Reasoning and Planning
Unit 5. Temporal Reasoning and Planning

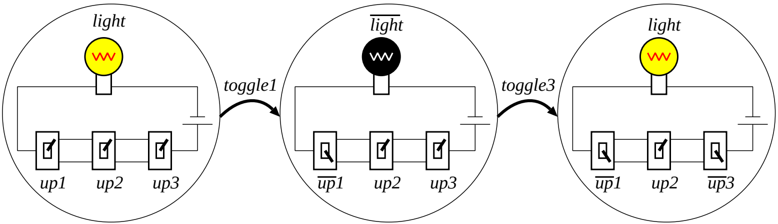
Pedro Cabalar

Dept. Computer Science
University of Corunna, SPAIN

November 18, 2022

Back to our simple example

- Lamp and switches revisited
- Fluents: *up*1, *up*2, *up*3, *light* (Boolean).
- Actions: *toggle*1, *toggle*2, *toggle*3.
- State: a possible configuration of fluent values. Example: $\{\overline{up1}, up2, up3, \overline{light}\}$.
- Situation: a moment in time. We can just use $0, 1, 2, \ldots$



Reasoning about actions with ASP

- Download system `telingo` (temporal `clingo`)
- We can make groups of rules

```
#program initial. % At timepoint t=0
...
#program dynamic. % Transition from t-1 to t
...
#program always. % Any timepoint t=0..n-1
...
#program final. % Last timepoint t=n-1
...
```

- Predicate names preceded by ' refer to timepoint `t-1`
- Predicate names preceded by _ refer to timepoint `t=0`
- Temporal formulas built with `&tel{ ... }`

Reasoning about actions with ASP

```
% File: switches.lp (domain description)
switch(1..3).
action(tog(X)) :- switch(X).

#program dynamic.
% Effect axioms
  h(sw(X),up)    :- 'h(sw(X),down), o(tog(X)).
  h(sw(X),down)  :- 'h(sw(X),up),   o(tog(X)).
  h(light,off)   :- 'h(light,on),   o(tog(_)).
  h(light,on)    :- 'h(light,off),  o(tog(_)).
% Executability constraints: none in this case
% Inertia: c(F)= fluent F has changed
  h(F,V) :- 'h(F,V), not c(F).
  c(F)   :- 'h(F,V), h(F,W), V!=W.

% Action generation
  1 { o(A): _action(A) } 1.
```

RAC goals

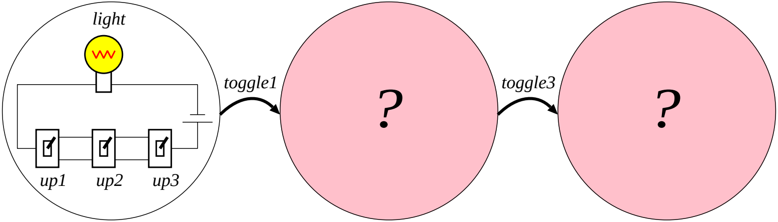
We want to solve some typical reasoning problems.

The most usual ones:

- Simulation (aka prediction, aka temporal projection):
  run a sequence of actions on an initial state
- Temporal explanation (aka postdiction):
  fill gaps from partial observations
- Planning: obtain sequence of actions to reach some goal
- Diagnosis: explain unexpected observed results
- Verification: check system properties

Prediction (simulation, or temporal projection)

- **Knowing**: initial state + sequence of actions
- **Find out**: final state (alternatively sequence of intermediate states)



Reasoning about actions with ASP

Prediction example

```
% File: switches-predict.lp (instance of prediction problem)
#program initial.
h(light,off).
h(sw(X),up) :- switch(X).
```

We assert a sequence of facts using:

```
% Sequence of performed actions
&tel{
    &true
 ;> o(tog(3))
 ;> o(tog(1))
 ;> o(tog(2))
 ;> o(tog(2))
}.
#show h/2.
#show o/1.
```

where `;>` is a sequence operator

Reasoning about actions with ASP

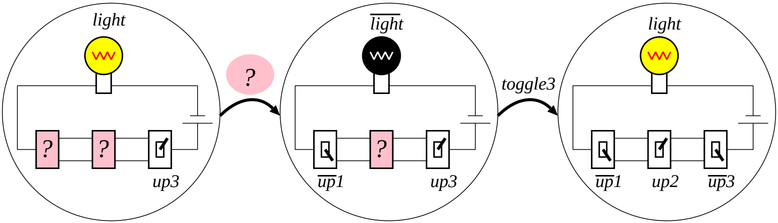
Prediction example

Calling `telingo switches.txt switches-predict.txt`

```
Answer: 1
 State 0:
  h(light,off) h(sw(1),up) h(sw(2),up) h(sw(3),up)
 State 1:
  o(tog(3))
  h(light,on) h(sw(1),up) h(sw(2),up) h(sw(3),down)
 State 2:
  o(tog(1))
  h(light,off) h(sw(1),down) h(sw(2),up) h(sw(3),down)
 State 3:
  o(tog(2))
  h(light,on) h(sw(1),down) h(sw(2),down) h(sw(3),down)
 State 4:
  o(tog(2))
  h(light,off) h(sw(1),down) h(sw(2),up) h(sw(3),down)
```

Postdiction (or temporal explanation)

- **Knowing**: partial observations of states and performed actions
- **Find out**: complete information on states and performed actions



Reasoning about actions with ASP

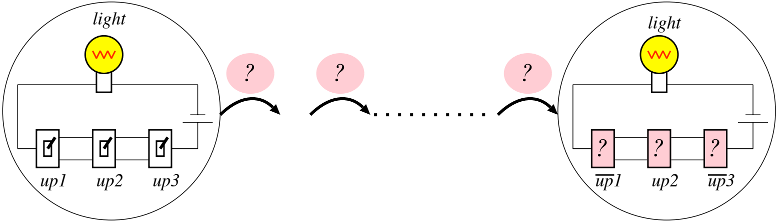
Postdiction example:

```
% switches-postdict.lp
#program initial.
% Completing unknown facts
 1 {h(sw(X),up); h(sw(X),down)} 1 :- switch(X).
 1 {h(light,on); h(light,off)} 1.

% Observations: we use a constraint!
 :- not &tel{
      h(sw(3),up) & h(light,on)
   ;> h(light,off) & h(sw(1),down) & h(sw(3),up)
   ;> o(tog(3))
 }.
```

Calling `telingo 0 switches.txt switches-postdict.txt` we get 4 possible explanations

Planning

- **Knowing**: initial state + goal (partial description of final state)
- **Find out**: plan (sequence of actions) that guarantees reaching the goal



Reasoning about actions with ASP

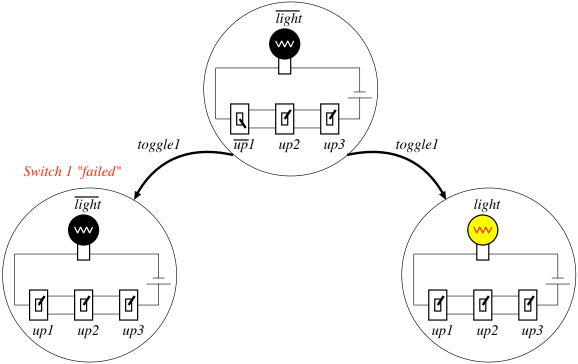
Planning example

```
% File: switches-plan.lp
#program initial.
h(light,on).
h(sw(X),up) :- switch(X).

#program final.
goal :- h(light,on),h(sw(1),down),
        h(sw(2),up),h(sw(3),down).
:- not goal.
```

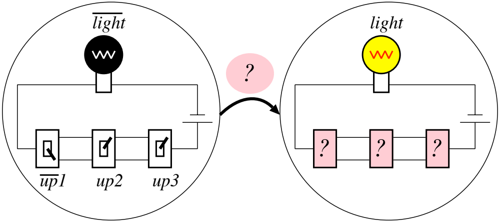
Calling `telingo 0 switches.txt switches-plan.txt` we get two minimal plans of length 2 toggling 1 and 3 or vice versa.

Planning vs Postdiction

- Note that planning seems a type of postdiction. For deterministic systems, this is true, but ...
- Nondeterministic transition system: fixing current state + performed action $\longrightarrow$ several possible successor states.
- For instance, switch 1 up may fail to turn the light on...



Planning vs Postdiction



- For postdiction, one valid explanation is: we performed *toggle*1, and it succeeded to turn the light on.
- For planning, *toggle*1 is not a valid plan: it does not guarantee reaching the goal *light*. Possible plans are *toggle*2 or *toggle*3.

Exercise

"Elaborating Missionaries and Cannibals Problem" [J. McCarthy]

> *3 missionaries and 3 cannibals come to a river and find a boat that holds two. If the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. How shall they cross?*

We will use the following fluents:

1. `n(G,B)` = is the number of persons of group `G` at bank `B`.

   Ex.: `h(n(mis,l),3)` = "*there are 3 missionaries in the left bank*"

2. `boat` points out the boat bank. Ex. `h(boat,l)` = "*the boat is at left bank*"

Exercise: missionaries and cannibals

We will use action:

- `move(M,C)` = move `M` missionaries and `C` cannibals.
- For simplicity, we include two action attributes `moved(mis,N)` and `moved(can,N)` that point out separately how many persons of each group are moved.

Exercise: missionaries and cannibals

We begin with types and initial state

```
#program initial.
% Some types
 group(mis;can).
 bank(l;r).
 opposite(l,r). opposite(r,l).
 action(move(M,C)) :- M=0..2, C=0..2, M+C<3, M+C>0.

% Initial state
 h(n(G,l),3)  :- group(G).
 h(n(G,r),0) :- group(G).
 h(boat,l).
```

Exercise: missionaries and cannibals

Rules for transitions

```
#program dynamic.
% Action generation
1 {o(A) : _action(A) } 1.

% Auxiliary (action attributes)
moved(mis,M) :- o(move(M,C)).
moved(can,C) :- o(move(M,C)).

% Executability axioms
:- moved(G,N), 'h(boat,B), 'h(n(G,B),M), N>M.

% Effect axioms (no inertia needed)
h(n(G,B),M+N) :- 'h(n(G,B),M), h(boat,B), moved(G,N).
h(n(G,B),M-N) :- 'h(n(G,B),M), 'h(boat,B), moved(G,N).
h(boat,B1)    :- 'h(boat,B), _opposite(B,B1).
```

Inertia not needed because all fluents are changed

Exercise: missionaries and cannibals

Rules for transitions

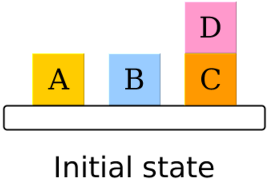
```
#program always.
% Missionaries not outnumbered by cannibals
:- h(n(mis,B),M), h(n(can,B),C), C>M, M>0.

#program final.
:- not goal.
goal :- h(n(mis,r),3), h(n(can,r),3).

#show o/1. % We only show performed actions
```

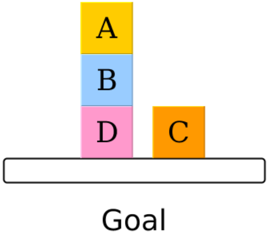
- We execute `telingo 0 mc.txt` and it will try length $t = 1, 2, \ldots$ until a solution is found.
- Four solutions of length $t = 11$ are eventually found.

Exercise: the Blocks World

Example

- Rearrange blocks of same size into goal stacks
- We can only move a free block (nothing on top) at a time
- We can put it on another block or on the table (it has room for all)





Exercise: the Blocks World

- Fluents:
  `h(on(B),L)` = block `B` is on location `L` (a block or the table)
- Actions:
  `o(move(B,L))` = move block `B` to location `L`
- To specify the goal we use a static predicate:
  `g(B,L)` = block `B` goal location is `L`

The problem instance:

```
blocks(a;b;c;d).
% Initial state
h(on(a),table). h(on(b),table). h(on(c),table). h(on(d),c).
% Goal positions
g(a,b). g(b,d). g(d,table). g(c,table).
```

Exercise: the Blocks World

A blocks world encoding:

```
#program initial.
location(table).   location(B) :- block(B).
#program dynamic.
h(on(B),L) :- o(move(B,L)).                 % effect axiom
:- o(move(B,_)), 'unclear(B).               % executability
:- o(move(_,L)), 'unclear(L).               % executability
:- o(move(B,table)), 'h(on(B),table).       % control constraint
#program always.
unclear(C) :- h(on(_),C),C!=table.
#program final.
:- _g(B,L), not h(on(B),L).                 % goal is reached
```

plus the general patterns:

```
#program dynamic.
1 {o(A): _action(A) } 1.         % action generation
h(F,V) :- 'h(F,V), not c(F).     % inertia
c(F)   :- h'(F,V),h(F,W),V!=W.   % change
#show o/1.
```

Exercise: the Blocks World

- An efficient encoding (goal oriented) may mean sacrifices in elaboration tolerance
- Strategy 1: restrict available actions
  ☞ Allow moving a block to the table or to its destination block

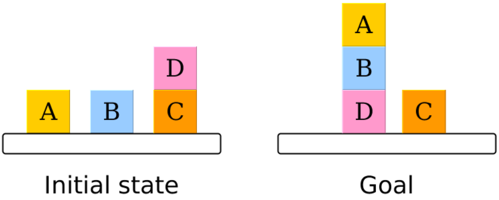
```
action(move(B,table)) :- block(B).
action(move(B,C))     :- g(B,C).
```

- Strategy 2: reduce generality of inertia. Replace by:

```
h(on(B),L) :- 'h(on(B),L), not o(move(B,_)).
```

  (Slight) frame problem (what if new actions for moving are defined)
- Strategy 3: control executability constraints = they tell you what (not) to do next, guided by our goal. Ex.: never undo a good tower.

Exercise: the Blocks World



Never undo a good tower:

- We should not start moving *A* on *B*, because *B* is not ready
- *B* will be ready when placed on *D*, being *D* ready in its turn
- *D* will be ready when placed on the table

Exercise: the Blocks World

- The `ready` auxiliary predicate is recursive

```
#program always.
ready(table).
ready(B) :- h(on(B),L), _g(B,L), ready(L).
```

- Finally, we can now add the control constraints:

```
#program dynamic.
% Don't move a ready block
:- o(move(B,_)), 'ready(B).
% Don't lay on a non-ready location
:- o(move(_,L)), not 'ready(L).
```

- These changes drastically reduce the search space, but the representation is now totally guided by goal location, predicate `_g(B,L)`.

Abductivon as best explanation

Abduction

- **Knowing**: a knowledge base KB + an observed result C
- **Find out**: hypotheses H such that $KB \cup H \models C$
  ☞ H should be the best explanation
- Example: we have $C = wetgrass$ and $KB =$

$$\begin{aligned} rain &\rightarrow wetgrass \\ sprinkle \wedge night &\rightarrow wetgrass \\ glass \wedge fill \wedge push &\rightarrow wetgrass \end{aligned}$$

  We can use $H_1 = \{rain\}$, ☜ simplest hypothesis
  $H_2 = \{sprinkle, night\}$ or $H_3 = \{glass, fill, push\}$
- If we have $KB' = KB \cup \{\neg rain\}$, the best hypothesis (less assumptions) becomes H_2

Abduction in ASP

- Atoms are reified: `h(A)` = atom `A` holds
- We distinguish the abducible atoms (they can form hypotheses)
  Generation of hypothesis becomes a choice rule

```
abducible(rain;sprinkle;night;push;glass;full).
{hyp(A)} :- abducible(A).        % generate hypothesis
h(A) :- hyp(A).                  % any hypothesis A holds
```

- Observations can be incorporated as constraints

```
h(wetgrass) :- h(rain).
h(wetgrass) :- h(night), h(sprinkle).
h(wetgrass) :- h(glass), h(full), h(push).
:- not h(wetgrass).                               % observation
```

  We cannot add `h(wetgrass)` as a fact, or as an abducible atom!

- We get 43 explanations! (including hypothesis with all abducible atoms). Smallest explanations = minimal sets of hypotheses

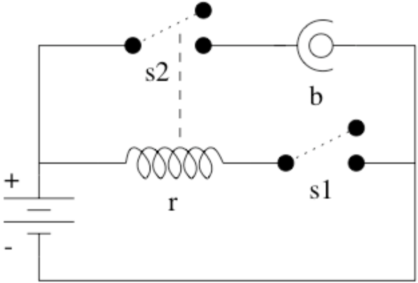
```
#minimize{1,A:hyp(A)}.
```

Diagnosis

- An agent acts in a dynamic environment and observes the results of her actions.
- Sometimes she gets discrepancies:
  observations $\neq$ expected result
- Diagnosis = search for abductive explanations
  - **Knowing**: a model distinguishing between normal and abnormal transitions + a partial set of observations (usually implying abnormal behavior).
  - **Find out**: the minimal set of abnormal transitions that explains the observations.

Diagnosis

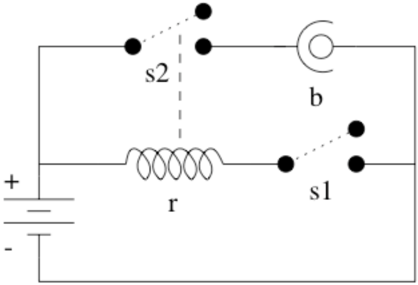
- Example [Balduccini & Gelfond 03]
  We have a circuit with lightbulb b and a relay r. The agent can close s1 causing s2 to close (if r is not damaged). The bulb emits light if s2 is closed and b is not damaged.



Diagnosis example

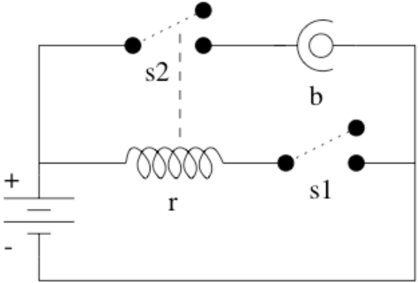
- Example [Balduccini & Gelfond 03]

  Exogenous action break damages the relay. Action power-surge damages r, and b too, if the latter is not protected (prot).



Diagnosis example

- Example [Balduccini & Gelfond 03]
  We close s1 but b does not emit light: what has happened?



Diagnosis example

- Types and domains

```
#program initial.
switch(s1;s2).
component(relay;bulb).
fluent(relay;light;b_prot).
fluent(S):-switch(S).
fluent(ab(C)) :- component(C).

value(relay,(on;off)).
value(light,(on;off)).
value(S,(open;closed)) :- switch(S).
% Fluents are boolean by default
domain(F,(true;false)) :- fluent(F), not value(F,_).
% otherwise, they take the specified values
domain(F,V) :- value(F,V).
```

- Fluents $ab(C)$ point out that a component is damaged

Diagnosis example

- Actions are exogenous *exog* or agent's *agent*:

```
agent(close(s1)).
exog(break;surge).
action(Y):-exog(Y).
action(Y):-agent(Y).
```

Diagnosis example

```
#program dynamic.
% Inertia
h(F,V) :- 'h(F,V), not c(F).
c(F)   :- 'h(F,V), h(F,W), V!=W.

% Direct effects
h(s1,closed) :- o(close(s1)).

#program always.
% Indirect effects
h(relay,on)   :- h(s1,closed), h(ab(relay),false).
h(relay,off)  :- h(s1,open).
h(relay,off)  :- h(ab(relay),true).

h(s2,closed)  :- h(relay,on).

h(light,on)   :- h(s2,closed), h(ab(bulb),false).
h(light,off)  :- h(s2,open).
h(light,off)  :- h(ab(bulb),true).
```

Diagnosis example

```
#program dynamic.
% Executability
:- o(close(S)), 'h(S,closed).

% Malfunctioning
h(ab(bulb),true)  :- o(break).
h(ab(relay),true) :- o(surge).
h(ab(bulb),true)  :- o(surge), not 'h(b_prot,true).
```

We use predicates obs_o and obs_h to denote observations

```
% Observed actions actually occur
o(A) :- obs_o(A).

#program always.
% Check that observations hold
:- obs_h(F,V), not h(F,V).

#program initial.
% Completing the initial state
1 {h(F,V):_domain(F,V)} 1 :- _fluent(F).
```

Diagnosis example

- These are the observations:

```
% A history
&tel {
     obs_h(s1,open) & obs_h(s2,open) &
     obs_h(b_prot,true) &
     obs_h(ab(bulb),false) &
     obs_h(ab(relay),false)

  ;> obs_o(close(s1)) &
     obs_h(light,off)
}.

#program dynamic.
% Generate exogenous actions
{ hyp(A): _exog(A) }.

o(A) :- hyp(A).
#show cause/1.
```

Diagnosis example

- This will provide all possible explanations, but not minimal diagnoses.

```
$ telingo 0 diag.lp
Answer: 1
 State 0:
 State 1:
  cause(break)
Answer: 2
 State 0:
 State 1:
  cause(break) cause(surge)
Answer: 3
 State 0:
 State 1:
  cause(surge)
SATISFIABLE
```

Diagnosis example

- We look for best explanations:

```
#minimize {1,A:hyp(A)}.
```

- To obtain all minimal solutions we use the options:

```
$ telingo --opt-mode=optN -n0 diag.lp
```

Two minimal solutions are found:

```
Answer: 1
 State 0:
 State 1:
  cause(surge)
Optimization: 1
Answer: 2
 State 0:
 State 1:
  cause(break)
Optimization: 1
OPTIMUM FOUND
```

Temporal Reasoning

- Until now, temporal expressivenes limited to:
  - program sections: `initial, dynamic, always, final`
  - previous situation `'h(sw(X),down)`
  - initial situation `_action(A)`
  - sequence of actions `;>`

  Can we go further?

- