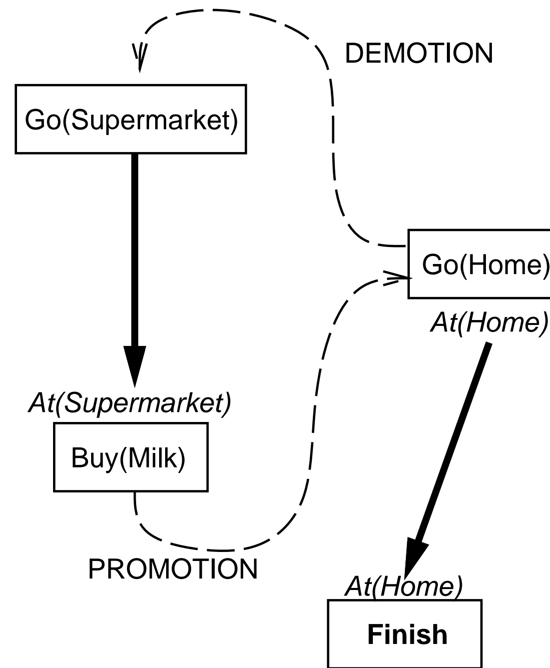
Finish

Example: shopping (2)



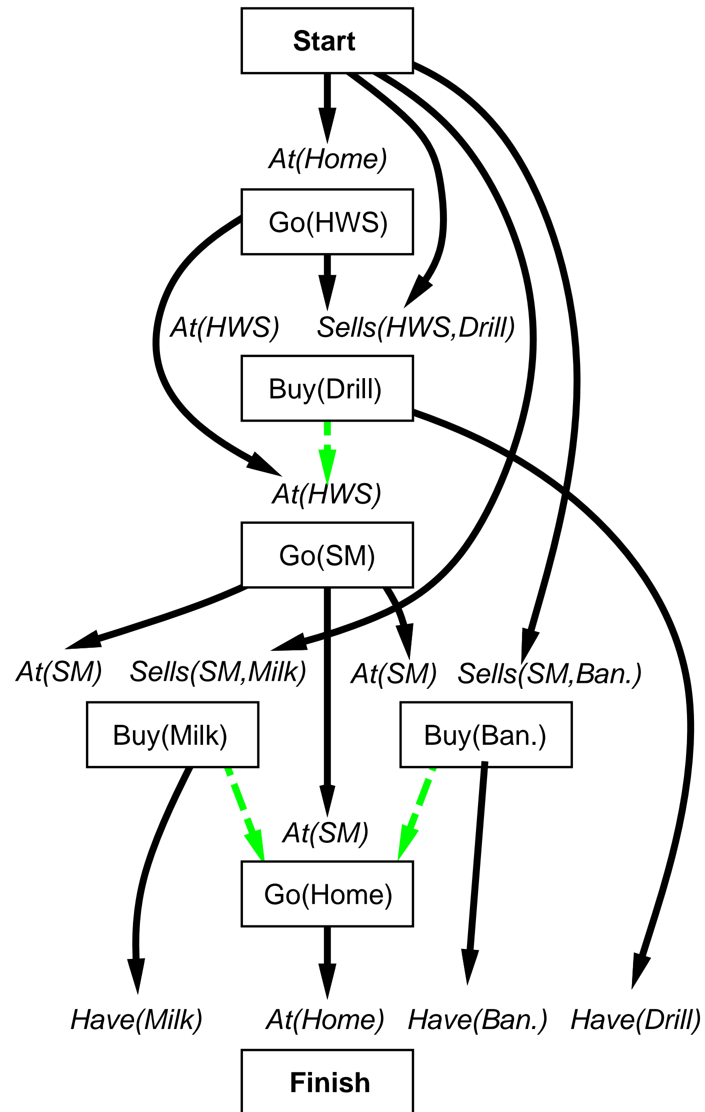
Threats

It is not sufficient to ensure that all the plan steps have their preconditions satisfied by earlier plan steps. This is because a condition satisfied by one plan step (`Go(Supermarket)`) — and needed by another plan step (`Buy(Milk)`), which is indicated by a causal link — could be affected by executing a yet another plan step (`Go(Home)`), if such step was executed between the causally linked plan steps.



Such other step is called a **threat** for the causal link. Threats can be eliminated by introducing additional chronological orderings. The threat step may precede the earlier step of the causal link (called a **demotion**), or come after the latter step of the causal link (a **promotion**).

