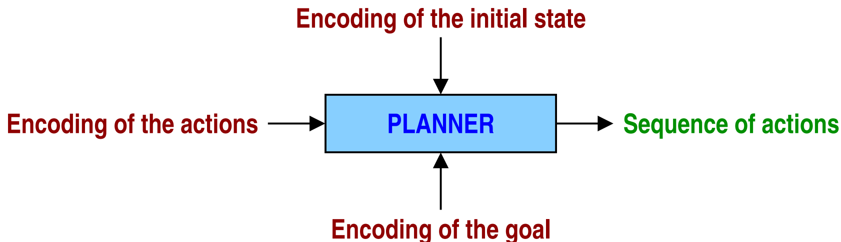
CS60045 Artificial Intelligence
Autumn 2023

Automated Planning

Classical planning

- We have an initial state, a goal state, and a set of allowed actions in each state.
- The task is to find a sequence of actions to reach the goal.
- So far, we had to write individual codes for solving individual search problems.
- An **automated planner** is a *single* planner program.
- We need to encode our problem in the language of the planner. We feed the encoding to the planner, and let the planner select a sequence of actions for solving our problem.
- A good planner will give optimal or near-optimal solutions.



Planning Domain Definition Languages (PDDL)

- Also called **Action Description Languages** (**ADL**).
- STRIPS (Stanford Research Institute Problem Solver) is one of the earliest automated planner developed by Richard Fikes and Nils Nilsson in 1971.
- STRIPS also refers to the problem-specification language.
- STRIPS is human-readable and easy to understand.
- Many modern PDDLs are inspired by STRIPS.
- We will not follow the exact syntax of STRIPS.
- STRIPS uses a subset of predicate-logic expressions.

STRIPS encoding

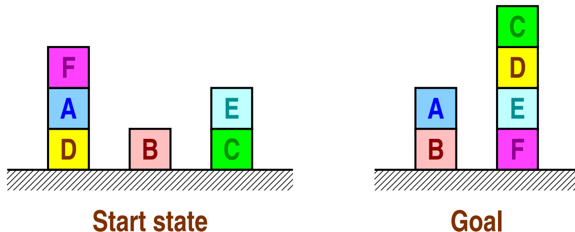
Encoding of states

- A **fluent** is a single predicate with constant arguments (if any).
- A state is specified by a set of fluents.
- All of the fluents in a state must be true.

Encoding of actions (Action schemas)

- An action may use variables.
- Quantifiers are not allowed.
- **Precondition**: A set of predicates that must be true for taking the action.
- **Delete list**: A set of predicates that are no longer true after the action.
- **Add list**: A set of predicates that are true after the action.

Encoding example: Blocks world

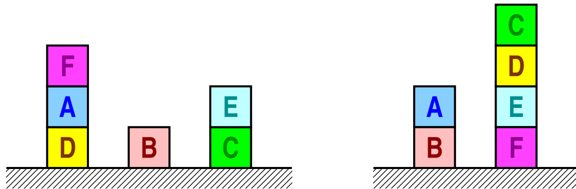


- The objects are the blocks A, B, C, D, E, and F, and the floor.
- Predicates
  - **on(x,y):** x is immediately on y.
  - **clear(x):** The top of x is clear.
- Action
  - **move(x,s,d):** Move a block x from s to d.
  - For this to apply, x must be sitting on s, and the tops of x and d must be clear.

Encoding example: Blocks world

Start state

on(A,D) ∧ on(B,floor) ∧ on(C,floor) ∧ on(D,floor) ∧ on(E,C) ∧ on(F,A) ∧ clear(B) ∧ clear(E) ∧ clear(F) ∧ clear(floor)



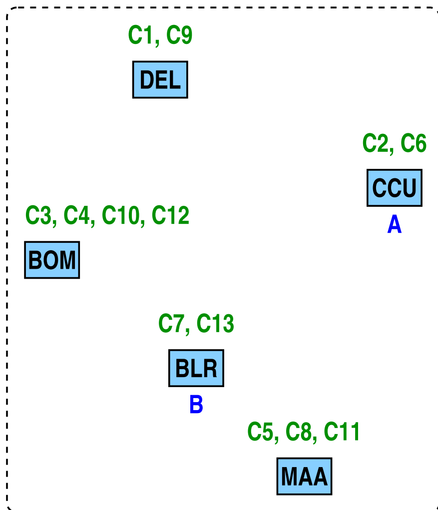
Goal state

on(A,B) ∧ on(B,floor) ∧ on(C,D) ∧ on(D,E) ∧ on(E,F) ∧ on(F,floor) ∧ clear(A) ∧ clear(C) ∧ clear(floor)

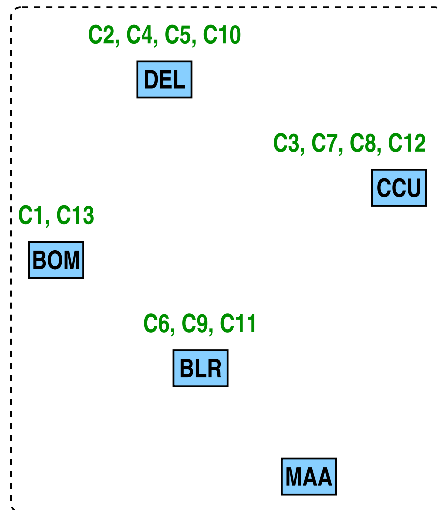
Action schema for move(x,s,d)

- **Precondition:** on(x,s) ∧ clear(x) ∧ clear(d)
- **Delete list:** on(x,s) ∧ clear(d)
- **Add list:** on(x,d) ∧ clear(s) ∧ clear(floor)

Encoding example: Cargo transport



Initial state



Goal

Encoding example: Cargo transport

Objects

- The airports BLR, BOM, CCU, DEL, and MAA
- The cargo planes A and B
- The cargo items C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, and C13

Predicates

- **plane_at(p,a):** The plane p is at airport a.
- **cargo_at(c,a):** The cargo item c is at airport a.
- **cargo_in(c,p):** The cargo item c is in plane p.

Action schemas

- **load(c,p,a):** Load cargo item c to the plane p at airport a.
- **fly(p,s,d):** Fly the airplane p from the source airport s to the destination airport d.
- **unload(c,p,a):** Unload the cargo item c from the plane p at airport a.

Encoding example: Cargo transport

Initial state

cargo_at(C1,DEL) $\wedge$ cargo_at(C2,CCU) $\wedge$ cargo_at(C3,BOM) $\wedge$ cargo_at(C4,BOM) $\wedge$ cargo_at(C5,MAA) $\wedge$ cargo_at(C6,CCU) $\wedge$ cargo_at(C7,BLR) $\wedge$ cargo_at(C8,MAA) $\wedge$ cargo_at(C9,DEL) $\wedge$ cargo_at(C10,BOM) $\wedge$ cargo_at(C11,MAA) $\wedge$ cargo_at(C12,BOM) $\wedge$ cargo_at(C13,BLR) $\wedge$ plane_at(A,CCU) $\wedge$ plane_at(B,BLR)

Goal state

cargo_at(C1,BOM) $\wedge$ cargo_at(C2,DEL) $\wedge$ cargo_at(C3,CCU) $\wedge$ cargo_at(C4,DEL) $\wedge$ cargo_at(C5,DEL) $\wedge$ cargo_at(C6,BLR) $\wedge$ cargo_at(C7,CCU) $\wedge$ cargo_at(C8,CCU) $\wedge$ cargo_at(C9,BLR) $\wedge$ cargo_at(C10,DEL) $\wedge$ cargo_at(C11,BLR) $\wedge$ cargo_at(C12,CCU) $\wedge$ cargo_at(C13,BOM)

It does not matter where the freight planes end up.

