Partial ordered plan

Planning and acting in the real world

Scheduling

Resource

Earliest Start ES

Latest Start time LS

Slack LS-ES

Planning and acting in the real world

- How can we make a plan when the actions are associated with durations and resource constraints?

- How can a plan be organised hierarchically?
  - Involve human inputs
  - Work at an abstract level

- How can an agent handle uncertain environment?

Time, Schedules and Resources

- Classical planning talks about what to do, and in what order
  - It does not talk about the duration and the start time of an action
  - Therefore it cannot be used for scheduling actions

- Classical planning does not have an easy way to account for the resource constraints or deadline constraints.

- We follow an approach of
  - Plan first (Planning Phase): select actions and order them so as to achieve the goals.
  - Schedule later (Scheduling Phase): Temporal information is added to the plan to meet the resource and deadline constraints.

Representing temporal and resource constraints

$$Jobs(\{AddEngine1 \prec AddWheels1 \prec Inspect1\},$$
$$\{AddEngine2 \prec AddWheels2 \prec Inspect2\})$$

$$Resources(EngineHoists(1), WheelStations(1), Inspectors(2), LugNuts(500))$$

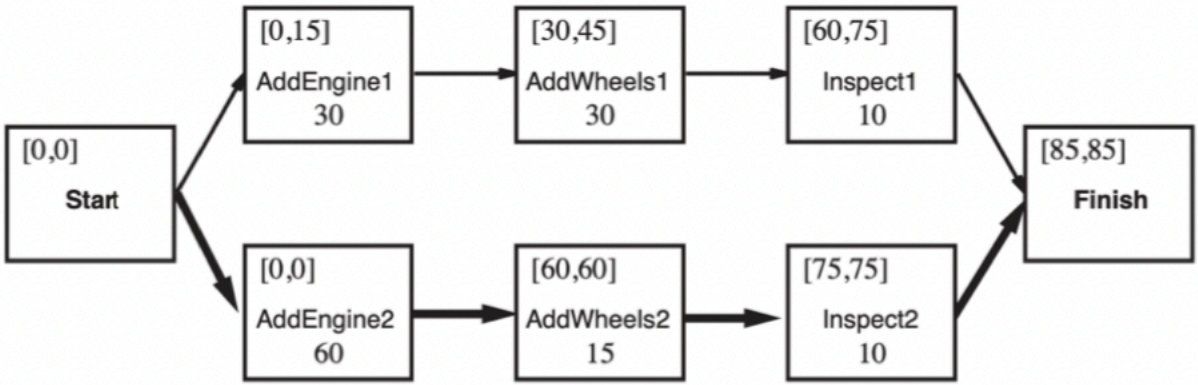
$$Action(AddEngine1, \text{DURATION}:30,$$
$$\text{USE}:EngineHoists(1))$$
$$Action(AddEngine2, \text{DURATION}:60,$$
$$\text{USE}:EngineHoists(1))$$
$$Action(AddWheels1, \text{DURATION}:30,$$
$$\text{CONSUME}:LugNuts(20), \text{USE}:WheelStations(1))$$
$$Action(AddWheels2, \text{DURATION}:15,$$
$$\text{CONSUME}:LugNuts(20), \text{USE}:WheelStations(1))$$
$$Action(Inspect_i, \text{DURATION}:10,$$
$$\text{USE}:Inspectors(1))$$

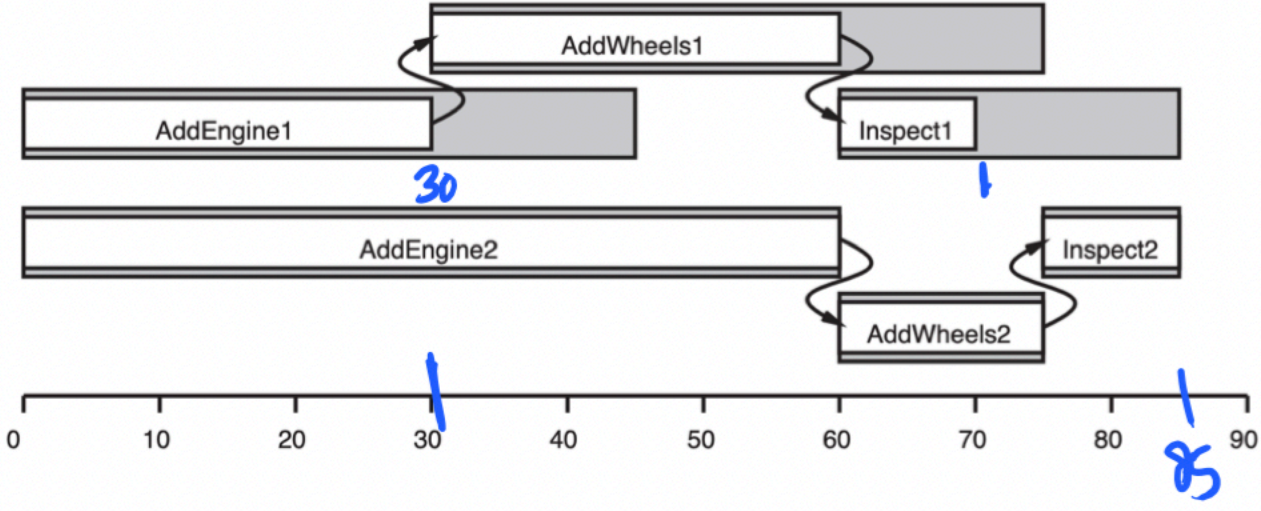
- A resource can be consumable or reusable
- Resources can also be produced by actions
- For a job shop scheduling problem we prefer a solution with the shortest total duration of the plan.
    - The total duration of a plan is called as makespan.

Solving scheduling problems

- To minimize makespan (plan duration), we must find the earliest start times of all the actions consistent with the ordering constraints
- Ordering constraints can be represented as a directed graph relating the actions.
- A path through a graph representing a partial-order plan is a linearly ordered sequence of actions beginning with Start and ending with Finish.
- A critical path is that path whose total duration is longest.