Example: shopping (3)



The POP planning algorithm

Using the partial plan representation we now introduce the POP (Partial Order Planner) algorithm which constructs complete partial order plans.

function POP(*Initial, Goal, Operators*) **returns** *Plan*
    Plan $\leftarrow$ Make-Minimal-Plan(*Initial, Goal*)
    loop do
        if Solution?(*Plan*) **then return** *Plan*
        $S_{need}, c \leftarrow$ 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

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 $S_{add} \xrightarrow{c} S_{need}$ to Links(*Plan*)

add the ordering constraint $S_{add} \prec S_{need}$ to Orderings(*Plan*)

if S_{add} is a newly added step from *Operators* **then**

add S_{add} to Steps(*Plan*)

add $Start \prec S_{add} \prec Finish$ to Orderings(*Plan*)

procedure Resolve-Threats(*Plan*)

for each $S_{threat} \in$ Steps(*Plan*) that threatens a $S_i \xrightarrow{c} S_j \in$ Links(*Plan*) **do**

choose either

Demotion: Add $S_{threat} \prec S_i$ to Orderings(*Plan*)

Promotion: Add $S_j \prec S_{threat}$ to Orderings(*Plan*)

if not Consistent(*Plan*) **then fail**

end

POP algorithm properties

The POP algorithm is **correct** and **complete**, which means that it returns only correct plans, and does produce one if it only exists (provided its search is implemented using a breadth-first or iterative deepening strategy).

The algorithm has a number of selection points, which can be executed nondeterministically:

- selection of a step S_{need}, c and an open (unsatisfied) condition
- selection of a step S_{add} to satisfy S_{need}, c
- selection of promotion or demotion when resolving a threat

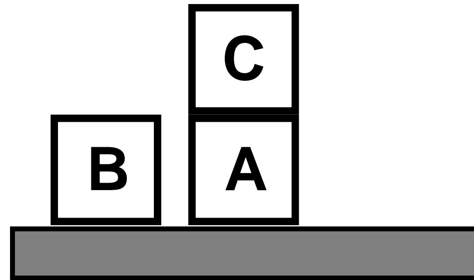
These elements can be backtracking points, if the algorithm encountered one of the following planning failures:

- inability to satisfy an open condition
- unresolvable conflict (plan steps threatening themselves mutually)

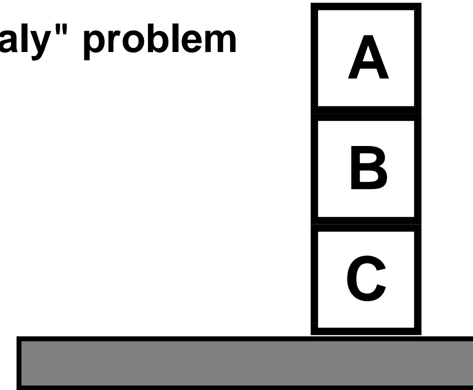
There exist a number of extensions of the above basic version of the algorithm, some of which discussed below.

Example: the blocks world (1)

"Sussman anomaly" problem



Start State



Goal State

Clear(x) On(x,z) Clear(y)

PutOn(x,y)

~On(x,z) ~Clear(y)
Clear(z) On(x,y)

Clear(x) On(x,z)

PutOnTable(x)

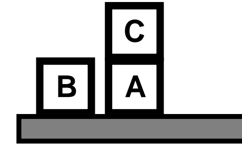
~On(x,z) Clear(z) On(x,Table)

+ several inequality constraints

Example: the blocks world (2)