Action schemas

	Preconditions	**Delete list**	**Add list**
fly(p,s,d)	plane_at(p,s)	plane_at(p,s)	plane_at(p,d)
load(c,p,a)	cargo_at(c,a) $\wedge$ plane_at(p,a)	cargo_at(c,a)	cargo_in(c,p)
unload(c,p,a)	cargo_in(c,p) $\wedge$ plane_at(p,a)	cargo_in(c,p)	cargo_at(c,a)

Algorithms for solving the classical planning problem

- Forward or progressive search
- Backward or regressive search
- Specialized planning algorithms
  - **Linear planning** [Early 1970s]
  - **WARPlan** [Warren, 1974]
  - **Partial-order planning** [Sacerdoti, 1975; Tate, 1975; Sussman, 1975]
  - **GraphPlan** [Blum and Furst, 1997]
  - **SATPlan** [Kautz and Selman, 1998]

Forward search

We can use the state-space search methods introduced earlier.

BFS, DFS, and A* (under a suitable heuristic) apply.

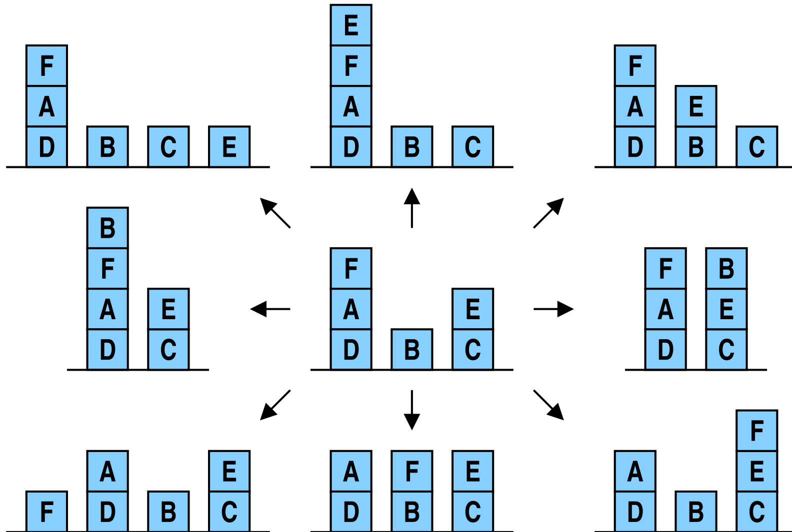
These methods are very inefficient.

The branching factor may be too high.

Example: Consider the blocks world with 100 blocks.

- The maximum number of actions is 200.
- Let t be the number of towers at some point of time. Only the tower tops can be moved. Each tower top can be moved to the top of another tower or to the floor (unless it is already on the floor). There is a maximum of t^2 possibilities.
- t can be as large as 100.
- Taking 10 as an average value of t, the branching factor would be about 100.
- The total size of the state-space graph can be $100^{200} = 10^{400} \approx 2^{1329}$.

Forward search: Example of the first step



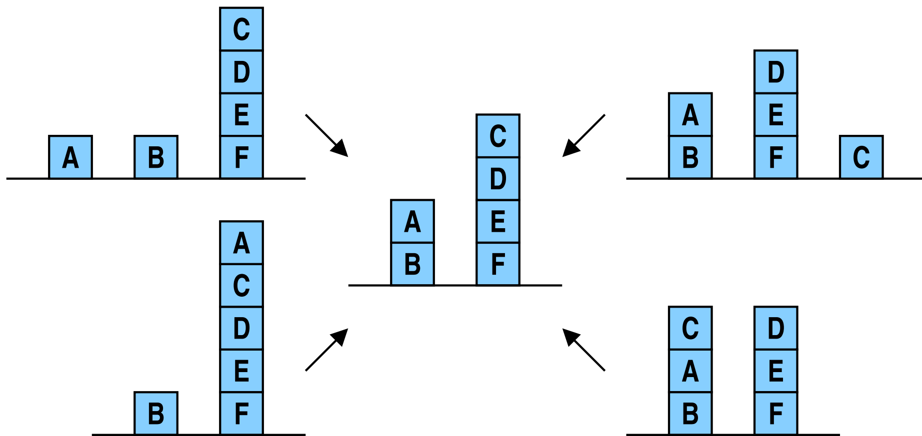
Backward search

- Start from the goal state, and move backward by undoing actions until the initial state is reached.
- Apparently, backward search generates the same state-space graph as forward search.
- Backward search can be goal-directed.
- Forward search explores all available actions. Good heuristics help reduce the explored space.
- Backward search from a state needs to explore only those actions that can result in that state.
- This may reduce the branching factor considerably.

Backward search: Example of the first step

In the last step:

A must have come from the floor or from the top of C.
C must have come from the floor or from the top of A.



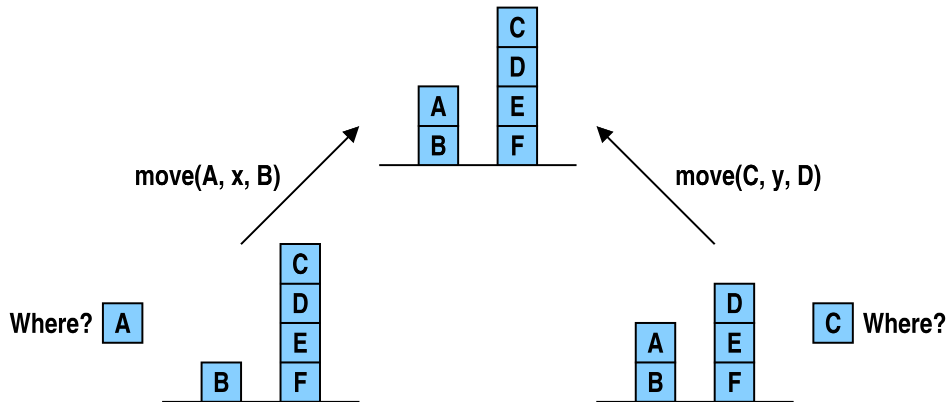
Backward search: Least commitment planning

- Uses variables.
- Do not commit where A or C has com from.
- Leave those places as variables.
- Later in the search, instantiate the variables.
- This reduces branching factor additionally.

- In practical implementations, backward search is found to be more efficient than forward search.
- Using variables in states makes programming backward search rather complicated.
- The idea of least commitment and delayed instantiation yields better planning algorithms.

- We are still in the need of efficient planning algorithms.

