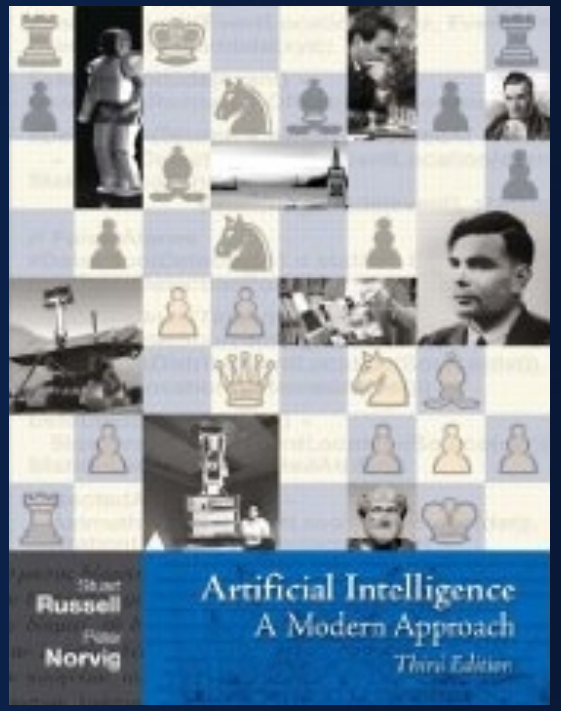


AI Fundamentals: planning

Maria Simi

Planning

LESSON 2: PLANNING GRAPHS, GRAPHPLAN, PARTIAL ORDER PLANNING

Planning graphs

A data structure called a **planning graph**, can be a rich source of information:

1. It can be used to give better heuristic estimates to employ in conjunction with search algorithms
2. It is the search space of an algorithm called **GraphPlan**

A search tree is exponential in size. A planning graph is polynomial size approximation of the search tree.

It can be constructed quickly.

The planning graph can't answer definitively whether G is reachable from S_0, but

- it may discover that the goal is not reachable
- it can estimate how many steps it takes, in the most optimistic case, to reach G, so it can be used to derive an *admissible heuristic*.

Planning graph definition

Planning graphs work only for **propositional planning** problems, with no variables.

A planning graph is a directed graph which is built *forward* and is organized into **levels**:

- a level S_0 for the initial state, representing each fluent that holds in S_0
- a level A_0 consisting of nodes for each ground action applicable in S_0
- alternating levels S_i followed by A_i are built until we reach a termination condition.

S_i contains all the **literals** that ***could*** hold at time i (even contrary literals P and $\neg P$)
A_i contains all the **actions** that ***could*** have their preconditions satisfied at time i

Mutual exclusion links (**mutex**) connect incompatible pairs of literals and actions

- Mutex between literals mean that two literals cannot appear in the same belief state.
- Mutex between actions mean that two actions cannot occur at the same time.

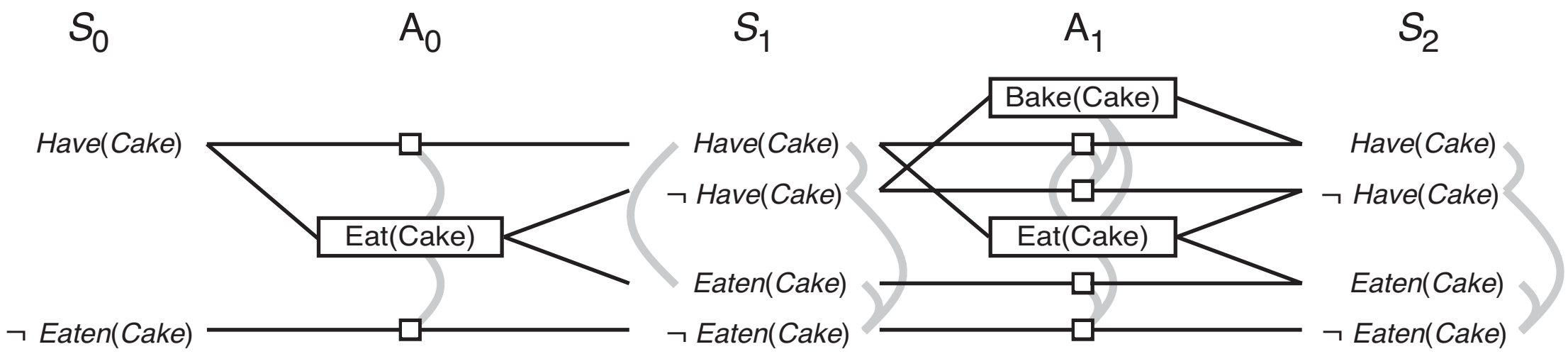
More in detail after the example …

An example of planning graph

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

You would like to eat your cake and still have a cake.
Likely, you can bake a new one, if you don't have one.