- The critical path determines the duration of the entire plan.
- Delaying the start of any action on the critical path slows down the whole plan.
- Actions that are off the critical path have a window (slack) of time in which they can be executed.
  - *ES:* earliest possible start time
  - *LS:* latest possible start time
  - LS - ES is known as the slack of an action


[0,0]
Start
[0,15]
AddEngine1
30
[30,45]
AddWheels1
30
[60,75]
Inspect1
10
[0,0]
AddEngine2
60
[60,60]
AddWheels2
15
[75,75]
Inspect2
10
[85,85]
Finish


AddWheels1
AddEngine1
Inspect1
30
1
AddEngine2
Inspect2
AddWheels2
1
1
0
10
20
30
40
50
60
70
80
90
85

- The following formulas serve as a definition of ES and LS
- A and B are actions and $A \prec B$ means that A comes before B

$$
\begin{aligned}
&ES(Start) = 0 \\
&ES(B) = \max_{A \prec B} ES(A) + Duration(A) \\
&LS(Finish) = ES(Finish) \\
&LS(A) = \min_{B \succ A} LS(B) - Duration(A)
\end{aligned}
$$

- We start by assigning $ES(\text{Start})$ to be 0

- As soon as we get an action B such that all the actions that come immediately before B have ES values assigned, we set $ES(B)$ to the maximum of the earliest finish times of those immediately preceding actions

$$ES(B) = \max_{A \prec B} ES(A) + Duration(A)$$

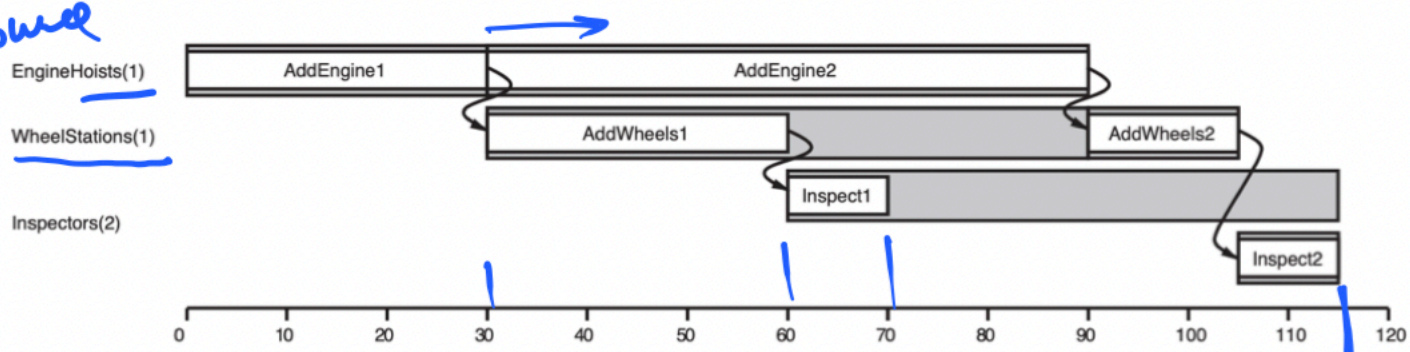
- Earliest finish time of an action is defined as the earliest start time plus the duration.
- The process repeats until every action has been assigned the ES value.

- The LS values are computed in a similar manner, working backward from the *Finish* action.

$$LS(Finish) = ES(Finish)$$
$$LS(A) = \min_{B \succ A} LS(B) - Duration(A)$$

- Though it is easy to find a minimum-duration schedule given a partial ordering on the actions, note that this method cannot deal with resource constraints.

- With the introduction of resource constraints
  - the constraints are no longer conjunctions of linear inequalities
  - For example, the 'cannot overlap' constraint is a disjunction of two linear inequalities.
- A popular heuristic is the minimum slack algorithm
  - On each iteration
    - Examine the unscheduled actions which have all predecessors scheduled
    - Choose the one which has the least slack. Schedule it for the earliest possible start.
    - Update the ES and LS times for each affected action
    - Repeat


EngineHoists(1)
WheelStations(1)
Inspectors(2)
AddEngine1
AddEngine2
AddWheels1
AddWheels2
Inspect1
Inspect2
0
10
20
30
40
50
60
70
80
90
100
110
120

- It is also possible to design algorithms that can integrate planning and scheduling by taking into account durations and overlaps during the construction of a partial-order plan.

Hierarchical Planning

- Planning becomes computationally expensive as the number of atomic actions increases.
- A human brain needs to make detailed motor plans by scheduling muscle activations.
  - There are 10^3 muscles
  - These muscles can be activated 10 times per second
  - We are alive and awake for 10^9 seconds
  - Thus, a human life contains about 10^{13} actions (or within one or two orders of magnitude)

A two-week vacation in Hawaii

- A detailed motor plan will have 10^{10} actions.

- However, at higher levels of abstraction the plan becomes simpler

  "Go to San Francisco Airport;

  Take Hawaiian Airlines flight 11 to Honolulu;

  Do vacation stuff for two weeks;

  Take Hawaiian Airlines flight 12 back to San Francisco;

  Go home."

Further planning the abstract actions

- Go to San Francisco Airport
  - Drive to the long-term parking lot
  - Park
  - Take the shuttle to the terminal

- Each of these actions can be further decomposed to generate the motor level control sequences
- Thus, planning can occur both before and during the execution of the plan

Hierarchical Decomposition

- Complex software is created from a hierarchy of subroutines or object classes.
- Armies, government, corporations operate in hierarchies.

- A task at a given level of the hierarchy is decomposed into a small number of activities at the next lower level.
    - This helps in managing complexity

- Non-hierarchical methods reduce a task to a large number of individual actions.

High-level action (HLA)

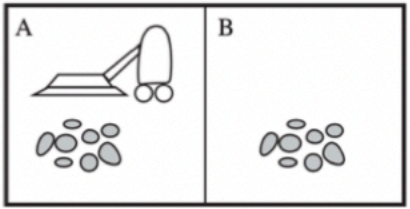
- A primitive action does not require any refinement.

- An HLA has one or more possible refinements into a sequence of actions - each of which can be an HLA or a primitive action.

$$
\begin{aligned}
&\mathit{Refinement}(\mathit{Go}(\mathit{Home}, \mathit{SFO}),\\
&\quad \textsc{Steps}: [\mathit{Drive}(\mathit{Home}, \mathit{SFOLongTermParking}),\\
&\qquad\qquad \mathit{Shuttle}(\mathit{SFOLongTermParking}, \mathit{SFO})]\ )\\
&\mathit{Refinement}(\mathit{Go}(\mathit{Home}, \mathit{SFO}),\\
&\quad \textsc{Steps}: [\mathit{Taxi}(\mathit{Home}, \mathit{SFO})]\ )
\end{aligned}
$$

Recursive refinement to navigate in Vacuum World

$Refinement(Navigate([a,b],[x,y]),$
PRECOND: $a = x \ \wedge \ b = y$
STEPS: [ ])
$Refinement(Navigate([a,b],[x,y]),$
PRECOND: $Connected([a,b],[a-1,b])$
STEPS: $[Left, Navigate([a-1,b],[x,y])]$)
$Refinement(Navigate([a,b],[x,y]),$
PRECOND: $Connected([a,b],[a+1,b])$
STEPS: $[Right, Navigate([a+1,b],[x,y])]$)
...