Backward search: Example of least commitment planning

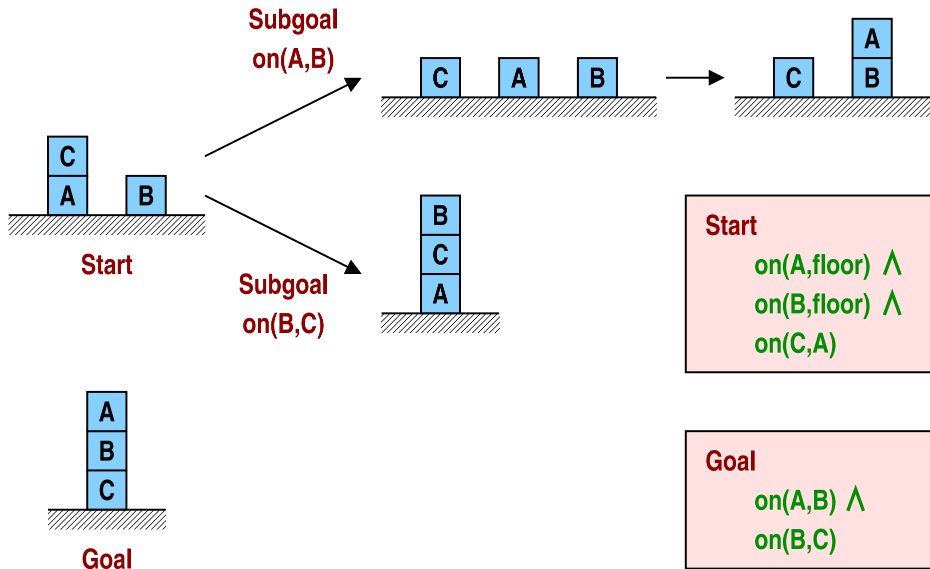


A divide-and-conquer approach

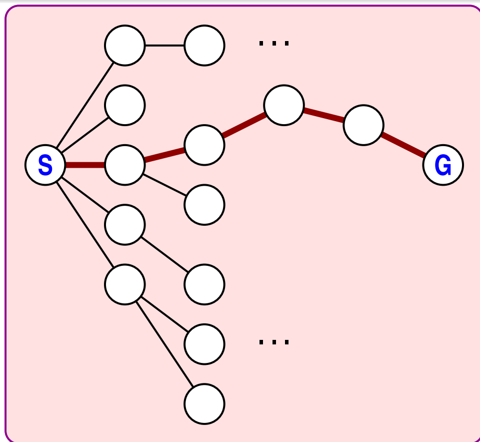
- The goal is a conjunction of literals.
- Achieve the goal literals one by one.

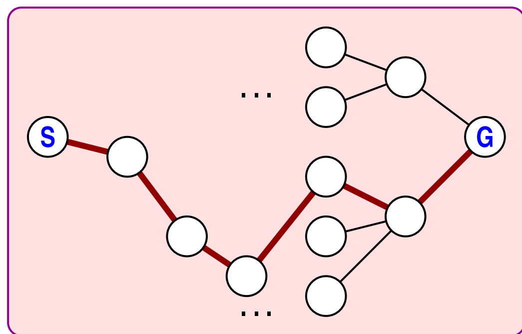
- Solving the goal literals independently of one another is not a good idea.
- **Sussman anomaly** is introduced by Allen Brown (1973), not by Gerald Sussman (1975).
- We need to interleave actions for achieving different goals.

Sussman anomaly



From state space to plan space





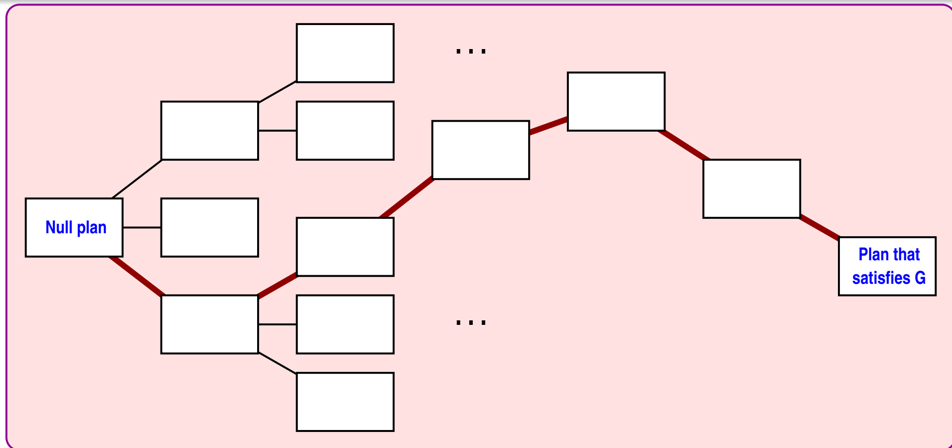
Progressive state–space planner

Regressive state–space planner

Each circle is a state and may contain variables (for regressive planners).

The **path** from S to G gives the plan.

Planning in the plan space



Plan–space planner

Each rectangle is a **partial plan**. A link stands for **plan refinement**.

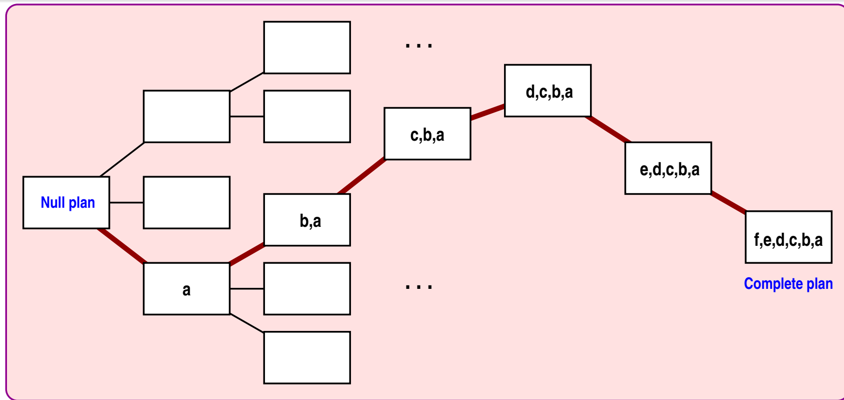
We will study goal-directed (that is, regressive) plane-space planners.

Plan operators

- **Add an action:** This is driven by a need to achieve some subgoal(s).
- **Link the actions:** Add the cause-effect relations between pairs of actions taken.
- **Order and reorder the actions:** Specify/Modify the precedence (order) of taking the actions.
- **Instantiate variables in the actions:** Unify predicates by variable substitution.

We eventually need a complete chain of actions to be followed sequentially from the beginning (start) to the end (goal).

Total-order planner



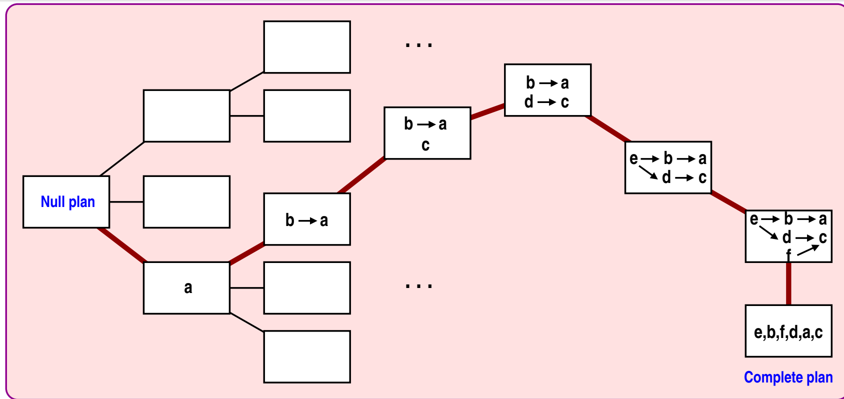
Regressive total–order planner

A total-order planner adds the next action always at the beginning.

Actions are taken in the reverse order as they are included in the plan.

Regressive total-order planning is essentially the same as regressive state-space planning.

Partial-order planner (POP)



Regressive partial–order planner (POP)

A partial-order planner adds the next action anywhere.

A precedence relation is maintained to indicate which action should be taken before which action.

Eventually, the partial order is converted to a total order.

POP: The data structures

A **partial plan** $\mathcal{P}$ is a 3-tuple $(\mathcal{A}, \mathcal{O}, \mathcal{L})$.

- $\mathcal{A}$ consists of all actions taken so far in the partial plan.
- $\mathcal{O}$ stores the partial order among the actions, discovered so far.
- $\mathcal{L}$ is a set of **causal links**. If the precondition of an action is the same as the effect of another action, then a causal link is established between the precondition and the effect.

We also maintain a set SG of **subgoals**.

- Each subgoal is a precondition Q of some action $a \in \mathcal{A}$.
- It is represented by the pair (Q, a).
- There is no causal link between the precondition Q of a and the effect of any other action in $\mathcal{A}$.
- This means that the subgoal (Q, a) is not yet achieved.
- The subgoal is initially the set of all preconditions of the goal.

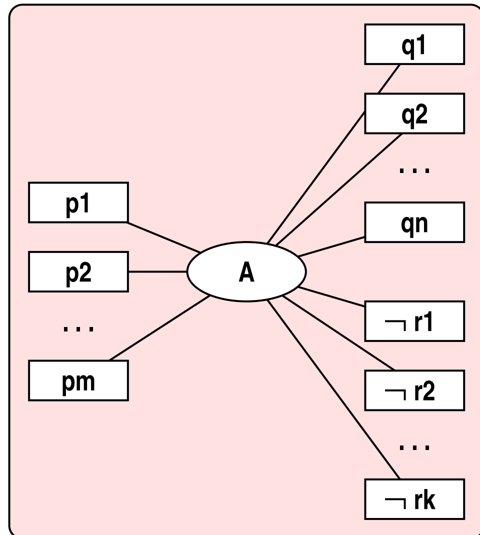
Representation of an action in a plan

Action: A

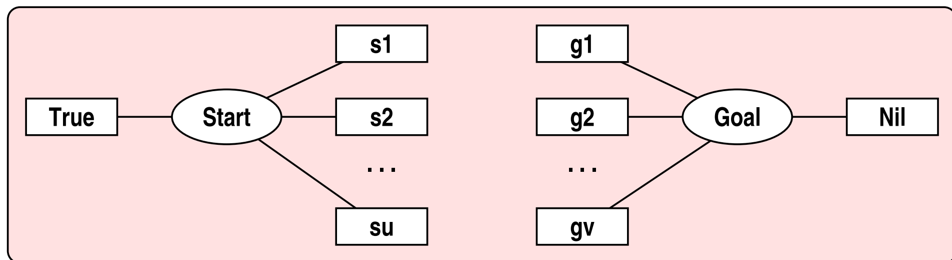
Preconditions: p1, p2, . . . , pm

Add list: q1, q2, . . . , qn

Delete list: r1, r2, . . . , rk



Initialization with the null plan



Start and Goal are dummy actions. Goal initializes the subgoals $SG = \{g_1, g_2, \ldots, g_v\}$. Start sets a set of preconditions, neither of which must be violated in the plan.

Initialize $\mathcal{O}$ with the ordering Start $<$ Goal. All other actions a satisfy Start $< a <$ Goal.

The set $\mathcal{L}$ of causal links is initially empty.

Threats

Consider a causal link $a_p \rightarrow a_q$ with the proposition Q on both the ends. a_p is called the **producer** of the link, and a_q the **consumer** of the link.

A third action a_t is called a **threat** to the link $a_p \rightarrow a_q$ (with Q on both the sides) if the following two conditions hold:

- The ordering $a_p < a_t < a_c$ does not violate any causal-link conditions.
- a_t produces $\neg Q$ as an effect.

If a_t is scheduled between a_p and a_c, then the applicability of a_c after a_p is threatened (in question). This is only a threat (perhaps not a killer), because a fourth action scheduled between a_t and a_c may make Q true again.

Threats are not good for health, and must be dealt with as soon as they are discovered.

- **Promote** a_t (schedule a_t later): $a_p < a_c < a_t$.
- **Demote** a_t (schedule a_t earlier): $a_t < a_p < a_c$.

If neither works, give up (a_t cannot be added to the current plan).

The nondeterministic POP algorithm

Inputs: A partial plan $\mathcal{P} = (\mathcal{A}, \mathcal{O}, \mathcal{L})$ and the subgoals SG.
Output: A complete plan or *failure*.

If SG is empty, return $\mathcal{P} = (\mathcal{A}, \mathcal{O}, \mathcal{L})$.
Choose a subgoal (Q, a) from SG (Q is a precondition in the action $a \in \mathcal{A}$, that waits to be satisfied).
Set $SG = SG - \{(Q, a)\}$.
If a can be linked with an existing action $b \in \mathcal{A}$ that can be scheduled before a, then:

    Set $\mathcal{O}' = \mathcal{O} \cup \{b < a\}$ and $\mathcal{L}' = \mathcal{L} \cup \{(Q, b < a)\}$,

else if there is a new action b that produces Q, do:

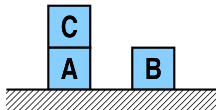
    Set $\mathcal{O}' = \mathcal{O} \cup \{(b < a), (\text{Start} < b), (b < \text{Goal})\}$ and $\mathcal{L}' = \mathcal{L} \cup \{(Q, b < a)\}$,
    Set $\mathcal{A}' = \mathcal{A} \cup \{b\}$, and
    Set $SG' = SG$. For all preconditions P of b, set $SG' = SG' \cup \{(P, b)\}$.

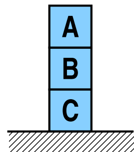
else return *failure*.
For every action $a_t \in \mathcal{A}$ that threatens an existing causal link:

    promote or demote a_t if that can be done consistently,
    else return *failure*.

Recursively call POP on $\mathcal{P}' = (\mathcal{A}', \mathcal{O}', \mathcal{L}')$ and SG'.

Example: Solving the Sussman anomaly by partial-order planning





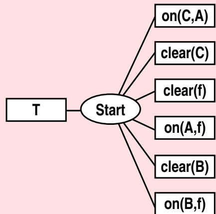
Start

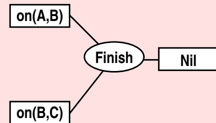
on(A,floor) ∧
on(B,floor) ∧
on(C,A) ∧
clear(B) ∧
clear(C) ∧
clear(floor)

Goal

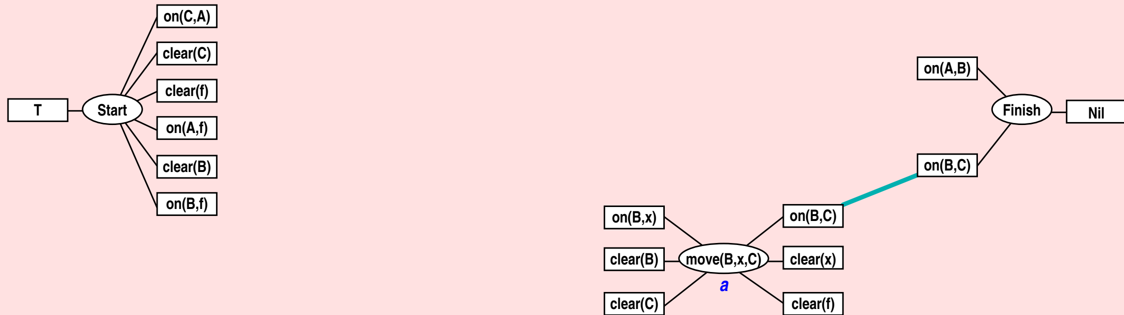
on(A,B) ∧
on(B,C)