Rectangles indicate actions
Small squares persistence actions (**no-ops**)
Straight lines indicate preconditions and effects
Mutex links are shown as curved gray lines



Mutex computation

- Mutex relations between **actions:**
  1. *Inconsistent effects*: one action negates an effect of another action, e.g. persistence of $Have(Cake)$ and $Eat(Cake)$ have inconsistent effects.
  2. *Interference*: one of the effects of one action is the negation of a precondition of the other: $Eat(Cake)$ interferes with the persistence of $Have(Cake)$.
  3. *Competing needs*: one of the preconditions of one action is mutually exclusive with a precondition of the other, e. g. $Bake(Cake)$ and $Eat\,(Cake)$.
- Mutex relations between **literals** at the same level
  1. if one is the negation of the other
  2. *inconsistent support*: if each possible pair of actions that could achieve the two literals is mutually exclusive. Example $Have(Cake)$, produced by $noOp$, is mutex with $Eaten(Cake)$, produced by $Eat(Cake)$

Properties of the planning graph

- Each level S_i represents a set of possible belief states. Two literals connected by a *mutex* belong to different belief states.
- The levels, alternating S's an A's, are computed until we reach a point where two consecutive levels are identical. The graph for the cake **levels off** at S_2.
- The level j at which a literal first appears is never greater than the level at which it can be achieved. We call this the **level cost** of a literal/goal.
- The process of constructing the planning graph does not require choosing among actions and is very fast.
- A planning graph is polynomial in the size of the planning problem:
  an entire graph with n levels, a actions, l literals, has size $O(n\,(a+l)^2)$.
  Time complexity is the same.

Use of planning graphs for heuristic estimation

Information that can be extracted from the planning graph:

1. If any goal literal fails to appear in the final level of the graph, then the **problem is unsolvable**;
2. We can estimate the cost of achieving a goal literal g_i as by its **level cost**. This estimate is *admissible*.
3. A better estimate can be obtained by **serial planning graphs**: by enforcing only one action at each level (adding mutex).

Estimating the heuristic cost of a conjunction of goals:

1. **max-level** heuristic: the maximum **level cost** of any of the sub-goals. Admissible.
2. **level sum** heuristic: the sum of the level costs of the goals. This can be inadmissible when goals are not independent, but it may work well in practice.
3. **set-level** heuristic: finds the level at which **all the literals** in the goal appear together in the planning graph, **without any mutex** between pairs of them. Admissible, accurate, but not perfect.

The planning graph as relaxed problem

The planning graph can be seen as a relaxed problem with the following characteristics. When g appears at level S_i we can prove that:

- If there exists a plan with i action levels that achieves g then g will appear at level i; if g does not appear there is no plan.
- but not vice versa: the fact that a g appears does not mean that there is a plan.

If g does appear at level i, the plan **possibly exists** but, in order to be sure, we need to check the *mutex* relations, pairs of conflicting actions and pairs of conflicting literals.

Even so, there are plans that are not recognized as impossible, for example the goal to get block A on B, B on C, and C on A. Mutex are not enough to detect that this problem is impossible: higher order constraints would be necessary.

We need to search *over the planning graph.*

The GRAPHPLAN algorithm

The GRAPHPLAN algorithm is a strategy for **extracting a plan** from the planning graph.

The planning graph is computed **incrementally** by the function EXPAND-GRAPH.

Once a level is reached where all the literals in the goal **show up as non-mutex**, an attempt to extract a plan is made with EXTRACT-SOLUTION.