- *Navigate*([1,3][3,2]) can be implemented as

  [Right, Right, Down] or

  [Down, Right, Right]

Implementation of an HLA

- An HLA refinement that contains only primitive actions is called an implementation of the HLA.

- An implementation of a a high level plan (sequence of HLAs) is the concatenation of implementations of each HLA in the sequence.

- A high level plan achieves the goal from a given state if at least one of its implementations achieves the goal from that state.
    - This can be determined by checking the precondition-effect definitions of each primitive action in the plan.

- Not all implementations of the high level plan need to achieve the goal.
- The agent can select an implementation which will achieve the goal.
- If an HLA has exactly one implementation we can treat it as if it were a primitive action itself.
- If an HLA has multiple possible implementations, then the agent needs to search among the implementations for the one that works.

Searching for a Plan

- We can search for a plan in terms of primitive actions

  OR

- We can search for a plan in terms of high-level actions (HLAs)

- HTN (Hierarchical Task Network) planning

  Basic Algorithm:

  Repeat the following until the plan achieves the goal:

  Choose an HLA in the current plan and replace it with one of its refinements

Choose an HLA in the current plan and replace it with one of its refinements

→ BFS

→ HS..

→ AS..

Searching for primitive solutions

function HIERARCHICAL-SEARCH(*problem*, *hierarchy*) **returns** a solution, or failure

frontier ← a FIFO queue with [*Act*] as the only element
loop do
  if EMPTY?(*frontier*) **then return** failure
  plan ← POP(*frontier*) /* chooses the shallowest plan in *frontier* */
  hla ← the first HLA in *plan*, or *null* if none
  prefix,*suffix* ← the action subsequences before and after *hla* in *plan*
  outcome ← RESULT(*problem*.INITIAL-STATE, *prefix*)
  if *hla* is null **then** /* so *plan* is primitive and *outcome* is its result */
    if *outcome* satisfies *problem*.GOAL **then return** *plan*
  else for each *sequence* **in** REFINEMENTS(*hla*, *outcome*, *hierarchy*) **do**
    frontier ← INSERT(APPEND(*prefix*, *sequence*, *suffix*), *frontier*)

possible refinements

HLA

HLA, HLA2 . . .

frontier Prefix HLA suffix

outcome.

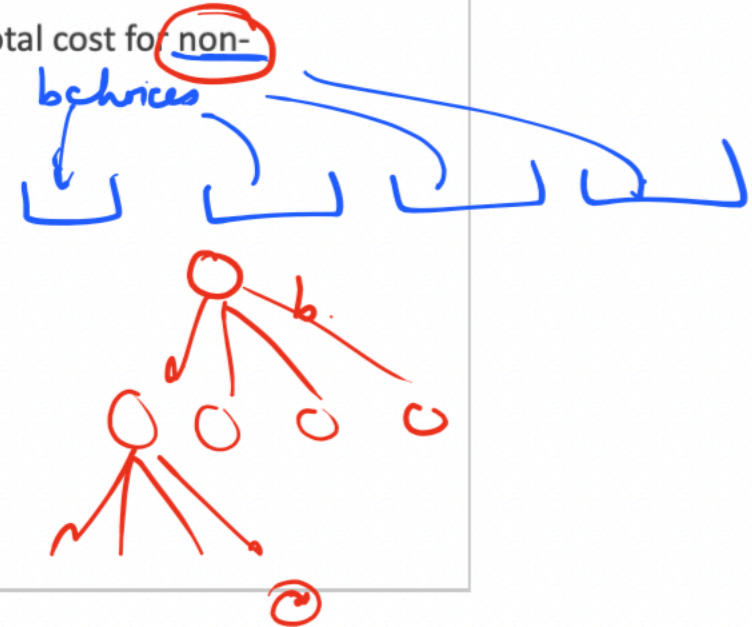
Computational cost of non-hierarchical search

Computational cost of non-hiearchical search

- Consider a planning problem which has the solution (plan) with d steps of primitive actions.
- If there are b allowable actions at each step, then the total cost for non-hierarchical search is $O(b^d)$.

1) No HLA. $O(b^d)$ d step

2) With HLA.



Computational Benefits of Hierarchical Search

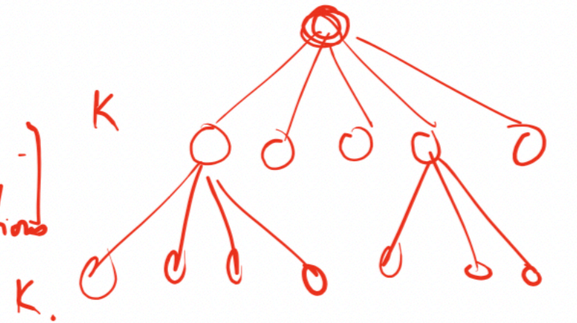
- Consider a regular refinement structure
  - Each non-primitive action has r possible refinements.
  - A refinement results in k actions at the next lower level.
  - How many possible refinement trees can be constructed?

- If there are d actions at the primitive level, then the recursive decomposition will have $\log_k d$ levels below the root.
- So the number of internal refinement nodes is $1 + k + k^2 + \ldots + k^{\log_k d - 1}$

- Each internal node has r possible refinements

HLA

k actions



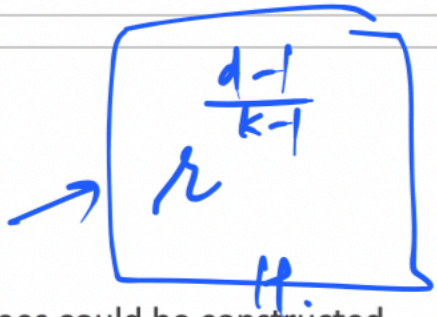
$1 + k + k^2 + k^3 + \cdots + k^{L-1} = \frac{k^L - 1}{k-1}$

$\frac{1}{1-k} - \frac{d}{1-k} = \frac{1-d}{1-k}$

$k^L = d$

$L = \log_k d$

- Each internal node has r possible refinements



- So $r^{(d-1)/(k-1)}$ possible regular decomposition trees could be constructed.

- Keeping r small and k large can result in huge savings.

- Small r and large k means a library of HLAs with a small number of refinements each yielding a long action sequence.

- Long action sequences usable across a wide range of problems are extremely precious.

reusability

- While constructing the HLA plan library, it is important to generalize the sequence of actions so that the details specific to a given problem instance can be avoided.