The null plan

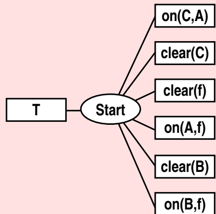


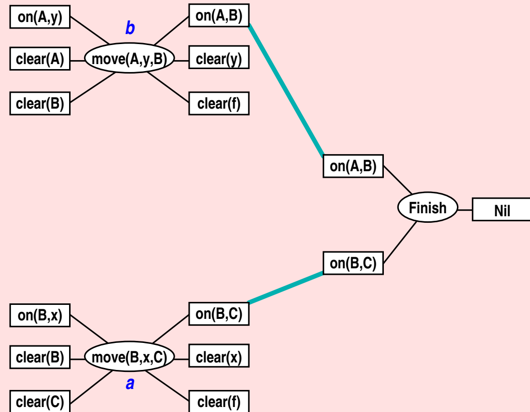


Achieve on(B,C) by moving B from somewhere to C

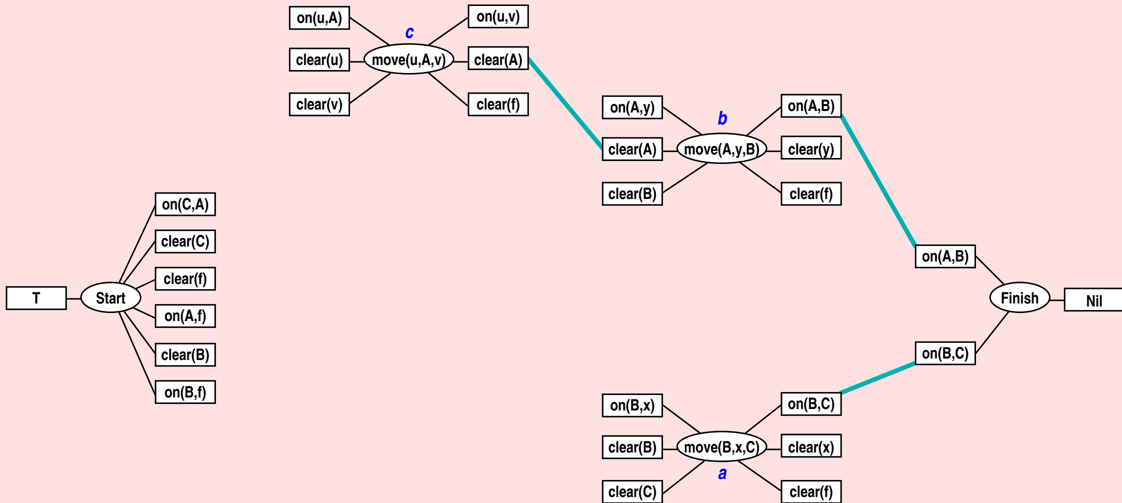


Achieve on(A,B) by moving A from somewhere to B

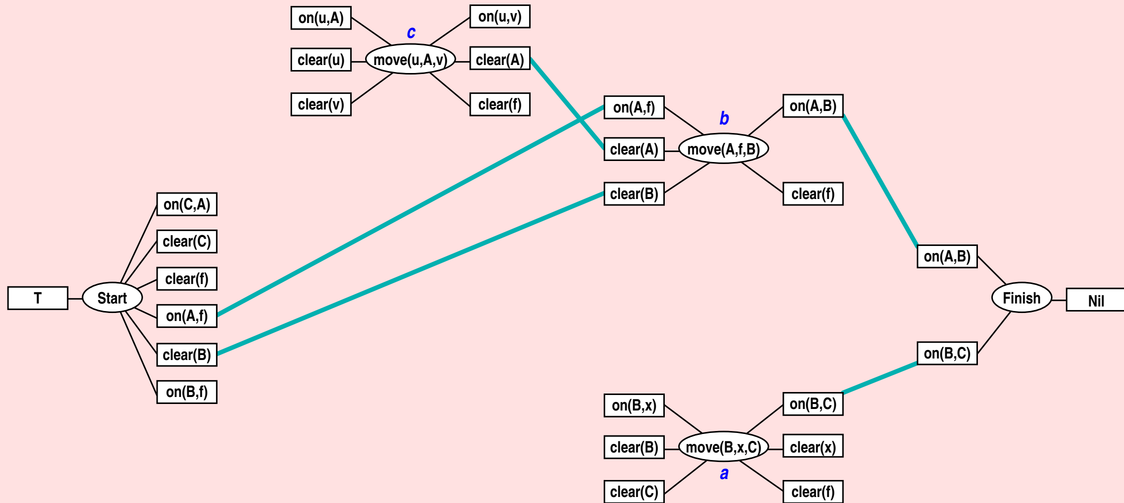




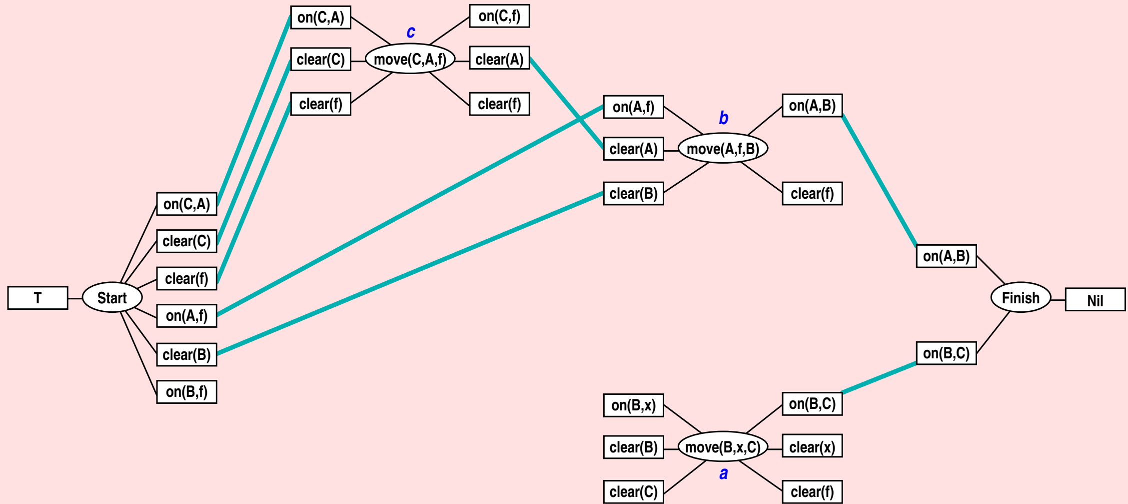
Achieve clear(A) by moving something from A to somewhere



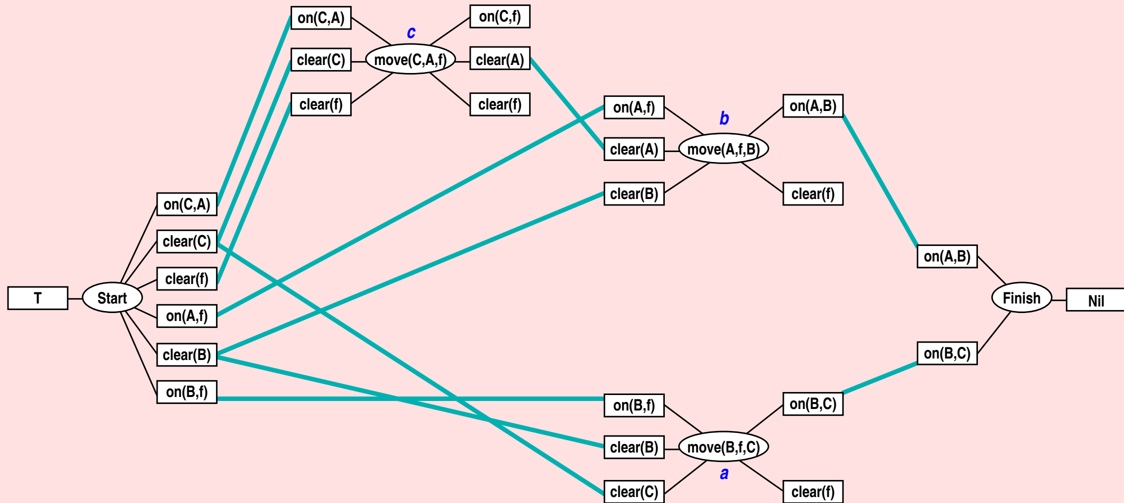
Achieve on(A,y) by the substitution A / f, and connect clear(B)