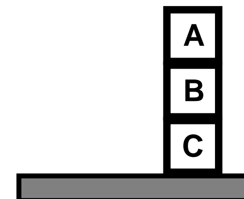
START

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

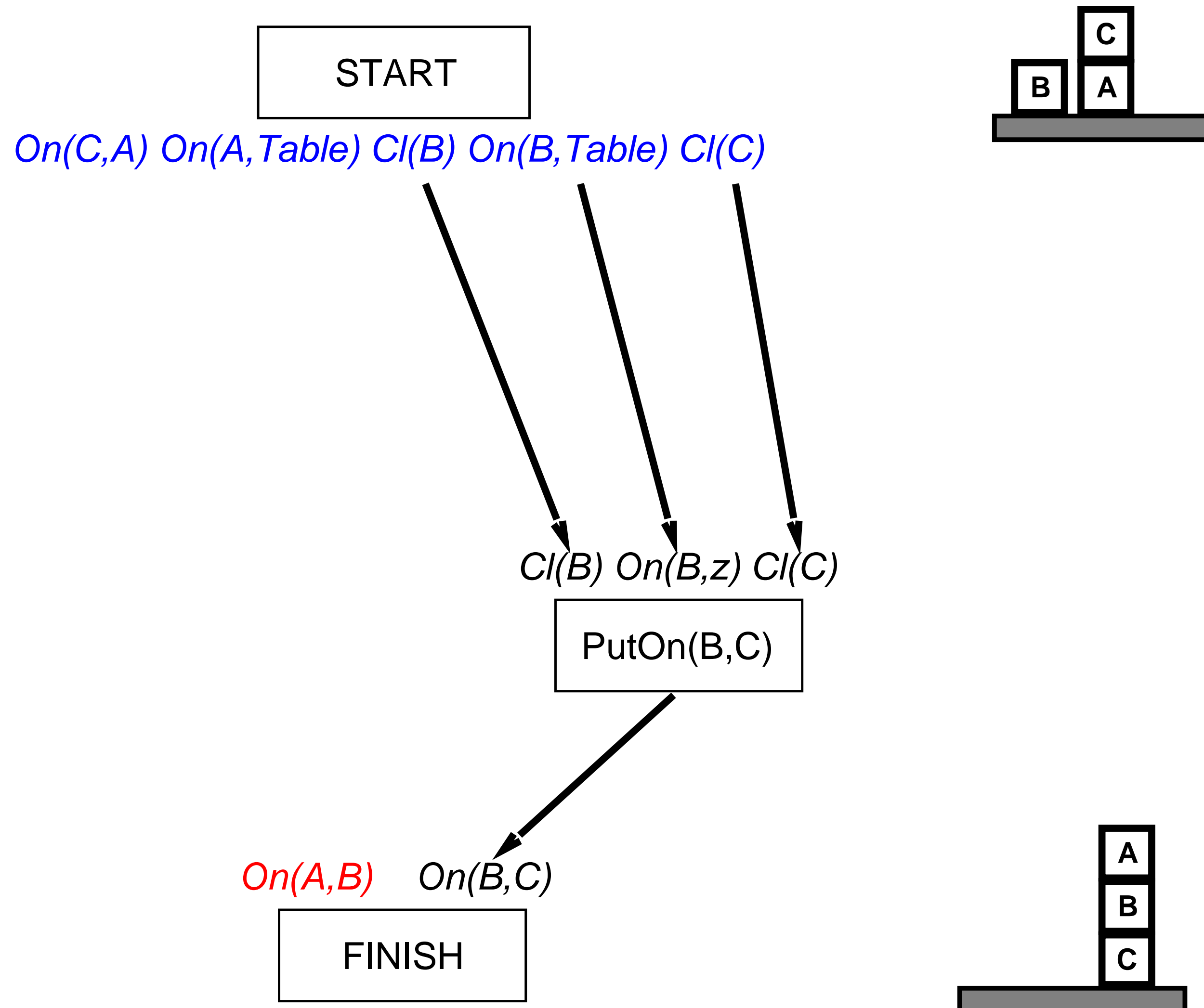