Library

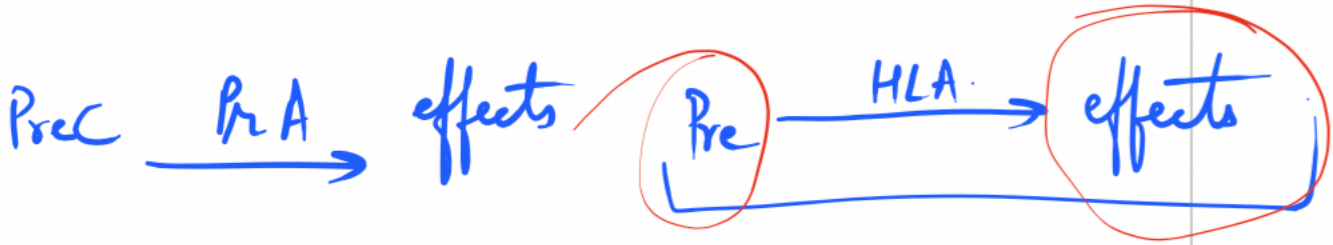
Generalizability

Searching for Abstract Solutions

- Without getting into ground level details, can we determine whether a high-level plan is able to achieve the goal?

$$[Drive(Home, SFOLongTermParking), Shuttle(SFOLongTermParking, SFO)]$$

- Yes we can, if we have the precondition-effect descriptions of the HLAs.



- If we can prove that a high level plan can achieve the goal, then we can commit to the plan and then take up refining each step of the plan.

- In that case we are guaranteed that at least one implementation of the high-level plan will achieve the goal.

- This is called as the downward refinement property for HLA descriptions.
    - Such guarantees are possible if the HLA descriptions are true and do not make false claims.

But

- How can we describe the effects of an action that can be implemented in many different ways?

Conservative Approach:

- Include only the positive effects that are achieved by every implementation of the HLA and the negative effects of any implementation.
- It treats as if HLAs were a non-deterministic action.

[Demonic Non-determinism] Equivalent to assuming that an adversary makes the choice.

Home → Taxi → Airport

Taxi → Drive

Common to every implementation

Union of the negative effects

For our problem of planning

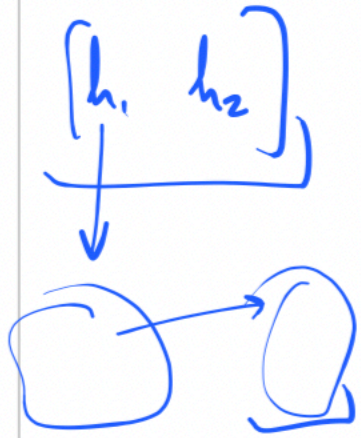
- We shall assume angelic nondeterminism
- The agent makes its choice for the HLA implementation
- To specify angelic semantics for HLA descriptions
  - We define a reachable set of an HLA
  - REACH(*s*,*h*) is the set of states reachable by any of the HLA *h*'s implementations from state s.

Angelic Semantics

- The agent can choose the implementation based on the element of the reachable set (target) it wants to achieve.

- The reachable set of a sequence of HLAs $[h_1, h_2]$ is the union of all the reachable sets obtained by applying h_2 in each state in the reachable set of h_1

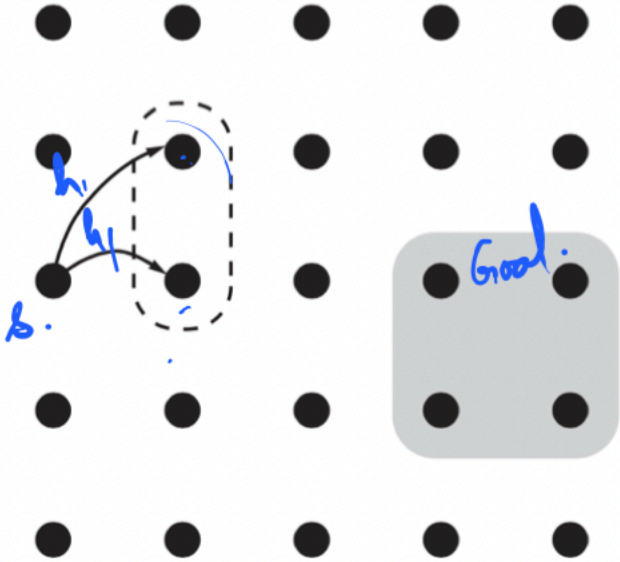
$$\text{REACH}(s, [h_1, h_2]) = \bigcup_{s' \in \text{REACH}(s, h_1)} \text{REACH}(s', h_2)$$

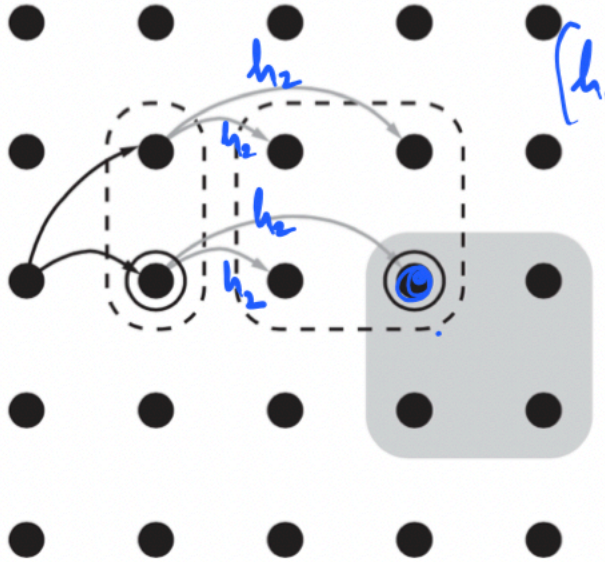


Can an HLA achieve the goal?

- A high-level plan (a sequence of HLAs) achieves a goal if its reachable set intersects the set of goal states.