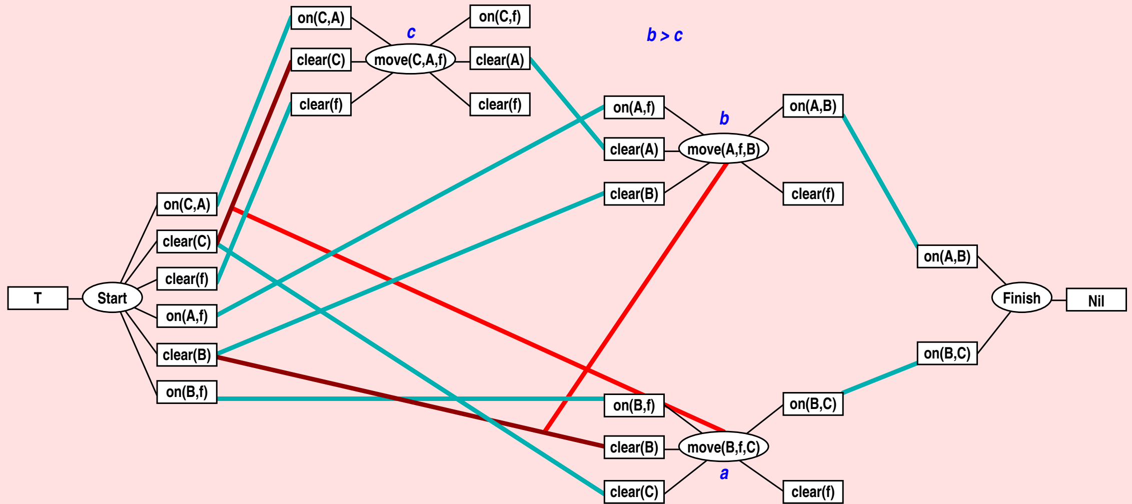
Achieve on(u,A) by the substitution u / C and clear(v) by the substitution v / f



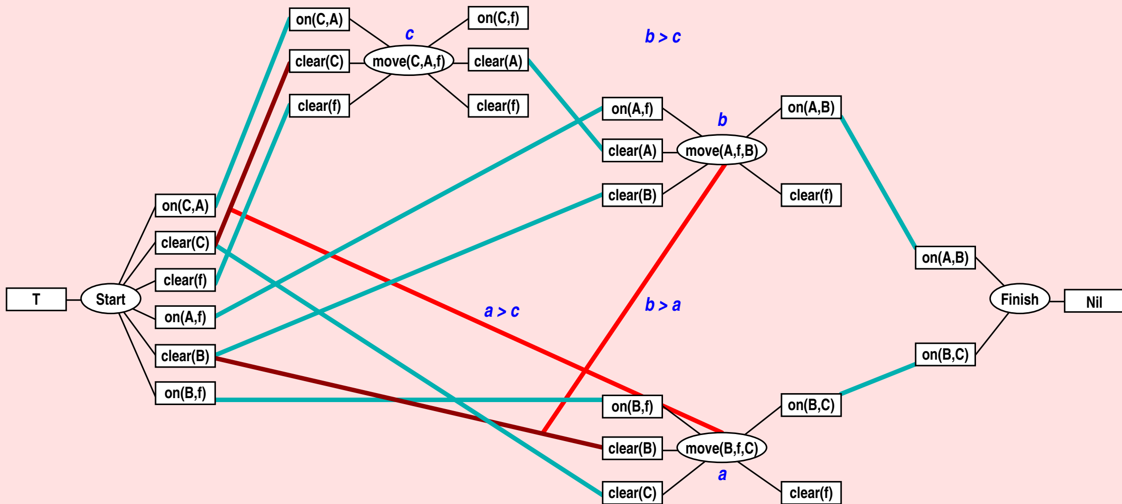
Add more causal links after the substitution x / f



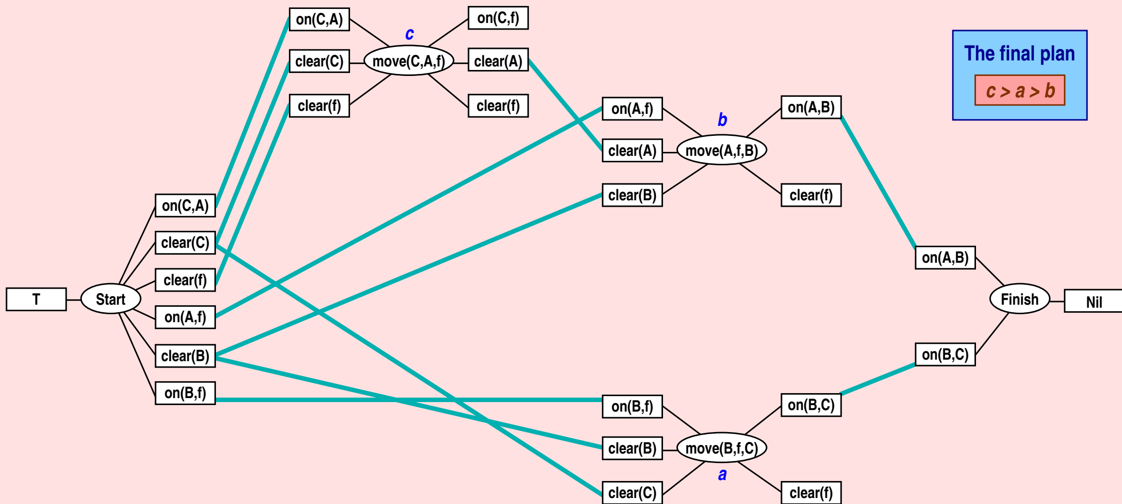
Threats in this plan



Threat removal



The final plan



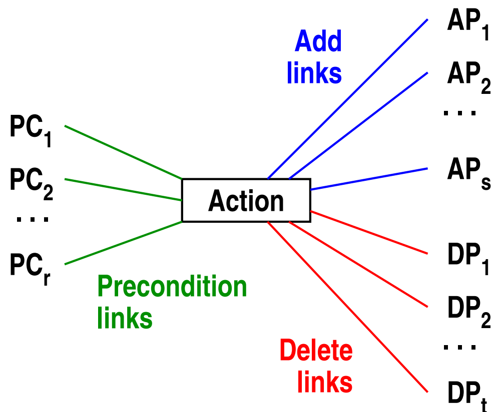
Notes on partial order planning

- The example does not show the deletion lists, but uses these for detecting threat. It is preferable to show the deletion lists as outputs of actions.
- The example handles all the threats at the end. That is not a good idea. Early attempts to remove threats may save time. If some threat cannot be removed consistently, there is no point wasting further time on the partial schedule.
- POP is shown as a nondeterministic algorithm. For such algorithms, a magical entity dictates doing the correct thing at every step.
- In reality, the magical entity does not exist, so the algorithm has to search and find out the correct thing itself.
- The running time is mostly influenced by the branching factor b.
- TOP is too rigid about putting the new actions (always at the beginning). POP is more flexible.
- Analyses suggest that we in general have b(forward search) $>$ b(backward search) = b(total-order planner) $>$ b(partial-order planner).
- POP was used as the most practical planner for about 20 years (mid 1970s to mid 1990s).