On(A,B) *On(B,C)*

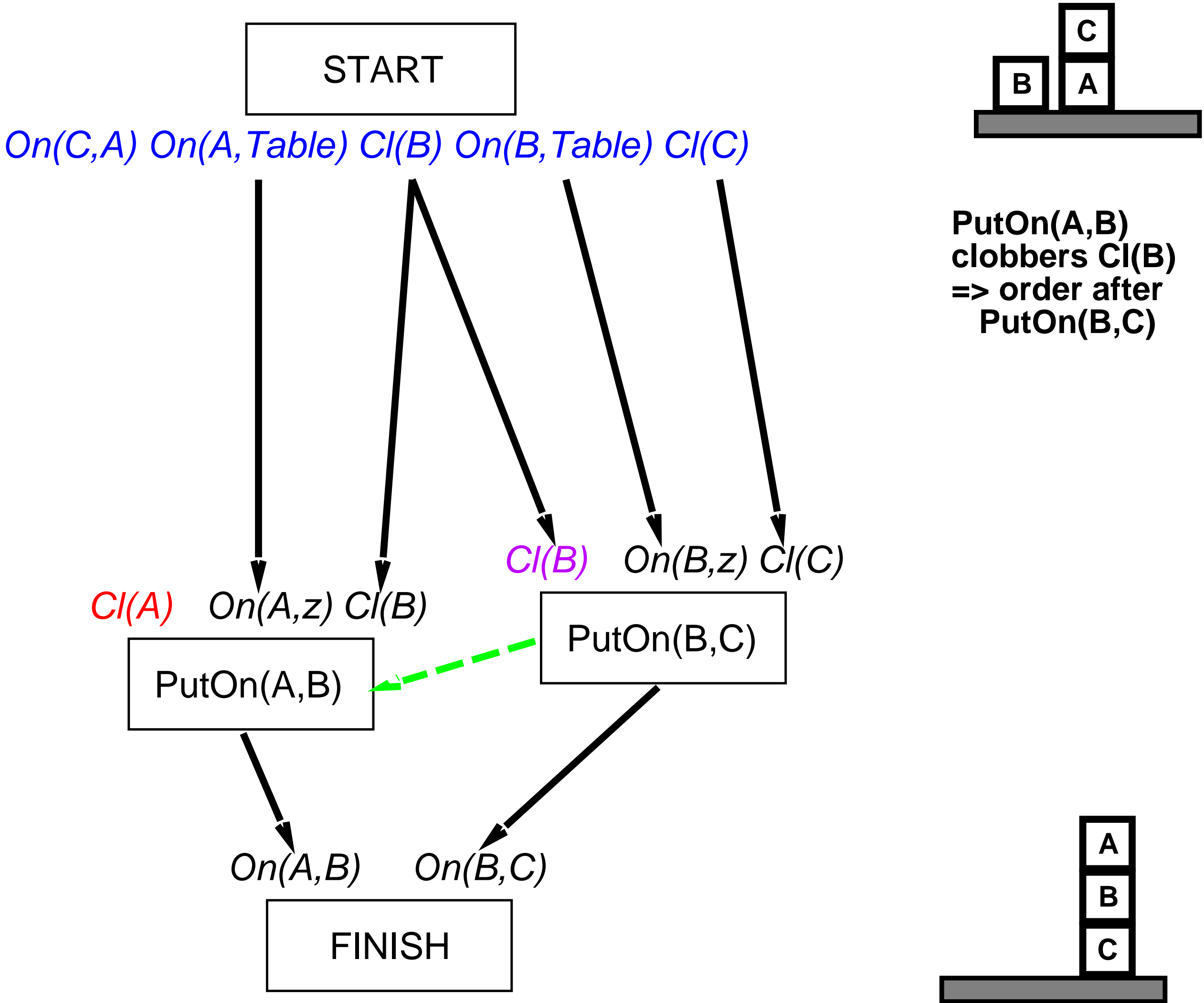
FINISH



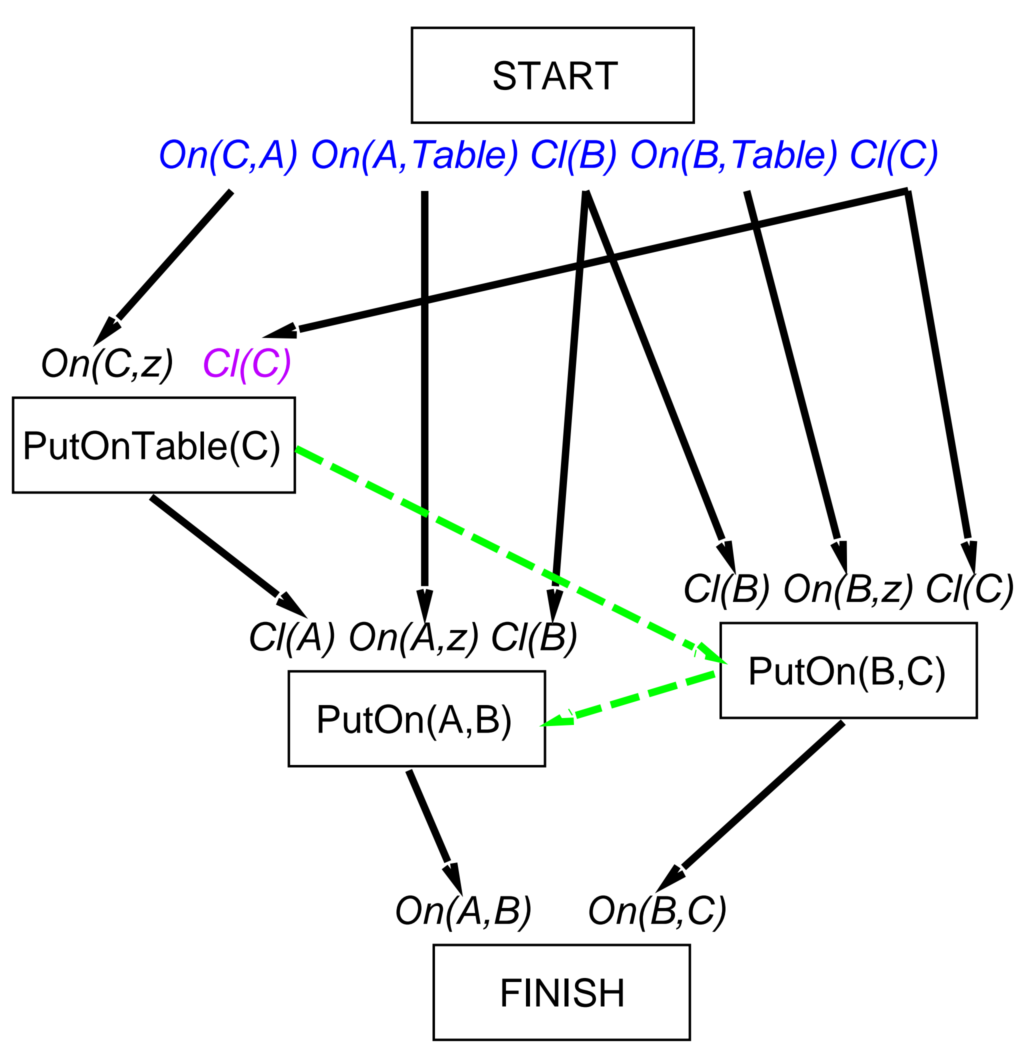
Example: the blocks world (3)