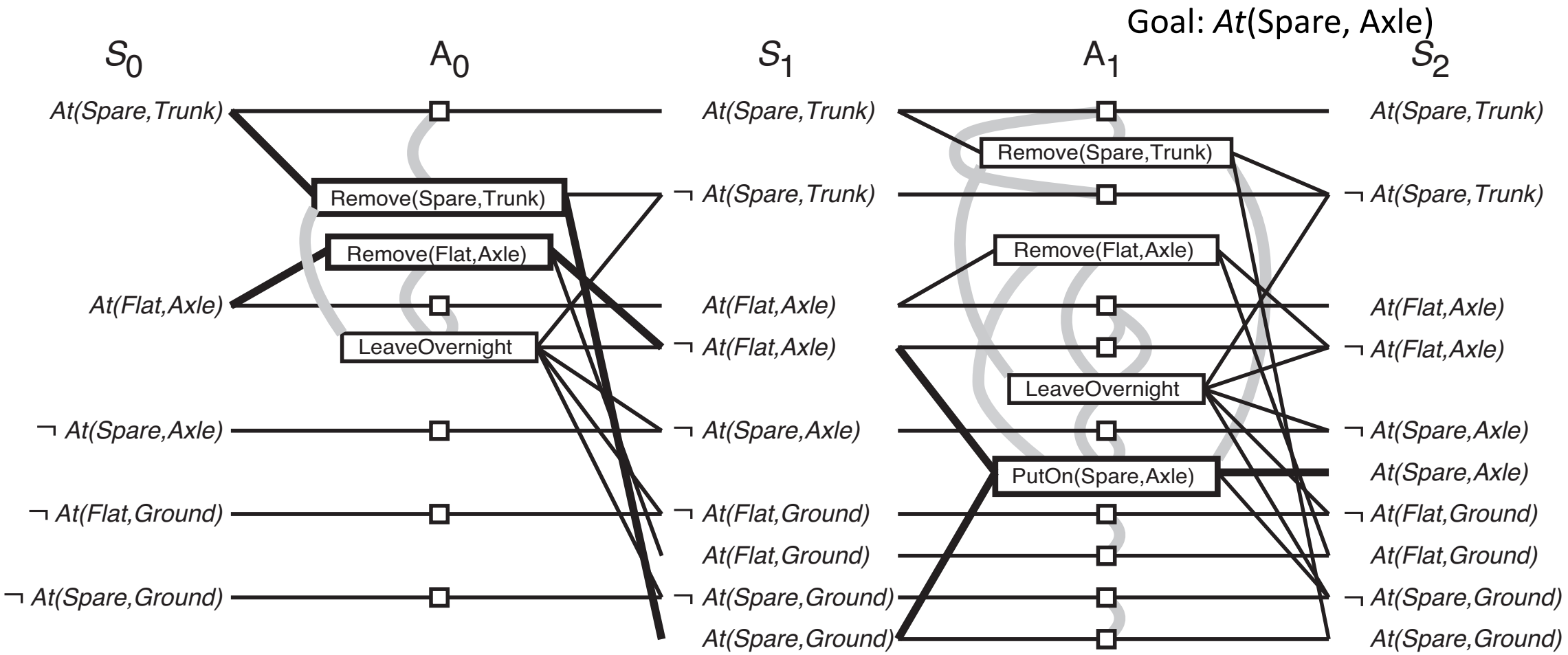
If EXTRACT-SOLUTION fails, the failure is recorded as a **no-good**, another level is expanded and the process repeats until a terminanation condition is met.

GRAPHPLAN pseudo-code

```
function GRAPHPLAN(problem) returns solution or failure

    graph ← INITIAL-PLANNING-GRAPH(problem)
    goals ← CONJUNCTS(problem.GOAL)
    nogoods ← an empty hash table
    for tl = 0 to ∞ do
        if goals all non-mutex in S_t of graph then
            solution ← EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
            if solution ≠ failure then return solution
        if graph and nogoods have both leveled off then return failure
        graph ← EXPAND-GRAPH(graph, problem)
```

GRAPHPLAN on the *spare-tire* example



Progress of the algorithm

1. The planning graph is initialized with S_0, representing the initial state. The goal $At(Spare, Axle)$ is not present in S_0.
2. EXPAND-GRAPH adds into A_0 the three applicable actions and persistence actions for all the literals in S_0. The effects of the actions are added at level S_1. Mutex relations are also added to the graph.
3. The goal $At(Spare, Axle)$ is not yet present in S_1, so we call EXPAND-GRAPH again adding A_1 and S_2
4. All the literals from the goal are present in S_2, and none of them is mutex with any other, … so we can call EXTRACT-SOLUTION.

