PLANNING

Finding a SEQUENCE of ACTIONS to achieve a GOAL.

partially or totally ordered

We need to represent INITIAL STATE, GOAL, ACTIONS.

Actions are formally defined through:
- PRECONDITION: Properties that must hold to execute the action
- POSTCONDITION: Effects of the actions in the environment

and works with parameters.

A planner is COMPLETE if it always finds a plan, when one exists, its CORRECT when the solution found leads to a goal state.

Executing a plan yields IRREVERSIBLE and NON DETERMINISTIC RESULTS as in the real world we can't backtrack nor we can expect an always reliable outcome in every possible environment condition.

We will from now on consider (OFFLINE) GENERATIVE PLANNERS which works on a complete description of the problem (SNAPSHOT) to decide which actions to do and when.

PLAN AS A SEARCH

LINEAR PLANNING

Planning problem is formulated as a search in the state space on uses

- FORWARD SEARCH → From initial state to the goal proceed until state is superset of goal
- BACWARD SEARCH → Proceed from goal until a state is superset of initial state.

Each state are a collection of properties but also properties that doesn't interfer could be there.

With BACWARD SEARCH we use REGRESSION to infer the actions needed to build a state, each actions adds or remove some conditions depending on if the conditions are on the add or delete actions.

DEDUCTIVE PLANNING

I order logic to represent goals, states and actions.

SITUATION: snapshot describing properties that hold in a given state

ACTIONS: which fluents (properties) are true as a sequence of an action

- GREEN DEFINITION: from the solution build the pbm using theorem proving.

  FRAME PROBLEM: we need to know which properties change (either TRUE or FALSE)

  in complex domains states becomes huge.

  To find a solution we start negating the goal and go on until a contraddiction (empty clause) is found.

  For each axiom that brings us to another state we insert FRAME AXIOMS which maintain previous state properties.

- KOWALSKY DEFINITION

  We use holds predicate to describe all TRUE relations in a state

  Poss to indicate if a state is reachable

  Pact to indicate if a state is applicable

  can be executed

  => $Poss(S) \wedge Pact(a, S) \rightarrow Poss(do(a, S))$

FRAME PROBLEM is not present as we holds clauses are not touched.

STRIPS

Ad-hoc language and algorithms for efficency

STATE / GOAL contains only fluents that are TRUE everything else is FALSE

ACTIONS are repr by 3 lists:

- PRECONDITIONS: required fluents
- DELETE LIST: fluents false after action
- ADD LIST: fluents true after action

sometimes together as EFFECTS
(positive/negative axioms for ADD/DELETE)
everything not specified is unchanged

LINEAR PLANNER w\ CLOSED WORLD ASSUMPTION based on
↓
initially we know everything

BACWARD SEARCH.

We have GOAL STACK and CURRENT STATE we search until head
↓ ↓
BACKWARD SEARCH FORWARD SEARCH

of GOAL STACK = CURRENT STATE.

Iteratively we find an action that produces as POSTCONDITION the head of the goal stack an push its PRECONDITIONS in the GOAL STACK.
↓
We update CURRENT STATE as soon as the head of GOAL STACK can be executed

When very large search spaces are found we make use of HEURISTICS to select GOALS to analyze and/or actions to SATISFY a GOAL.

We could reach a case where $G_1, G_2 \subseteq$ GOAL and by satisfying G_2 we obstroy the work done by G_1
↓
We solve G_1 and G_2 indipendently and if at the end the conjunction is not true we change the solving ordering
↓
Therefore when we unpack an action and insert its PRECONDITIONS in the GOAL LIST we push first of all the action itself, to execute it and the we push the conjunction of the preconditions before pushing the preconditions. Doing so makes us sure that satisfying a subgoal hasn't made any interference with the action.

PLAN AS A SEARCH IN THE SPACE OF PLANS

The search space is defined as the set of plans where we search for a correct one.

↓

each node is a partial plan on actions obey refinement operations

This kind of planners are called **NON LINEAR PLANNERS** and works under **CLOSED WORLD ASSUMPTIONS** and **LEAST COMMITMENT**

↓

never impose an action if it's not strictly necessary

The initial plan is an empty plan w\ fake **START** and **END** actions

↓ preconditions are initial state

↓ postconditions are goal

A solution is a set of partially specified and ordered operators that will be linearized at the end to obtain a fully specified set of operators.

To keep track of the states relations **CAUSAL LINKS** are deployed.

↓

a triple in the form

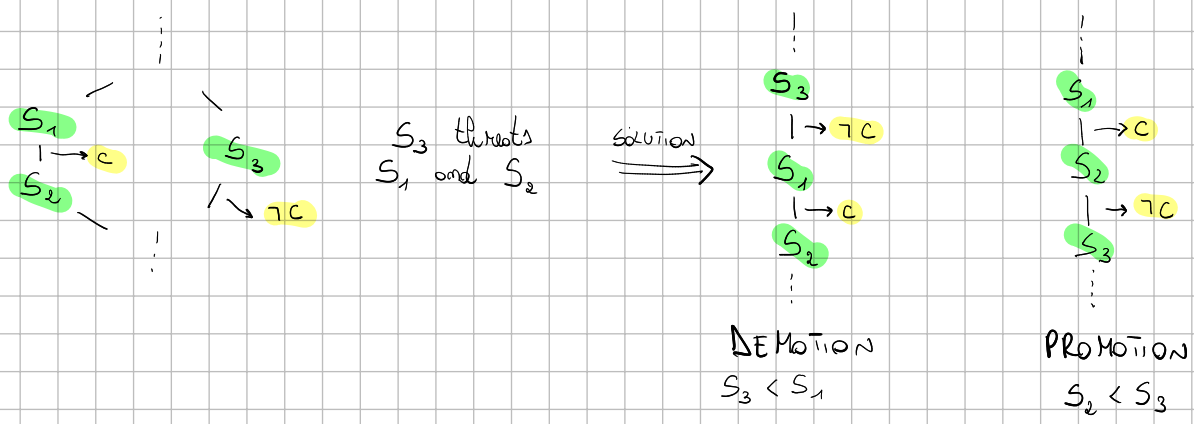
$< S_i, S_j, c >$

- S_i: from state
- S_j: to state
- c: causality (action) which relates S_i and S_j; c should be in S_i post conditions and in S_j precondition

It could happen that an action **THREATEN** another action

↓

S_3 is a threat for $< S_1, S_2, c >$ if it has an effect that negates c and no ordering prevents S_3 to be executed between S_1 and S_2.



PoP (Partial order planning algorithm)

Basically

1. INIT EMPTY PLAN
2. ADD ACTION TO REFINE PLAN
3. RESOLVE THREATS TROUGH EITHER PROMOTION OR DEMOTION

repeat until plan is not a solution

MTC (Modal truth criterion)

Promotion and demotion are not enough to ensure completness.

MTC implements a more detailed threat protection phase instead of 1 phase used by PoP.

↓

mainly trough the use of WHITE KNIGHTS : insert an action that restabilishes the condition being threatened.

HIERARCHICAL PLANNING

More efficient in complex domains.

Planning is done at different levels of abstraction by solving more difficult preconditions first and then refining w\ details.

↓

Define macro actions and refinement atomical actions

ABSTRIPS

Build a plan by satisfying first more difficult preconditions.

↓

defined by a THRESHOLD

gradually lower the threshold until plan is correct.

Macro-actions can also make use of precompiled actions where the correct order is already enforced. Trough action DECOMPOSITION plans are built more efficiently.

↓

only possible if through it we don't threat any part of the plan already built.

Some planners works in **OPEN WORLD ASSUMPTION**

↓

Some uncertainty could be present in the environment

SENSING ACTIONS are used to retrieve unknown informations.

CONDITIONAL PLANNING

Generate a plan for each source of uncertainty in the plan.

Combinatorial explosion happens when there are a lot of sources of uncertainty

⇒ use probabilistic approach planner instead that plan only for most probable contexts

REACTIVE PLANNERS

Choose actions based on environment constraints.

HYBRID SYSTEMS : Mix generative and reactive approaches.

GRAPH PLAN

Crate a **PLANNING GRAPH** while planning inserting also the time dimension.

↓

one of the most efficient **CORRECT** and **COMPLETE** algorithms

Works under CLOSED WORLD ASSUMPTION, returning the SHORTEST PATH or an INCONSISTENCY.

States are SETS of PREDICATES true in a given state.

The PLANNING GRAPH is a DIRECTED LEVELED GRAPH where nodes belongs to different levels and arcs connects adjacent levels.

connecting arcs can be:

PRECONDITION ARCS	ADD ARCS	DELETE ARCS
proposition → action	action → proposition	action → proposition

Each action level contains: applicable action, constraints connecting mutex actions
" proposition " " : literals built from the previous action level combinations

A level corresponds to a time-step.

Inconsistencies could emerge when building a graph plan:

- 2 actions inconsistent at the some timestep
- 2 propositions " " " " "

In both cases they CAN'T APPEAR TOGETHER ON PLAN but MAY APPEAR AT SAME LEVEL

Actions could be inconsistent because have: INCONSISTENT EFFECT, INTERFERENCE,

↓ one action negates the effect other

↓ action delete the other precondition

mutex preconditions

↑

COMPETING NEEDS

Similar goes for PROPERTIES.

PLANNING GRAPH CREATION:

① All initially TRUE propositions → INITIAL STATE

② ACTION LEVEL CREATION:
- add action if 2 propositions aren't mutex and can be unified by using PRECONDITIONS
- add a NO-OP for each proposition
  ↓
  FRAME PROBLEM
- mark interferring actions as MUTEX

until goal in proposition

③ PROPOSITION LEVEL CREATION:
- add prop. in ADD and DELETE list of actions and no-op
- mark as mutex proposition such that all actions building the 2 are mutex

A valid plan needs to be extracted from the GRAPH PLAN
↓
actions in some ts don't interfer
propositions " " " aren't mutex
last time step contains goal or goal's literals don't interfer

Used to prune paths
- if ∃ valid plan => valid plan ⊆ GRAPH
- 2 actions are mutex at a ts if ∄ a valid plan containg both
- " propositions " " " " " " one of them denies the other.

MEMOIZATION → if at some ts a subset of goal is not satisfiable it's remembered in the future

FAST FORWARD : efficient planner that uses heuristics

CONSTRAINT SATISFACTION

Find a state that meets a given set of constraints.

A CONSTRAINT SATISFACTION PROBLEM (CSP) is defined on a finite set of variables:

1. decisions that have to be taken
2. domain of possible values
3. constraints on 1 and 2

↓

$C(x_1, \dots, x_K)$ in K variable

IN

$D_1 \times \dots D_K$

specify which values of the variables are compatible each other.

a solution to a CSP is simply the variable assignment we can do.

CSP can be solved through STATE SEARCH where each level correspond to a variable and each node to its assignment

↓

max depth = n (number of variables)
but leaf number is d^n
↓
domain size

Tree size can be pruned through PROPAGATION ALGORITHM where subtree that leads to a failure are removed.

We can in fact solve a CSP trough:

- PROPAGATION ALGORITHM
- CONSISTENCY TECHNIQUES

Applying the tree search as a depth first search at each level we either:

find a solution | discover a failure | assign a new variable

we should decide however how to:

- order the variables
- order the values to assign

} heuristics

- propagate the assign → depend on strategies.

PROPAGATION STRATEGIES

GENERATE AND TEST

Assign randomly the variables and subsequently check if the constraints are satisfied

Use an a-posteriory technique (constraints are used AFTER assign)

↓

the possible permutations grow as the factorial of the items to permute.

STANDARD BACKTRACKING

Better than GEN & TEST however still works a-posteriori.

Each time we assign a variable constraints on previous variable are checked.

↓

assign randomly and check if should undo.

FORWARD CHECK

After each instantiation PROPAGATE ALL THE CONSTRAINTS, moving those who depends on the assigned variable in context.