GraphPlan

- Based on creating and finding paths in the plan graph.
- The plan graph is neither a state-space graph nor a plan-space graph.
- The plan graph is a leveled graph.
- The levels alternate between those consisting of propositions (literals without variables) and those consisting of actions (without variables).
- The first level is a proposition level consisting of the propositions supplied in the initial state.
- Each action level looks at the propositions of the previous level. If all preconditions are true for the action to proceed, a node is created for that action in that action level. Also the new action node is connected to all its preconditions at the previous level.
- At a proposition level, a proposition appears as a node if and only if it is in the add list or the delete list of an action in the previous action level. A link of add type is established between a proposition and an action that produces it. A link of delete type is established between a proposition and an action that consumes it.
- For each proposition, a special action called no-operation is allowed. The only precondition of the action is that proposition, and its only effect is the production of the same proposition.

The nodes and the links



A real action

A no–operation

The GraphPlan algorithm in a nutshell

Start with a proposition level populated by the propositions in the start state.
Repeat the following steps:
- Add the next action level to the graph.
- Add the next proposition level to the graph.
- If all the propositions in the goal state appear in the new proposition level, do:
  - Try to find a valid plan starting backward from the goal propositions.
  - If a valid plan is found, return the plan.
- Run a feasibility check to detect whether any plan can be developed in future.
- If the check returns *no solutions possible*, return *failure*.

A toy example: Cargo transport

Two cargo items

(a) a box of apples, and

(b) a crate of books

are waiting at the Calcutta airport (C) for dispatch to the Delhi airport (D)

A cargo plane p is at the Delhi airport.

The initial state is: at(a,C) $\wedge$ at(b,C) $\wedge$ at(p,D)

The goal state is: at(a,D) $\wedge$ at(b,D)

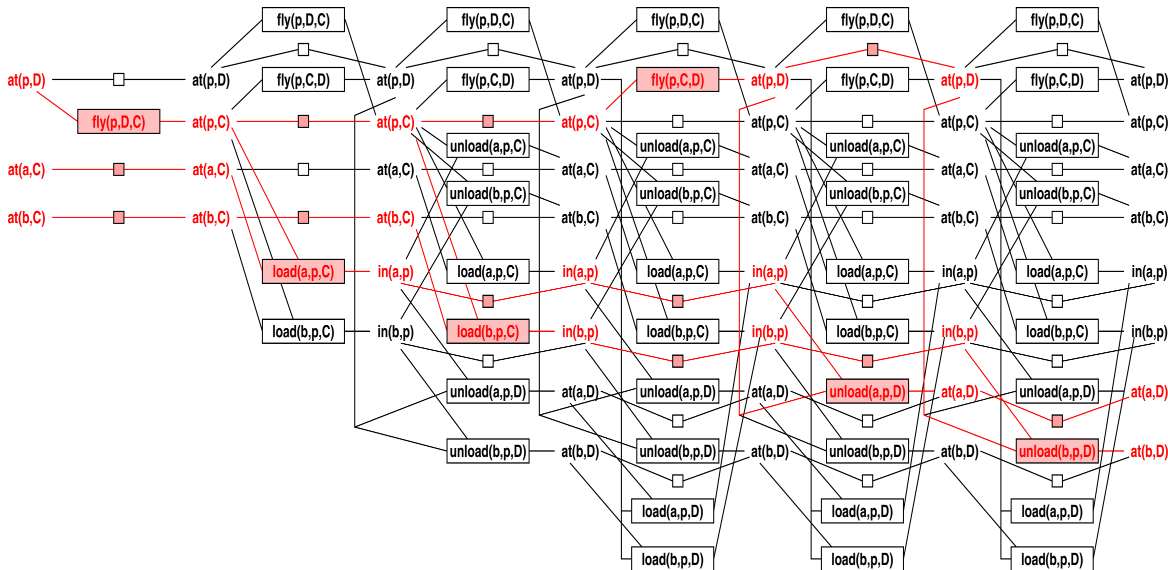
Actions:

fly(p,s,d)
- Preconditions: at(p,s)
- Add list: at(p,d)
- Delete list: at(p,s)

load(c,p,t)
- Preconditions: at(p,t) $\wedge$ at(c,t)
- Add list: in(c,p)
- Delete list: at(c,t)

unload(c,p,t)
- Preconditions: at(p,t) $\wedge$ in(c,p)
- Add list: at(c,t)
- Delete list: in(c,p)

The plan graph and the plan (delete links are not shown)



What does this example tend to show?

Constructing the plan graph is *easy* (doable in time polynomial in the input size and the number of steps to reach the goal).

Extracting a feasible plan from the graph is *difficult*.

- at(a,D) and at(b,D) are available from the fourth proposition level onward. At one unit of time (at each action level), multiple actions cannot be taken. At the third action level, all of the actions fly(p,C,D), load(b,p,C), unload(a,p,D), and unload(b,p,D) cannot all happen together. If we plan to sequentialize these actions, then we must note that all permutations are not feasible. The plane cannot fly without b and unload b at D.
- Multiple actions (including no-op) may lead to a goal proposition. Which one would we take? For each goal proposition? At every step in the backward search? For example, at(b,D) is in the last level for two reasons: no-op and unload(p,b,D). If no-op is chosen, at(b,D) again has two options in the previous level. If unload(p,b,D) is chosen, then both the preconditions at(p,D) and in(b,p) must hold. Each precondition is a result of two actions in the previous level. And so on.
- Can all preconditions for taking an action be simultaneously true at a time instant?

Defining (in)dependence

At any point of time:

> Two actions are called **mutually exclusive** if they cannot be taken together at that time.
>
> Two propositions are called **mutually exclusive** if they cannot be true together at that time.

Define two actions to be mutually exclusive (at a point of time) if:

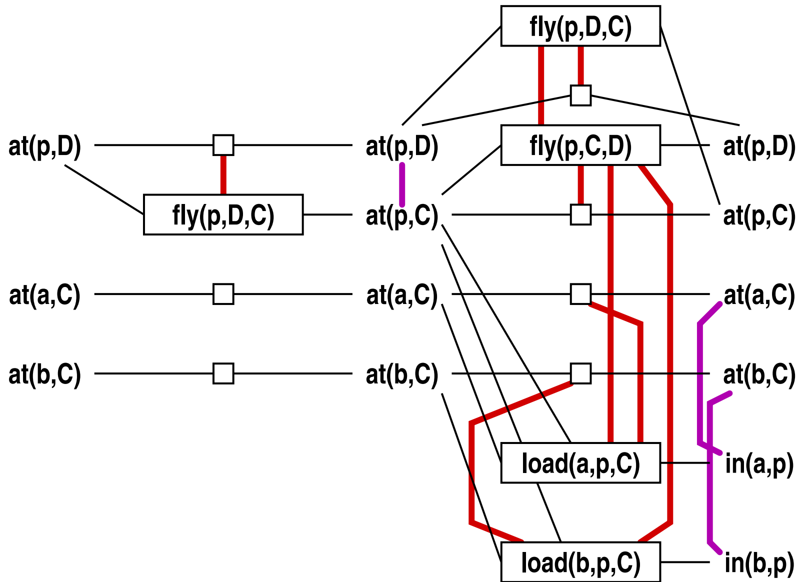
- If either of the two actions deletes a precondition or an add effect of the other.
- The two actions depend on mutually exclusive propositions.

Define two propositions to be mutually exclusive (at a point of time) if:

- The two propositions are complements of one another.
- All ways of producing the two propositions are mutually exclusive.