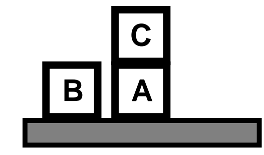


Example: the blocks world (4)



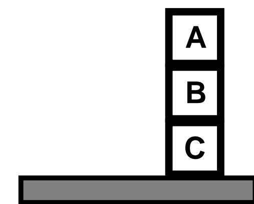
Example: the blocks world (5)





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

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



Planning with partially instantiated operators

In case any of a plan's operators did not have some of its parameters assigned, **possible threats** may appear, dependent on specific value assignments. The POP algorithm can be extended to accommodate such cases.

procedure Choose-Operator(*Plan, Operators,* S_{need}, c)
- $S_{add} \leftarrow$ from *Operators* or Steps(*Plan*) choose a step with an effect
  - c_{add} such, that $u =$Unify(c, c_{add},Bindings(*Plan*))
- **if** no such step exists **then fail**
- to Bindings(*Plan*) add the substitution u
- to Links(*Plan*) add the causal link $S_{add} \xrightarrow{c} S_{need}$
- to Orderings(*Plan*) add the ordering $S_{add} \prec S_{need}$
- **if** S_{add} is a newly added step from *Operators* **then**
  - add S_{add} to Steps(*Plan*)
  - add $Start \prec S_{add} \prec Finish$ to Orderings(*Plan*)

```
procedure Resolve-Threats(Plan)
    for each S_i --c--> S_j ∈ Links(Plan) do
        for each S_threat ∈ Steps(Plan) do
            for each c' ∈ Effects(S_threat) do
                if Subst(Bindings(Plan),c) = Subst(Bindings(Plan),¬c')
                then
                    choose one of
                        Demotion: add S_threat ≺ S_i to Orderings(Plan)
                        Promotion: add S_j ≺ S_threat to Orderings(Plan)
                if not Consistent(Plan) then fail
            end
        end
    end
```

The POP algorithm extended this way is still correct and complete.

Further extensions of POP

The POP algorithm can deal efficiently with some planning tasks, but this is due in part to the limitations of it applicability, including the restricted description language where, for example, the operators specify only deterministic, independent effects, etc. This language can be extended in several ways:

- operators with conditional effects — these require additional conditions in Resolve-Threats
- achieving negated goals (necessary for the conditional effects), which requires using negation, as well as introducing the closed world assumption for the initial state description (to avoid enlisting facts which do not hold)
- operator applicability conditions with alternatives
- universally quantified applicability conditions and effects of the operators

It should be noted that this extended description language, although it now contains more elements of the first-order predicate calculus, is not equivalent to FOPC. It still is a simple language allowing certain types of clauses in certain roles, and nothing more. For example, we still do not allow alternative effects of the operators. The necessary algorithms include formula unification, but not the full theorem proving.

Conditional planning

The conditional planning takes into account both the variable effects of the agent's actions, and the facts unknown at the time of planning.

The CPOP algorithm for creating conditional plans is an extension of POP taking into account: the **context** (conditions that must be met) for the plan steps, conditional steps (which do not cause any specific effects, but acquire knowledge about facts becoming true), duplicate *Finish* states (to take account of contexts, which are not taken into account by the constructed plan), and conditionally solving threats.