Suppose

x_1 x_2 x_3 x_4

already assigned → FC checks constraints on x_3 and x_4 using the assigned value x_2 and removes all of those values that can't be later used.

If the domain becomes empty we have a failure.

LOOKAHEAD ALGORITHMS (PARTIAL and FULL)

Perform also a forward check on non assigned variables so that we can rapidly know when the domain becomes empty

Partial lookahead only look forward, Full lookahead also backward:

$x_0 \overset{=0}{<} x_1 < x_2 < x_3$ → PLA $x_1 = [1, 2, \cancel{3}]$ cannot be used as valid (breaks constraint w\ x_3)

$x_2 = [1, 2, \cancel{3}]$

$x_3 = [1, 2, 3]$ → can be used here!

$x_1, x_2, x_3 \in [1,2,3]$

FLA $x_1 = [1, 2, \cancel{3}]$

$x_2 = [\cancel{1}, 2, \cancel{3}]$ would break constraint on x_1

$x_3 = [\cancel{1}, \cancel{2}, 3]$ would break constraint on x_1, x_2

Domain and variable order heuristics are divided in:

VARIABLE SELECTION HEURISTICS: which variable assign first

- first-fail → variables w/ smallest domain first
- most-constrained-first → variable in most constraints first (probably the most problematic to assign)

VALUE SELECTION HEURISTICS: which value assign first

- most likely to succeed values (least constraining effect)

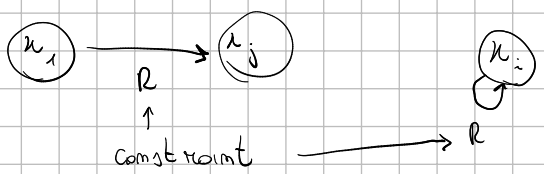
Heuristic can be STATIC (determine order initially and stay fixed) or DYNAMIC (determine order at each assignment)
↓
complex so usually a mix is used.

CONSISTENCY STRATEGIES

Instead of propagating the constraints to reduce domains we reduce the domain by eliminating values that cannot appear in the final solution.

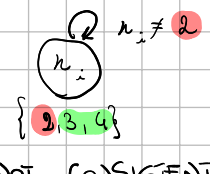
Consistency techniques base their use on graph-representation of the problem

Each CSP can be represented as a graph

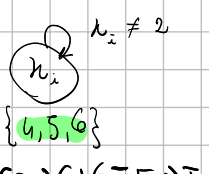


NODE CONSISTENCY - consistency of level 1

A node x_i is consistent if for each value $x_k \in$ DOMAIN the unary constraint R



NOT CONSISTENT



CONSISTENT

A graph is node-consistent if all nodes are consistent.

ARC CONSISTENCY - consistency of level-2

An arc $a(i,j)$ is consistent if $\forall x_i \in D_i$ exist at least one value y_j such that the constraint between i and j are satisfied.

↓
SUPPORT

A graph is arc consistent if every arc is arc consistent.

↓
obtained iteratively until the network is QUIESCENT

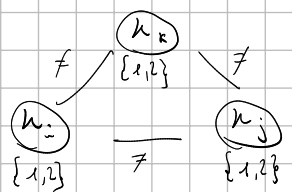
↓
state or condition called mutte by modifying a domain.

Arc consistency is highly complex → FLA performs a partial arc-consistency in a less complex way.

Arc-consistency is usually applied:

- BEFORE SEARCH → produce a simplified graph
- AS PROPAGATION → Maintaining arc consistency (MAC) STEP

Sometimes AC is not enough



→ NO SOLUTION as $|Domain| < |Variables|$ but its AC!

PATH-CONSISTENCY - level 3 consistency

A path is PATH-CONSISTENT if $\forall x \in D_i, y \in D_j$ in the path (i,j,k) $\exists z \in D_k$ that satisfyies $P(i,k)$ and $P(k,j)$

for a CSP of n variables we need K-CONSISTENCY with $K = n$

if we have $K < n$ (domain big enough to have a different value for each path) we can solve without search, otherwise we need to search.