EXTRACT-SOLUTION

Two approaches:

1. **Solve as a boolean CSP**: the variables are the actions at each level, the values for each variable are *in* or *out* of the plan, and the constraints are the mutexes and the need to satisfy each goal and precondition.
2. **Solve as a backward search problem**:
   - Start with S_n, the last level of the planning graphs, and the goals.
   - For each level S_i select a number of non-conflicting actions in A_{i-1} whose effects cover the goals in S_i. The resulting state is S_{i-1} with goals the preconditions of the selected actions.
   - The process is repeated until level S_0 hoping all the goals are satisfied.

If EXTRACT-SOLUTION fails to find a solution for a set of goals at a given level, we record the (*level*, *goals*) pair as **no-good**, so that we can avoid to repeat the computation.

Going back to the example …

Complexity and heuristics

Constructing the planning graph takes polynomial time

Solution extraction is **intractable** in the worst case.

Heuristics exist.

Greedy algorithm based on the level cost of the literals:

1. Pick first the literal with the highest level cost.
2. To achieve that literal, prefer actions with easier preconditions. That is, choose an action such that the sum (or maximum) of the level costs of its preconditions is smallest.

Termination of GRAPHPLAN

We can prove that GRAPHPLAN will in fact **terminate** and return **failure** when there is no solution.

But we may need to expand the graph even after it levels off.

Theorem: If the graph and the no-goods **have both leveled off**, and no solution is found we can safely terminate with failure.

Sketch of the proof:

1. Literals and actions increase monotonically and are finite, we need to reach a level where they stabilize.
2. Mutex and no-goods decrease monotonically and cannot become less than zero, so they too must level off.
3. When we reach this stable state if one of the goals is missing or is mutex with another goal it will remain so. We may as well stop computation.

Other approaches to planning

The most popular and effective approaches nowadays:

- Planning as a Boolean satisfiability problem (SATPLAN)
- Forward state-space search with carefully crafted heuristics
- Search using a planning graph (GRAPHPLAN)

How do you translate a problem in PDDL to a Boolean SAT?

Other approaches that have been attempted:

- Planning as a Constraint Satisfaction problem.
- Planning as refinement of **partially ordered plans**

International Planning Competition

Year	Track	Winning Systems (approaches)
2008	Optimal	GAMER (model checking, bidirectional search)
2008	Satisficing	LAMA (fast downward search with FF heuristic)
2006	Optimal	SATPLAN, MAXPLAN (Boolean satisfiability)
2006	Satisficing	SGPLAN (forward search; partitions into independent subproblems)
2004	Optimal	SATPLAN (Boolean satisfiability)
2004	Satisficing	FAST DIAGONALLY DOWNWARD (forward search with causal graph)
2002	Automated	LPG (local search, planning graphs converted to CSPs)
2002	Hand-coded	TLPLAN (temporal action logic with control rules for forward search)
2000	Automated	FF (forward search)
2000	Hand-coded	TALPLANNER (temporal action logic with control rules for forward search)
1998	Automated	IPP (planning graphs); HSP (forward search)

Translating to a Boolean SAT problem

How to translate a PDDL description into a form that can be processed by SATPLAN?

1. Propositionalize the actions: create ground instances of the actions. Used in step 4.
2. Define the initial state: F^0 for every fluent F in the initial state, $\neg F^0$ for every fluent F *not* in the initial state.
3. Proposizionalize the goal, by instantiating variables to a disjunction of constants.
   $On(A, x) \wedge Block(x)$ becomes $On(A, B) \vee On(A, C) \vee On(A, D)$ … for each block mentioned
4. Add successor-state axioms. For each fluent:
   $F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t)$
   where $ActionCausesF^t$ is a disjunction of all the **ground actions** that have F in their add list and $ActionCausesNotF^t$ is a disjunction of all the ground actions that have F in their delete list.
5. Add precondition axioms. For every ground action: $A^t \Rightarrow \mathrm{PRE}(A)^t$
6. Add action exclusion axioms: every action is distinct from every other action.

The resulting translation is in the form that can be given in input to SATPLAN to find a solution.

Planning as Constraints Satisfaction

SKIPPED

It is possible to encode a **bounded** planning problem (i.e. the problem of finding a plan of length k) as a constraint satisfaction problem (CSP). k is the planning horizon.