The conditional planning algorithm can be further extended to multivalued conditional steps (instead of simply checking a logical condition), and the repetitive loops which are similar to the conditional steps, but instead of conditionally executing a plan step (or a branch), they cause its conditional repetition, and re-testing the condition.

Such plans have a strong character of programs, their implementation resembles program interpretation, and the plan construction is similar to automatic writing programs from specifications.

Short review

1. What are the components of a problem representation for planning in the plan space? Does it have any elements in common (or similar) with the STRIPS representation?

2. What is a partially ordered plan? Give an example of such a plan for the previously considered problem of a student having to pass two exams.

3. Name the operations performed on the partially ordered plans by a system planning in the plan space.

4. What are threats in partially ordered plans?
   What threat could arise (will rather surely rise) while solving the previous student's problem?

5. What are the non-deterministic elements of the POP algorithm, and what role do they play in it?

Planning graphs

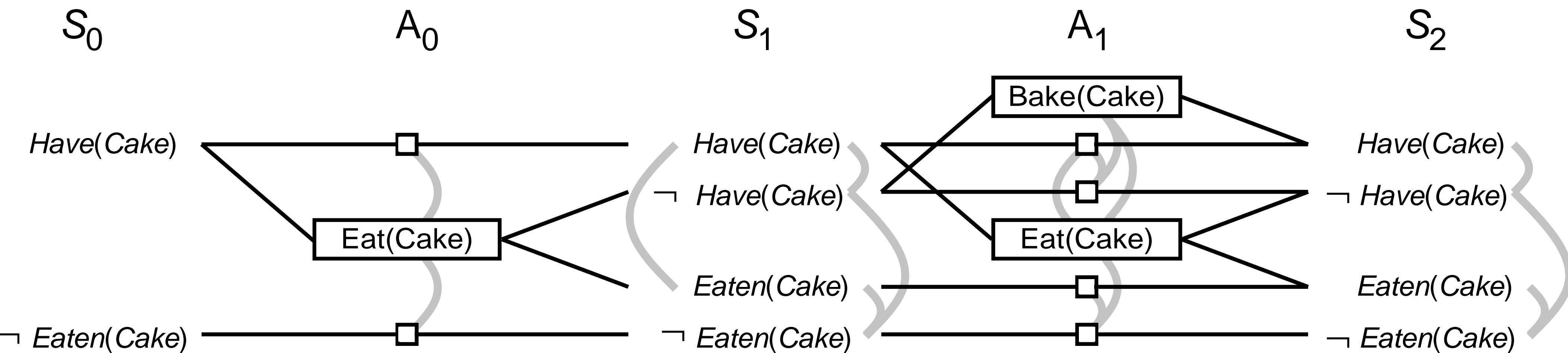
Planning graphs are a different approach to action planning, which can contribute more useful search heuristics. This permits an application of a broader range of searching algorithms. It is also possible to directly derive a complete plan from the planning graph.

A planning graph consists of levels corresponding to steps in time, where the first level describes the start state S_0. The levels are composed alternately of state descriptions and actions. State descriptions are not complete descriptions of some real state, but rather are sets of logical literals, which can be elements of some state description, resulting from some action. The actions are operators linked on one side with literals assuring the satisfaction of their preconditions, and on the other side with the literals describing the effects of the action. In addition to real actions there are empty actions, which correspond to non-change axioms from the situation calculus.

An important constraint of the planning graphs is the lack of variables. When a specific planning domain contains operators with arguments, then they must be rewritten into many fully instantiated cases.

Planning graphs — an example

Init(Have(Cake))
Goal(Have(Cake) $\wedge$ *Eaten(Cake))*
Action(Eat(Cake)
PRECOND: *Have(Cake)*
EFFECT: $\neg$ *Have(Cake)* $\wedge$ *Eaten(Cake))*
Action(Bake(Cake)
PRECOND: $\neg$ *Have(Cake)*
EFFECT: *Have(Cake))*



The grey arcs between actions and literals denote **mutual exclusion (mutex)**.