Mutual exclusion helps not only in finding feasible plans, but also in reducing the size of the plan graph.

Example of mutual exclusion



Notes on GraphPlan

- Need to maintain a list of incompatible (mutually exclusive) others for each item at each level.
- Finding mutually exclusive pairs is computationally easy.
- Mutual exclusion may reduce the size of the graph.
- A solution can be searched for only if all the goal propositions are available in the mutually non-exclusive form. If a planning problem is solvable, this would happen at some level.
- Finding a path from the goal predicates (in case of no mutual exclusion and solvable) is computationally hard. You need to ensure that every action on the path uses mutually non-exclusive preconditions. This is a constraint-satisfaction problem.
- Despite that, GraphPlan is shown to be much faster than POP.
- If a planning problem is not solvable, GraphPlan can detect that. So GraphPlan is sound and complete and total.

SATPlan

This is a *non-deductive* planning algorithm. It involves no search.

SATPlan converts the planning instance to a propositional-logic formula, and then invokes a SAT solver on this formula. The formula involves no variables.

If the formula is satisfiable, then a satisfying truth assignment is returned. The plan is extracted from the satisfying truth assignment.

If the formula is not satisfiable, then the planning problem has no solution.

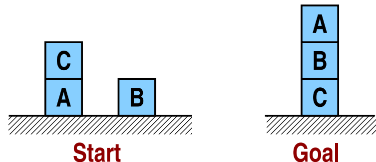
Propositionalization

- The initial state is converted to a conjunct of positive and negative literals.
- The actions are converted to implication statements at every unit of time.
- The maximum time units (steps) allowed is pre-specified as N.
- We need to ensure that at each time unit, only one action can be taken.
- We handle the situation that some propositions remain unaffected by moves.
- Propositionalize the goal.

Propositionalize the start state and the goal state

Create all the positive and negative literals at time 0.

The goal state need not be fully explicit.



$$\begin{gathered}\text{on}(A,\text{floor},0) \wedge \neg\text{on}(A,B,0) \wedge \neg\text{on}(A,C,0) \wedge \\ \text{on}(B,\text{floor},0) \wedge \neg\text{on}(B,A,0) \wedge \neg\text{on}(B,C,0) \wedge \\ \neg\text{on}(C,\text{floor},0) \wedge \text{on}(C,A,0) \wedge \neg\text{on}(C,B,0) \wedge \\ \neg\text{clear}(A,0) \wedge \text{clear}(B,0) \wedge \text{clear}(C,0) \wedge \text{clear}(\text{floor},0) \wedge \\ \text{on}(A,B,N) \wedge \text{on}(B,C,N)\end{gathered}$$

The goal may involve logical ORs. For example, the goal "A is on some other block and B is on some other block" can be encoded as:

$$\left(\text{on}(A,B) \vee \text{on}(A,C)\right) \wedge \left(\text{on}(B,A) \vee \text{on}(B,C)\right)$$

Propositionalize actions

If an action takes place at time t, then the preconditions must be true at time t, and the propositions in the delete list must be false and the propositions in the add list must be true at time $t+1$.

Example: move(x,y,z)

Preconditions: on(x,y) $\wedge$ clear(x) $\wedge$ clear(z)
Delete list: on(x,y) $\wedge$ clear(z)
Add list: on(x,z) $\wedge$ clear(y) $\wedge$ clear(floor)

Here, x can be instantiated from {A,B,C}, y can be instantiated from {A,B,C,floor}, z can be instantiated from {A,B,C,floor}, and this may happen at any time $t = 0, 1, 2, \ldots, N-1$. You must convert the action for all legal instantiations.

move(B,C,floor) at time 2 is encoded as:

$$\text{move(B,C,floor,2)} \Rightarrow \Big(\text{on(B,C,2)} \wedge \text{clear(B,2)} \wedge \text{clear(floor,2)}\Big) \bigwedge \Big(\neg\text{on(B,C,3)}\Big) \bigwedge \Big(\text{on(B,floor,3)} \wedge \text{clear(C,3)} \wedge \text{clear(floor,3)}\Big)$$

Only one action at a time

Let $A_1, A_2, \ldots, A_r$ be all the propositionalized actions at time t. Then, we must have the formula:

$$\begin{aligned}
&\left(A_1(\ldots,t) \Rightarrow \neg\left(A_2(\ldots,t) \vee A_3(\ldots,t) \vee \cdots \vee A_r(\ldots,t)\right)\right)\\
\bigwedge &\left(A_2(\ldots,t) \Rightarrow \neg\left(A_1(\ldots,t) \vee A_3(\ldots,t) \vee \cdots \vee A_r(\ldots,t)\right)\right)\\
\bigwedge &\cdots\\
\bigwedge &\left(A_r(\ldots,t) \Rightarrow \neg\left(A_1(\ldots,t) \vee A_2(\ldots,t) \vee \cdots \vee A_{r-1}(\ldots,t)\right)\right)
\end{aligned}$$

This can be rewritten as

$$\bigwedge_{1 \leqslant i < j \leqslant r} \neg\left(A_i(\ldots,t) \wedge A_j(\ldots,t)\right)$$

No-operation on a proposition

A proposition P can be true at time $t+1$ if and only if one of the following happens at time t.

- The action taken at time t introduced P as a member of its add list.
- P was already true at time t, and the action taken at time t did not have P in its delete list.

Let $A_1, A_2, \ldots, A_r$ be all the propositionalized actions at time t, that contain P in their add lists. Let $B_1, B_2, \ldots, B_s$ be all the propositionalized actions at time t, that contain P in their delete lists.

$$P(\ldots, t+1) \Leftrightarrow \Big(A_1(\ldots, t) \vee A_2(\ldots, t) \vee \cdots \vee A_r(\ldots, t)\Big) \bigvee \Big(P(\ldots, t) \bigwedge \neg\big(B_1(\ldots, t) \vee B_2(\ldots, t) \vee \cdots \vee B_s(\ldots, t)\big)\Big)$$

$$\text{on(B,floor,2)} \Leftrightarrow \Big(\text{move(B,A,floor,1)} \vee \text{move(B,C,floor,1)}\Big) \bigvee \Big(\text{on(B,floor,1)} \bigwedge \neg\big(\text{move(B,floor,A,1)} \vee \text{move(B,floor,C,1)}\big)\Big)$$

The final steps

The final formula is the AND of all the formulas introduced so far.

The formula is rather big (compared to the PDDL). There are too many variables at every time t.

With propositions and actions of constant arities, the size of the formula is polynomial in the input size and N (the number of steps).

Modern SAT solvers are rather efficient except in esoteric cases.

Extracting a plan (if it exists)

- Look at the actions taken at each time $t \in \{0, 1, 2, \ldots, N-1\}$.
- There can be at most one action at each time.
- A satisfiable formula will indicate that.
- Ignore the no-action steps.
- You get your plan.

We do not know N beforehand

A plan exists

- Let n be the minimum number of actions in a plan for the given problem.
- Try for $N = 1, 2, 4, 8, 16, \ldots$ until you get a plan for $N = 2^k$.
- You have then identified a k such that $2^{k-1} < n \leqslant 2^k$.
- If you are happy, you are done.
- Otherwise, stage a binary search with N in the range $(2^{k-1}, 2^k]$.
- You need $\mathrm{O}(\log n)$ invocations of the SATPlan procedure.

A plan does not exist

- Running SATPlan for $N = 1, 2, 4, 8, 16, \ldots$ will never reveal a plan.
- How long will you keep on trying?
- If an upper bound on n is known beforehand, search with N up to that bound.
- SATPlan is sound and complete, but not total.