The encoding is similar to the encoding to a SAT problem:

1. State variables: a variable for each fluent and time
2. Action variables: a variable $Action_t$ for each action and time t, leading from state t to t+1 (the domains are the possible actions). *Note*: not a variable for each action.
3. Precondition, effect, and frame constraints for each $Action_t$

   $\neg(Var_t \neq v \wedge Action_t = A)$ for each precondition $Var = v$

   $\neg(Var_{t+1} \neq v \wedge Action_t = A)$ for each postcondition $Var = v$

   $Action_t \notin A_S \Rightarrow Var_{t+1} = Var_t$ A_S is the set of actions that include Var in the effect
4. Initial-state and goal constraints. Possibly, other state constraints.

Note: we don't need exclusion axioms.

Partial Order Planning (POP)

Partial Order Planning is an interesting approach, very popular in the nineties.

Interesting, since it addresses the issue of independent subgoals, that can be performed in parallel. For some specific tasks, such as operations scheduling is the technology of choice.

Interesting, because it represents a **change of paradigm**: planning as **search in the state of partial plans** rather than in space of states.

The plan refinement approach is also more **explainable**: it makes it easier for **humans to understand** what the planning algorithms are doing and verify that they are correct.

Partial Order Planning: ideas

The driving principle is **least-commitment**.

Partially ordered plans:

- Do not order steps in the plan unless necessary to do so.
- In a partial-order plan steps are partially ordered.
- *Plan linearization*: to impose a total order to a partially ordered plan.

Partially instantiated plans:

- Leave variables uninstantiated until is necessary to instantiate them
- A plan without variables is said to be **totally instantiated**.

Searching in the space of partial plans

Instead of searching in space of states as in the classical formulation, we **search in the space of partial plans**.

1. We start with an empty plan.
2. At each step we use operators for plan construction and refinement:
   - We can add actions in order to satisfy some pre-condition, i.e. fixing **flaws** in the plan.
   - We can instantiate variables
   - We can add ordering constraints between steps.
3. We stop when we obtain a **consistent** and **complete** plan where:
   - All the preconditions of all the steps are satisfied
   - Ordering constraints do not create cycles

Every linearization is a solution.

Representation for plans

Partial plan are represented as:

- A set of actions, among them *Start* and *Finish*.
- A set of open preconditions.
- Constraints among actions of two different types:
  - Ordering relations: $S_1 < S_2$ (S_1 before S_2)
  - Causal links $S_1 \longrightarrow_{cond} S_2$ (S_1 achieves *cond* for S_2)

  Note: If $S_1 \longrightarrow_{cond} S_2$ then $S_1 < S_2$ but not vice versa

Example:

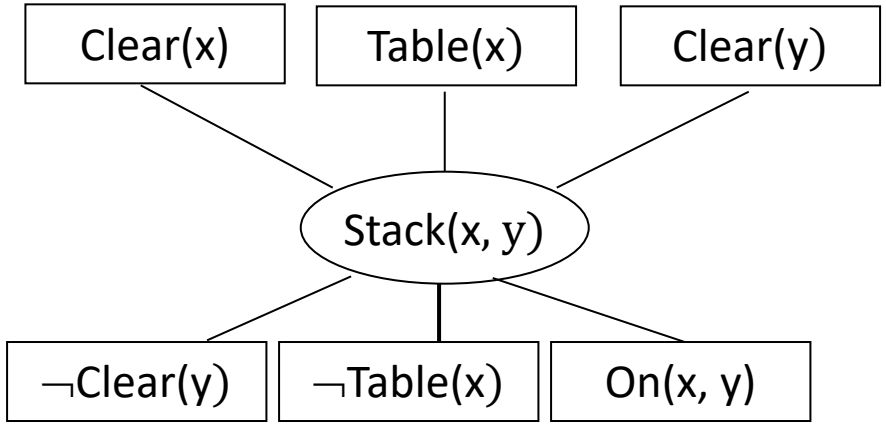
$\{Unstack(A, B), Unstack(C, D), Stack(B, A), Stack(D, C), Start, Finish\}$

$Unstack(A, B) < Stack(B, A)$ $\quad$ $Unstack(A, B) \rightarrow_{Clear(B)} Stack(B, A)$