Mutexes in a planning graph

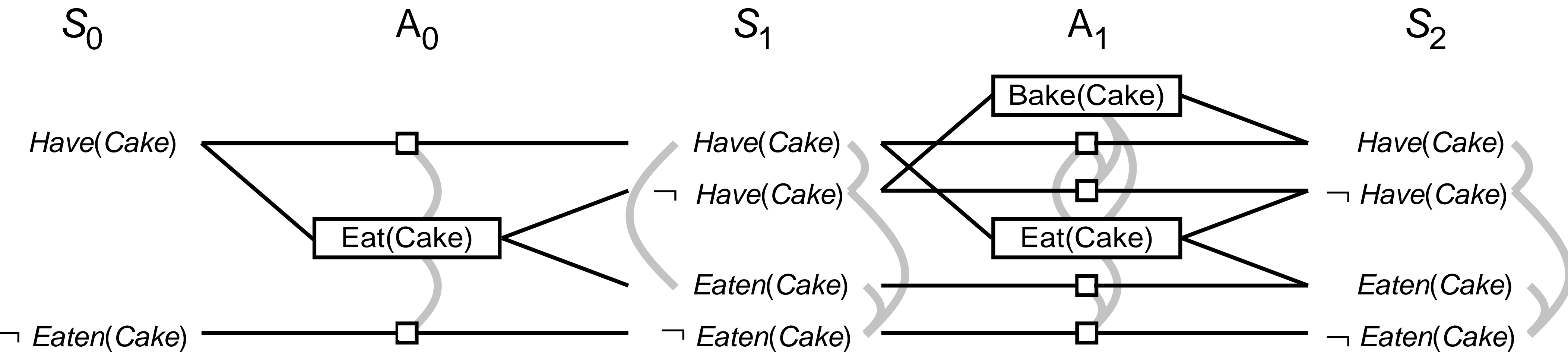
The constructed planning graph does not determine the selection or ordering of actions, but only the potential possibility of taking actions and achieving goals. This way the algorithm is simple and efficient (polynomial in the number of actions and literals), while the planning search space is exponential in the number of literals.

The mutex computing algorithm is given by the following rules:

A mutex exists between two actions at each graph level if any of the following holds:

- one action negates the effects of the other,
- one of the effects of one action negates a precondition of the other,
- one of the preconditions of one action is mutex with one of the preconditions of the other.

Mutex exists between two literals at each level of the planning graph if: (i) one is a negation of the other, or (ii) there is a mutex between each pair of actions achieving both literals.



Note that the literals $Have(Cake)$ i $Eaten(Cake)$ are mutex at level S_1, since the only way to achieve $Have(Cake)$, which is the no-op action, is mutex with the only action achieving $Eaten(Cake)$, which is $Eat(Cake)$.

These two literals are not mutex at level S_2, since new actions to achieve these literals have appeared, and they are not all mutex: $Bake(Cake)$ and preservation of $Eaten(Cake)$.

What are the planning graphs

A planning graph is a structure in which each level A_i contains all actions that are possible in S_i, indicating which actions are not simultaneously possible.

The fact that an action appeared in the planning graph at A_i does not mean that it will be possible to perform this action after i steps. However, the earliest level A_n, in which an action appeared indicates that the execution of this action will not be possible before n steps.

Each level S_i contains all literals that can be due to select the actions from the A_{i-1}, indicating which pairs of literals are not simultaneously possible.

Note that the construction of a planning graph does not require the selection of actions, which would require a combinatorial search. Here we only find the possible actions, and mark some impossible choices by mutexes. A graph with n levels, a actions, and l literals, has size $O(n(a+l)^2)$, and the time required to build it has the same complexity.

Planning graph as a source of search heuristics

It is possible to obtain useful information from the planning graph. For example, a literal absent from the final graph level is impossible to achieve by any plan. In searching the state space, this state could be assigned the heuristic value $h(n) = \infty$. Likewise, in the space of partial plans, we could exclude any partial plan, which contains an unreachable open condition.

This observation could be generalized by defining the cost of achieving the goal literal in the planning graph to be the level number in the planning graph at which this literal first appeared. This number is an approximation of the real cost of reaching each goal, and often this approximation is optimistic. It can be proved that such heuristic is admissible in the sense of the A* algorithm.