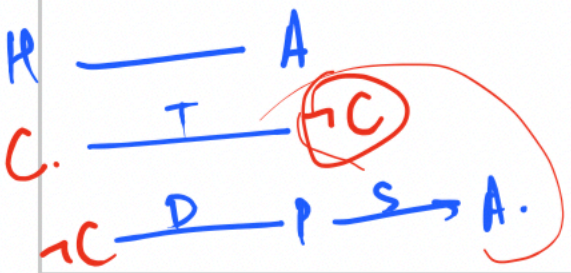
- Conversely, if the reachable set doesn't intersect the goal, then the plan definitely doesn't work.


h_1
h_1
s.
Good.


h_2
h_2
h_2
h_2
$[h_1 \quad h_2]$

Algorithm for searching abstract solutions (HLA plans)

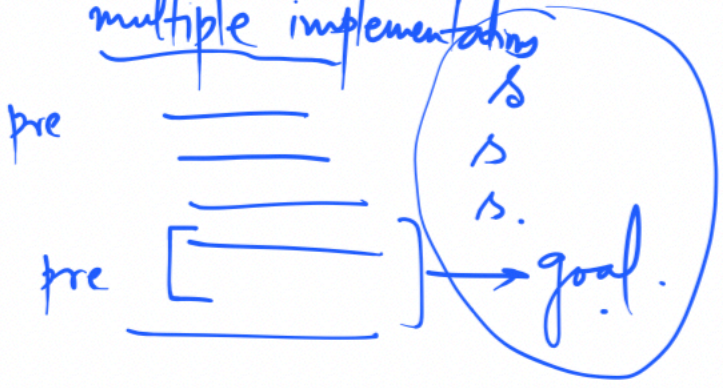
- Search among high-level plans, looking for the one whose reachable set intersects the goal.
- Once that happens, the algorithm can commit to that abstract plan.
- The committed plan is refined further.



Reachable set

REACH(h).

multiple implementations



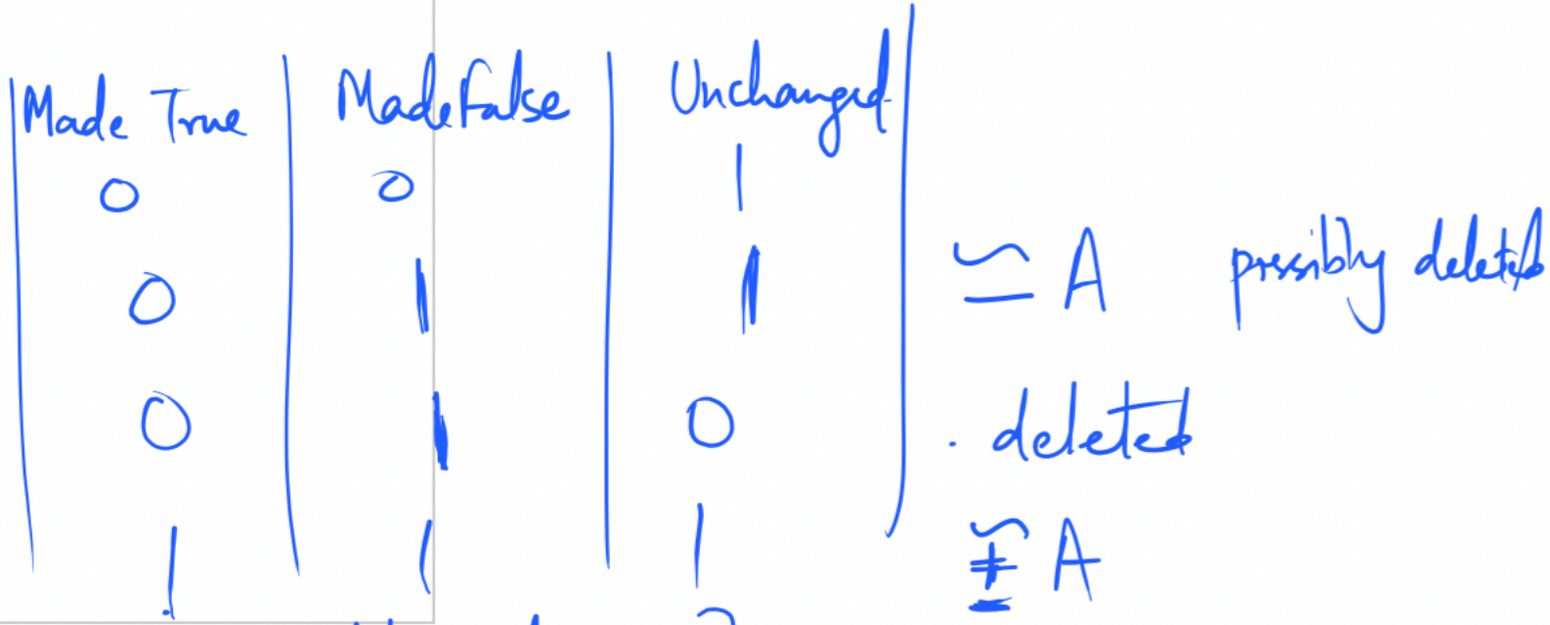
Representing the reachable set

- For a given HLA, we need to mention the reachable set for each possible initial state.
- We represent the changes made to each fluent (variable).
- A primitive action can *add* or *delete* a variable, or leave it *unchanged*.
- Therefore, an HLA can have the following effects on a variable
  - For a variable which is initially true [false], it can
    - Keep it true [keep it false]
    - Make it false [make it true]
    - Have a choice [have a choice]

- The three possible effects can be made combined arbitrarily, making 9 different effects on a variable.

Notation Used

- ~ symbol means possibly if the agent so chooses
- $\tilde{+}A$ means possible add A
- $\tilde{-}A$ means possible delete A
- $\tilde{\pm}A$ means possible add or delete A



- For example, the HLA $Go(Home, SFO)$ with its two refinements possibly deletes cash, so it should have the effect $\simeq Cash$

- Thus, we can derive the descriptions of HLAs from the descriptions of their refinements.

Example

- Suppose we have the following schemas for the HLAs h_1 and h_2

$$Action(h_1, \text{PRECOND:} \neg A, \text{EFFECT:} A \wedge \tilde{-}B),$$
$$Action(h_2, \text{PRECOND:} \neg B, \text{EFFECT:} +A \wedge \pm C)$$

- If B is true in the initial state and the goal is $A \wedge C$ then the sequence $[h_1, h_2]$ achieves the goal
- We choose an implementation of h_1 that makes B false, then choose an implementation of h_2 that leaves A true and makes C true.

- The reachable set for any given initial state can be described exactly by describing the effect on each variable.
- However, an HLA can have infinite implementations and may produce arbitrary reachable sets.
- For example *Go(Home, SFO)* possibly deletes *Cash*; it also possibly adds *At(Car, SFOLongTermParking)*; but it cannot do both.

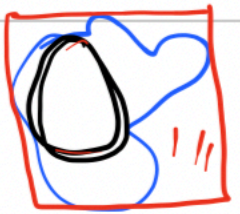
Belief state 1-CNF

literals.