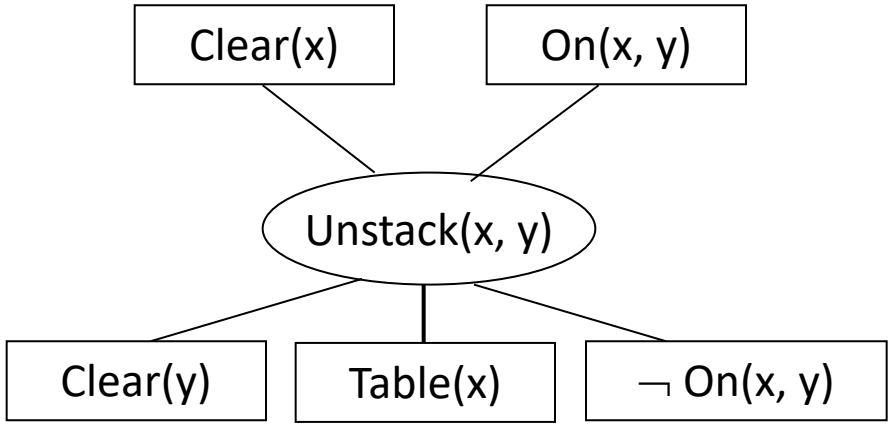
$Unstack(C, D) < Stack(D, C)$ $\quad$ $Unstack(C, D) \rightarrow_{Clear(D)} Stack(D, C)$

Representation for actions

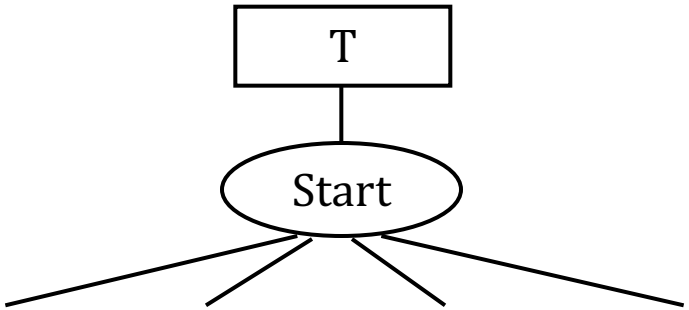
Action **Stack**



Action **Unstack**

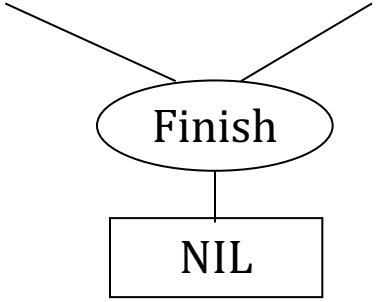


Empty plan



Facts holding in the initial state

Facts that must hold in the goal state



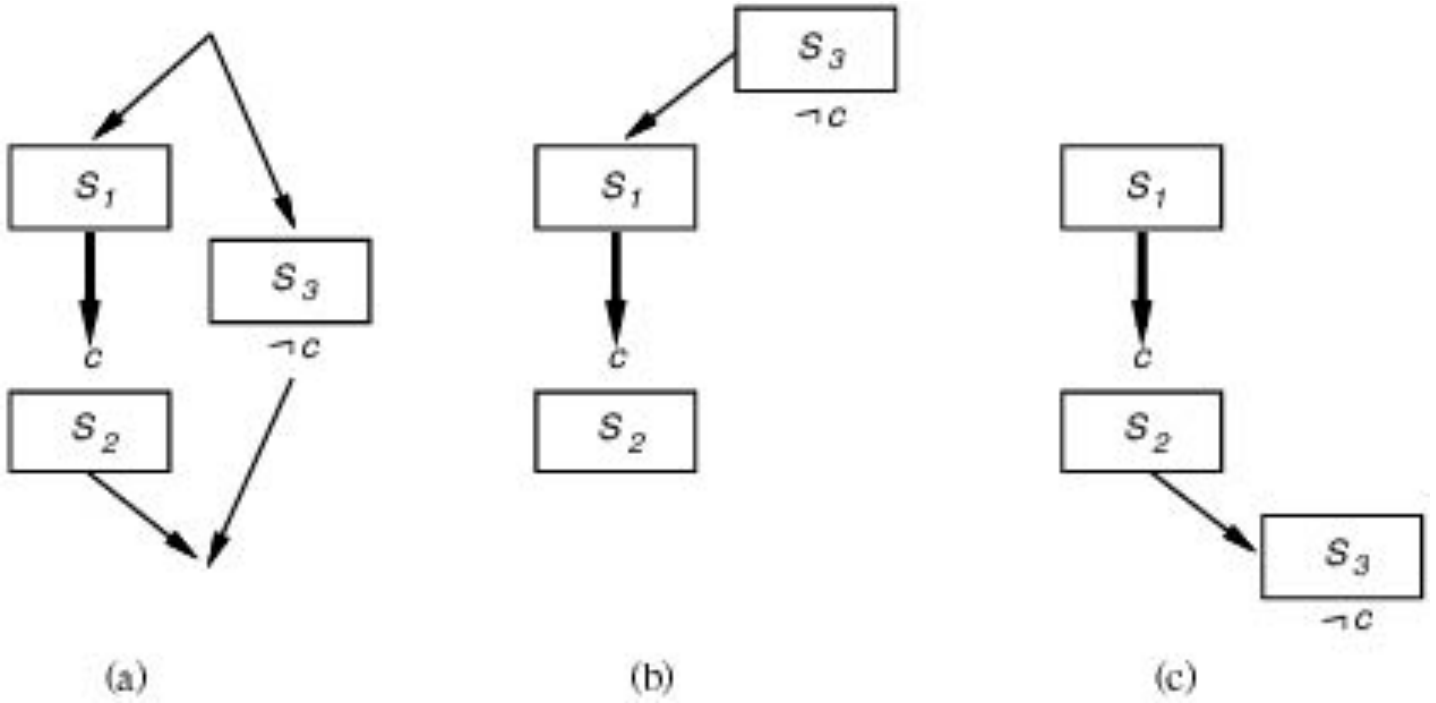
PoP algorithm

We start with the empty plan, with *Start* and *Finish*.

At each step:

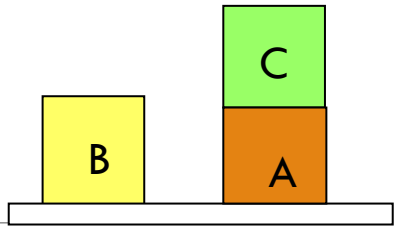
- We choose a step B and one of its **open preconditions** p and we generate a successor plan for each action A (old or new) having p among the effects
- After choosing an action A consistency is re-established as follows:
  - Add to the plan the constraints $A < B$ and $A \rightarrow_p B$
  - Possible actions *C* having $\neg p$ as effect, are potential conflicts (or ***threats***). They need to be anticipated or delayed, adding the constraints $C < A$ or $B < C$. This step may fail.
- We stop when the set of open pre-conditions is empty.