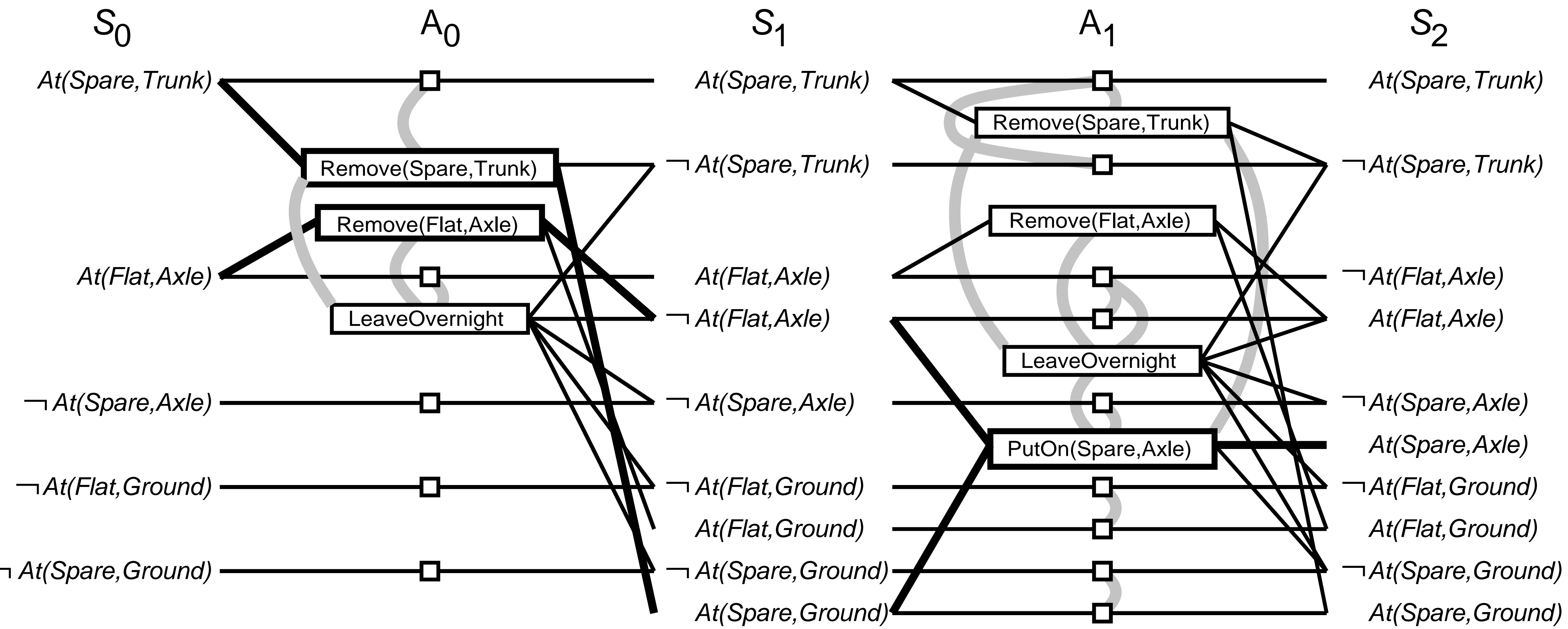
An even better approximation is obtained by defining a **serial planning graph** containing the mutexes between all the pairs of nonempty actions, which reflects the necessity of executing actions step by step, and generates the appropriately longer graph.

To approximate the cost of achieving a conjunction of goals one can compute: (i) the maximum of the costs, (ii) the sum, or (iii) the number of steps to the state in which all goals are present, and are not mutex. This last heuristic works quite well for many problems with significant interactions between goals.

Direct plan generation — the GRAPHPLAN algorithm

```
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)
```

Initial-Planning-Graph — creates a graph with the S_0 state.
Expand-Graph — adds one step to the plan: all possible actions (including empty), their results, and computes mutexes.
Extract-Solution — computes a binary CSP problem, whose variables are actions at each level, their values are: *in* and *out*, and the constraints are to satisfy the preconditions, and to have all the goals among the action effects in the last level. The sequence of *in* operators makes up a complete partial order plan.



S_0
A_0
S_1
A_1
S_2
At(Spare,Trunk)
At(Flat,Axle)
¬ At(Spare,Axle)
¬ At(Flat,Ground)
¬ At(Spare,Ground)
Remove(Spare,Trunk)
Remove(Flat,Axle)
LeaveOvernight
At(Spare,Trunk)
¬ At(Spare,Trunk)
At(Flat,Axle)
¬ At(Flat,Axle)
¬ At(Spare,Axle)
¬ At(Flat,Ground)
At(Flat,Ground)
¬ At(Spare,Ground)
At(Spare,Ground)
Remove(Spare,Trunk)
Remove(Flat,Axle)
LeaveOvernight
PutOn(Spare,Axle)
At(Spare,Trunk)
¬ At(Spare,Trunk)
¬ At(Flat,Axle)
At(Flat,Axle)
¬ At(Spare,Axle)
At(Spare,Axle)
¬ At(Flat,Ground)
At(Flat,Ground)
¬ At(Spare,Ground)
At(Spare,Ground)

Short review

1. What is a planning graph, what is its structure?
2. What is the purpose of mutexes in the planning graphs?
3. Build the planning graph for the previously considered problem of a student having to pass exams for two courses.