Reachable sets

wiggly sets

Approximate descriptions of reachable sets



- We can use two kinds of approximation
  - An optimistic description $\text{REACH}^{+}(s, h)$

    It may overstate the reachable set.
  - A pessimistic description $\text{REACH}^{-}(s, h)$

    It may understate the reachable set.

$$\text{REACH}^{-}(s, h) \subseteq \text{REACH}(s, h) \subseteq \text{REACH}^{+}(s, h)$$

Pessimistic exact optimistic

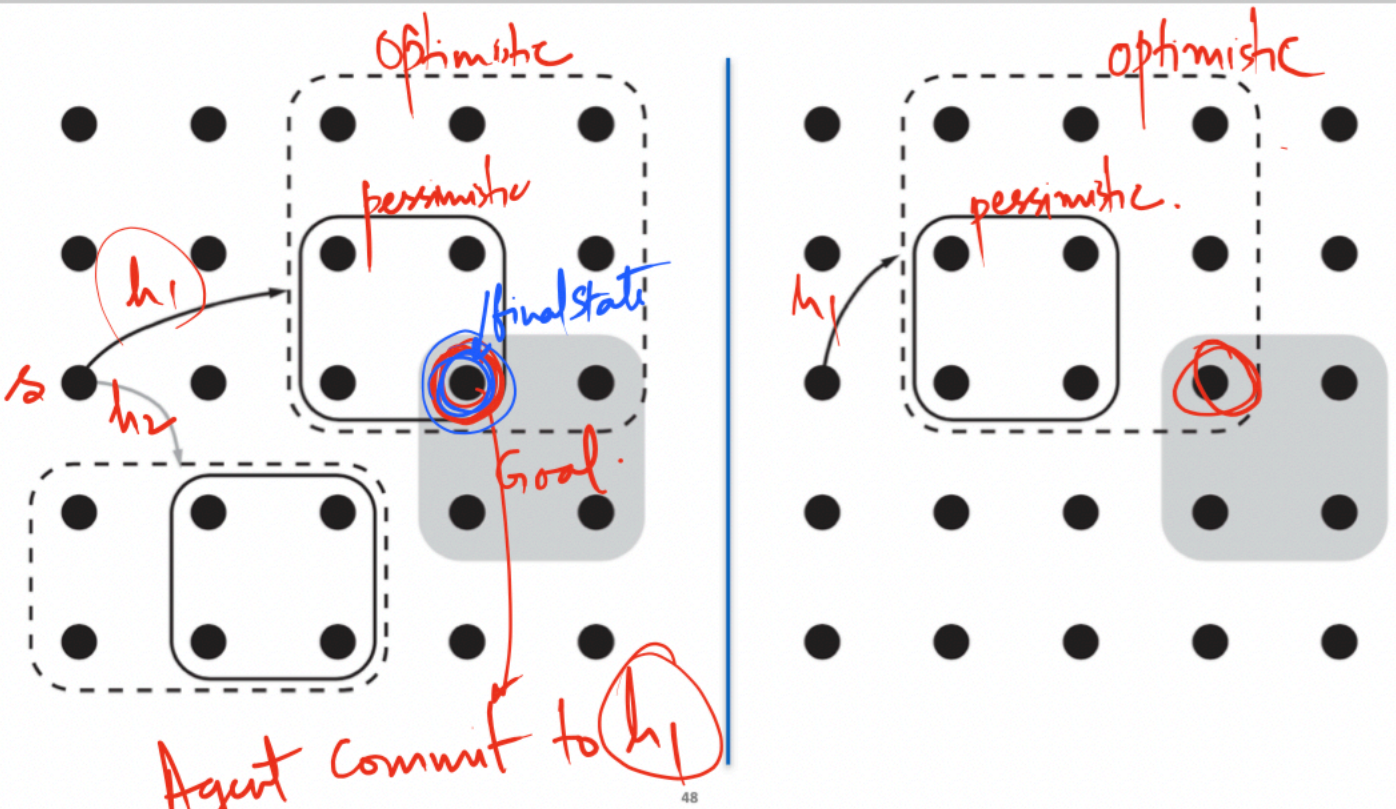
Approximate Descriptions

- For example, an optimistic description of $Go(Home, SFO)$ says that it
  - possibly deletes $Cash$, and
  - possibly adds $At(Car, SFOLongTermParking)$

- With approximate descriptions, we can test whether a plan achieves the goal as follows:
  - If the optimistic reachable set for the plan doesn't intersect the goal, then the plan doesn't work.
  - If the pessimistic reachable set for the plan intersects the goal, then the plan does work.

- If the optimistic set intersects the goal and the pessimistic set doesn't, then there is uncertainty if the plan works.

- The uncertainty can be resolved by refining the plan.

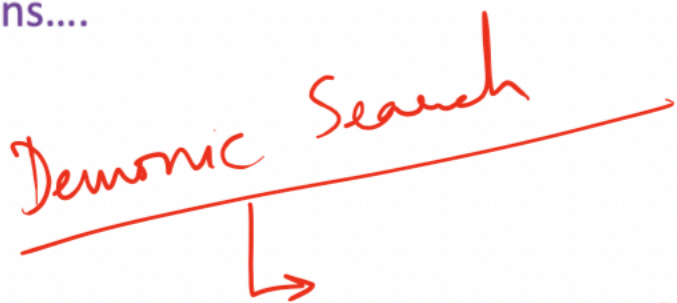


??

No guarantees of reaching the goal.

Further refinement.

An algorithm for Hierarchical Planning with approximate angelic descriptions....



collection of possible plans that need to be explored.

function ANGELIC-SEARCH(*problem*, *hierarchy*, *initialPlan*) **returns** solution or *fail*

  frontier ← a FIFO queue with *initialPlan* as the only element

  loop do

    if EMPTY?(*frontier*) **then return** *fail*

    plan ← POP(*frontier*) /* chooses the shallowest node in *frontier* */

    if REACH$^+$(*problem*.INITIAL-STATE, *plan*) intersects *problem*.GOAL **then**

      if *plan* is primitive **then return** *plan* /* REACH$^+$ is exact for primitive plans */

      guaranteed ← REACH$^-$(*problem*.INITIAL-STATE, *plan*) ∩ *problem*.GOAL

      if *guaranteed*≠{ } and MAKING-PROGRESS(*plan*, *initialPlan*) **then**

        finalState ← any element of *guaranteed*

        return DECOMPOSE(*hierarchy*, *problem*.INITIAL-STATE, *plan*, *finalState*)

      hla ← some HLA in *plan*

      prefix, *suffix* ← the action subsequences before and after *hla* in *plan*

      for each *sequence* **in** REFINEMENTS(*hla*, *outcome*, *hierarchy*) **do**

        frontier ← INSERT(APPEND(*prefix*, *sequence*, *suffix*), *frontier*)