Threats removal

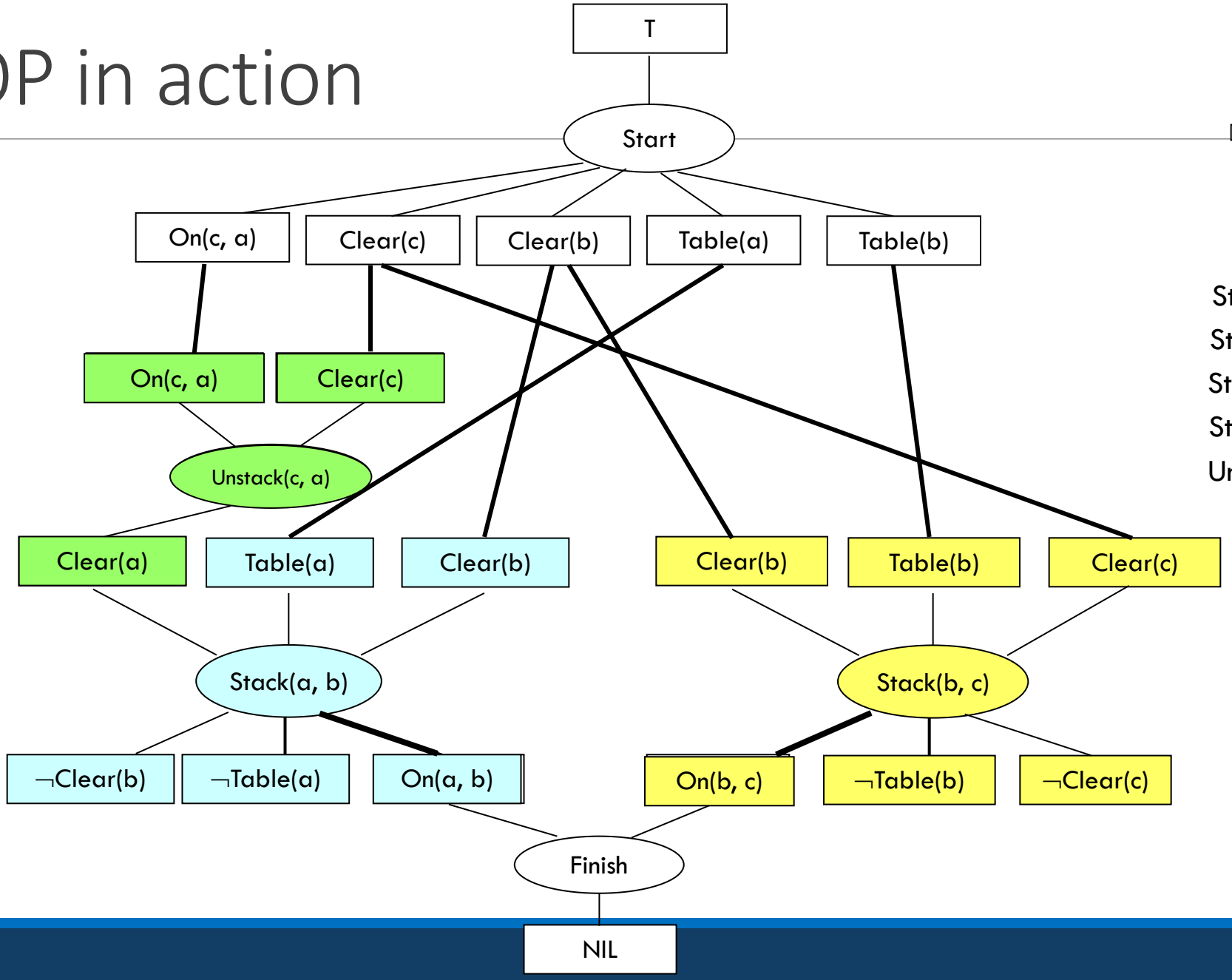


(a) S_3 is a threat for pre-condition c di S_2, achieved by S_1
(b) The *threat* is resolved by *demotion*
(c) The *threat* is resolved by *promotion*

POP in action



Initial state



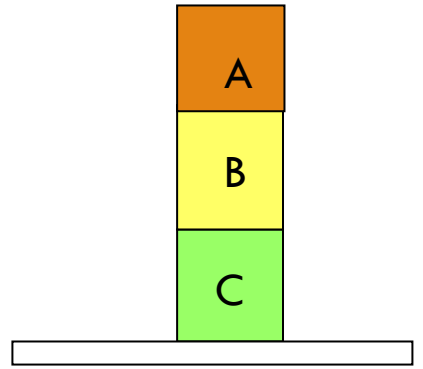
Start < Finish

Start < Stack(a, b) < Finish

Start < Stack(b, c) < Finish

Stack(b, c) < Stack(a, b)

Unstack(c, a) < Stack(b, c)



goal

PoP: analysis

- We obtained a complete and consistent partial plan
- Any linearization is a solution, in this case only one:

  $[Unstack(\mathrm{C}, \mathrm{A}), Stack(\mathrm{B}, \mathrm{C}), Stack(\mathrm{A}, \mathrm{B})]$
- The PoP algorithm is correct and complete:
  - any plan computed is a solution
  - If a plan exists, the algorithm finds it
- The Sussman's anomaly was solved without problems.

Planning in the real world

1. Actions with duration and resource constraints
2. Hierarchical planning
3. Planning and acting in nondeterministic domains
4. Planning in a multi-agent environments

Conclusions

- ✓ Planning combines **search** and **logic**.
- ✓ Progress in **forward planning** (GRAPHPLAN and PLANSAT**)** has been steady in the last ten years and there is an increasing use of planners in industrial applications.
- ✓ Planners rely on a combination of heuristics and there is no clear winning approach for every domain.
- ✓ An important speed-up can be obtained by recognizing that a problem has **serializable sub-goals,** i.e. sub-goals can be ordered in such a way that they can be achieved in that order without having to undo any of the previously achieved sub-goals.
- ✓ Issues going beyond “classical planning”, to more realistic scenarios, make the field challenging.

References

- ✓ Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach* (3rd edition). Pearson Education 2010 [Chapter 10]