BFS

can have HLA

Commit

Explore further

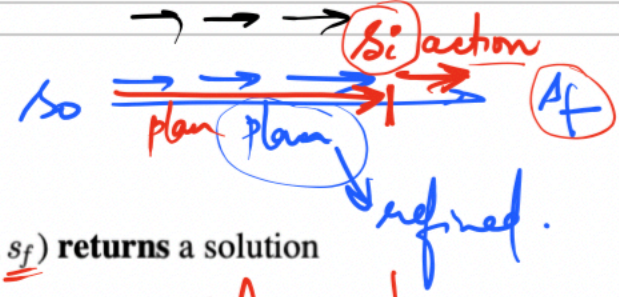
intersection

prefix HLA suffix (always primitive)

Backward Search.

function DECOMPOSE(*hierarchy*, s_0, *plan*, s_f) **returns** a solution

Backward Search.



function DECOMPOSE($hierarchy, s_0, plan, s_f$) **returns** a solution

solution ← an empty plan
while *plan* is not empty **do**
  action ← REMOVE-LAST(*plan*)
  s_i ← a state in REACH$^-$(s_0, *plan*) such that $s_f \in$ REACH$^-$(s_i, *action*)
  problem ← a problem with INITIAL-STATE = s_i and GOAL = s_f
  solution ← APPEND(ANGELIC-SEARCH(*problem*, *hierarchy*, *action*), *solution*)
  $s_f \leftarrow s_i$
return *solution*

can have HLAs

HLA solution ← prefix + solution

Sensorless Agent

Outcomes of actions are uncertain

Planning & Acting in Non-deterministic Domains

Challenges of real world environments

- Environments may be
  - Non-observable
  - Partially observable and non-deterministic
  - Unknown
- We need special planning formulations
  - Sensorless planning (conformant planning)
  - Contingency planning
  - Online planning and replanning

State Description
Action Schema.
if — then —

The Painting Problem

- Initial conditions and Goal:

$$Init(Object(Table) \land Object(Chair) \land Can(C_1) \land Can(C_2) \land InView(Table))$$
$$Goal(Color(Chair, c) \land Color(Table, c))$$

variable

- Available Actions:

$$Action(RemoveLid(can),$$
$$\text{PRECOND: } Can(can)$$
$$\text{EFFECT: } Open(can))$$
$$Action(Paint(x, can),$$
$$\text{PRECOND: } Object(x) \land Can(can) \land Color(can, c) \land Open(can)$$
$$\text{EFFECT: } Color(x, c))$$

c : can take any value.

Percept Schema

- Percepts are supplied by the sensors.
- For planning we need a model of the sensors, specified as a percept schema

$$Percept(Color(x, c),$$
$$\text{PRECOND:}\ Object(x) \wedge InView(x)$$
$$Percept(Color(can, c),$$
$$\text{PRECOND:}\ Can(can) \wedge InView(can) \wedge Open(can)$$

Color(Table, white)

- The agent also uses a *LookAt* action that causes objects to come into view

$$Action(LookAt(x),$$
$$\text{PRECOND:}\ InView(y) \wedge (x \neq y)$$
$$\text{EFFECT:}\ InView(x) \wedge \neg InView(y))$$

- A sensorless environment has no Percept axioms.
  - A sensorless agent can solve the painting problem.
- A fully observable environment would have Percept axioms with no preconditions
- If the agent has sensors, it can come up with a better (cost/time) contingent plan.
- An online planning agent doesn't need to plan in advance for any unexpected situation.

Sensorless Planning

- The agent cannot sense the environment.

  So it will maintain a belief state about the environment.

- The initial belief state has some unchanging fluent (common to all states)

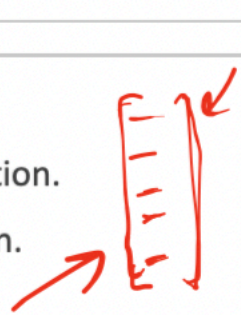
- $$Object(Table) \wedge Object(Chair) \wedge Can(C_1) \wedge Can(C_2)$$

- We add another fact to the initial belief that objects and cans have colours.

$$b_0 = Color(x, C(x))$$

Sensorless Planning

- To represent the belief state, we make the open-world assumption.
  That is, if a fluent doesn't appear in a state, its value is unknown.



- We can compactly represent the belief state as a 1-CNF formula comprising fluent that are known.
  We simply exclude the unknown fluents.

F1 ∧ F2 ∧ F3

- Sensorless planning is formulated as a search problem.
  We should know how to update a belief state for a given action sequence.

Updating the belief state

- The general formula

$$b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ and } s \in b\}$$

- We must consider what happens to each literal l in each physical state s in b when action a is applied. We consult the add/delete lists available with the action schema.
  - If the action adds l, then l will be true in b' regardless of its initial value.
  - If the action deletes l, then l will be false in b' regardless of its initial value.
  - If the action does not affect l, then l will retain its initial value.

    If the initial value is unknown then it will not appear in b'

Updating the belief state

- We calculate b' as:

$$b' = \text{RESULT}(b, a) = (b - \text{DEL}(a)) \cup \text{ADD}(a)$$

- Instead of using the set semantics, we simply make any atom which appears in $DEL(a)$ to false and set any atom that appears in $ADD(a)$ to true.

- For example, applying action $RemoveLid(Can_1)$ to the initial belief $b_0 = Color(x, C(x))$ gives

$$b_1 = Color(x, C(x)) \wedge Open(Can_1)$$

- Applying the action $Paint(Chair, Can_1)$

$$b_2 = Color(x, C(x)) \wedge Open(Can_1) \wedge Color(Chair, C(Can_1))$$

- Finally applying the action $Paint(Table, Can_1)$

$$b_3 = Color(x, C(x)) \wedge Open(Can_1) \wedge Color(Chair, C(Can_1)) \wedge Color(Table, C(Can_1)) .$$

- The final belief state satisfies the goal $Color(Table, c) \wedge Color(Chair, c)$ with the binding $\{c, C(Can_1)\}$

1-CNF property of the belief state

- Fortunately, the belief state continues to be (compactly) representable using 1-CNF form when updates are defined by PDDL action schemas.
- However, the action schemas must have the same effects for all states in which their preconditions are satisfied.
- Else, the 1-CNF property is lost.
  Action schemas need to include **conditional effect**.

$$Action(Suck,$$
$$\text{EFFECT: } \textbf{when } AtL\text{: } CleanL \wedge \textbf{when } AtR\text{: } CleanR)$$

Handling unmanageably wiggly belief states

- The resulting belief state after applying the *Suck* action is

$$(AtL \wedge CleanL) \vee (AtR \wedge CleanR)$$

- One way to manage the wiggly belief states is to use their 1-CNF approximation.

  Such an approach is sound, but incomplete.

  OR

  We can look for action sequences that keep the belief state as simple as possible.

1-CNF

↓ avoid CE.

↓

1-CNF

Action sequence to maintain 1-CNF belief

- For example, in the senseless vacuum world, the action sequence [Right, Suck, Left, Suck] generates the following sequence of belief states:

$$
\begin{aligned}
b_0 &= True \\
b_1 &= AtR \\
b_2 &= AtR \wedge CleanR \\
b_3 &= AtL \wedge CleanR \\
b_4 &= AtL \wedge CleanR \wedge CleanL
\end{aligned}
$$

1-CNF

Handling unmanageably wiggly belief states

- We represent a belief state as just the initial belief b_0 and the action sequence

  i.e. b_0 then $[a_1, \ldots, a_m]$? dont bother about representing

- However, determining whether the goal is satisfied, or an action is applicable, may require a lot of computation.

  We need to perform the entailment test $b_0 \wedge A_m \models G_m$ — Goal.

  where A_m are the successor state axioms required to define the occurrences of the actions $[a_1, \ldots, a_m]$

- We can also use heuristic functions to guide the state space search.
- Solving any subset of a belief state is necessarily easier than solving the belief state:

$$\text{if } b_1 \subseteq b_2 \text{ then } h^*(b_1) \leq h^*(b_2)$$

- That is, any admissible heuristic computed for a subset is admissible for the belief state itself. For example,

$$H(b) = \max\{h(s_1), \ldots, h(s_N)\}$$

b.

$s_1 \rightarrow h(s_1)$

$s_2 \rightarrow h(s_2)$

s_1

s_N

Sensorless Planning
Contingent Planning ← percepts.
Online Replanning

Contingent Planning

Percepts

- In environments with partial observability or non-determinism, the plans can depict conditional branching based on percepts.

$[LookAt(Table), LookAt(Chair),$
$\quad \textbf{if } Color(Table, c) \wedge Color(Chair, c) \textbf{ then } NoOp$ ← Do nothing
$\quad\quad \textbf{else } [RemoveLid(Can_1), LookAt(Can_1), RemoveLid(Can_2), LookAt(Can_2),$
$\quad\quad\quad \textbf{if } Color(Table, c) \wedge Color(can, c) \textbf{ then } Paint(Chair, can)$
$\quad\quad\quad \textbf{else if } Color(Chair, c) \wedge Color(can, c) \textbf{ then } Paint(Table, can)$
$\quad\quad\quad \textbf{else } [Paint(Chair, Can_1), Paint(Table, Can_1)]]]$

- Variables are considered as existentially quantified.

$\exists x .$

Involvement of Percept axioms

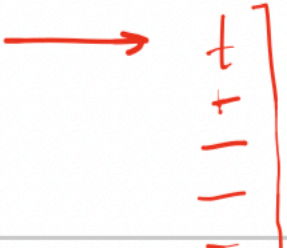
- As we take actions, percept literals $p_1, p_2, \ldots p_k$ are received.
- A percept can have percept axioms associated with it

  Percept (p, PRECOND:c)

- The resulting belief state will contain the percept p as well as the conjunction of literals c that define the preconditions for the percept.

Online Replanning

- Some branches of a partially constructed contingent plan can simply say *Replan*.
- An agent may also keep monitoring its execution to determine the need for a new plan.
- If the agent does not have the ability to replan then the contingent plan needs to take care of every possible situations that can occur.

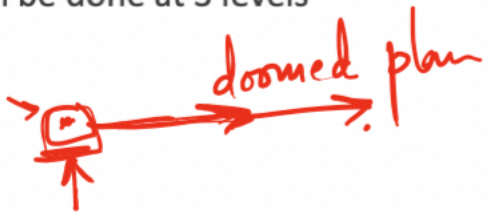


- Replanning may also be needed if the agent's model of the world is incorrect.
  - Missing precondition
  - Missing effect
  - Missing state variable
  - Exogenous events

Levels of execution monitoring

- Monitoring of the execution/environment can be done at 3 levels
  - Action monitoring
  - Plan monitoring
  - Goal monitoring

Preconditions (Remaining plan)

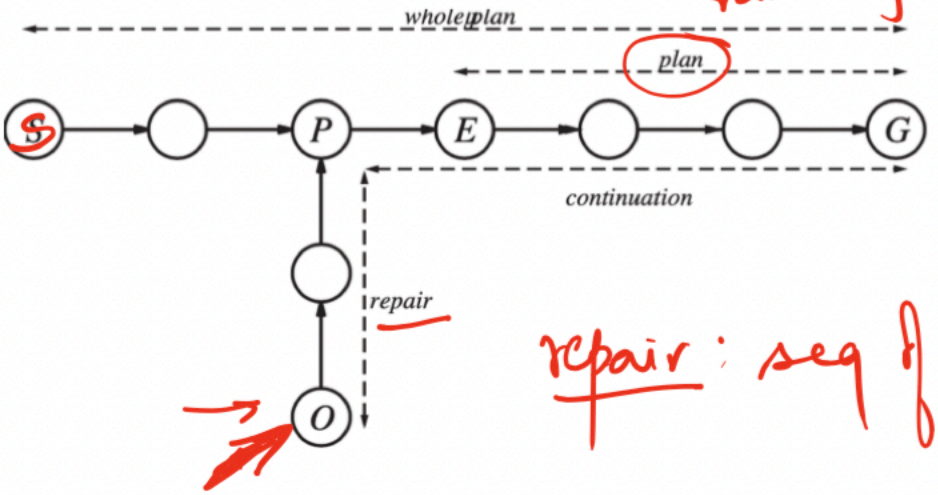


better goals

Action Monitoring

whole plan

remaining plan.



repair : seq of actions

The Painting problem

$[LookAt(Table), LookAt(Chair),$
$\quad \textbf{if } Color(Table, c) \wedge Color(Chair, c) \textbf{ then } NoOp$
$\quad\quad \textbf{else } [RemoveLid(Can_1), LookAt(Can_1),$
$\quad\quad\quad \textbf{if } Color(Table, c) \wedge Color(Can_1, c) \textbf{ then } Paint(Chair, Can_1)$
$\quad\quad\quad \textbf{else } \textsc{Replan}]] .$

Replan
- repair
- Continuation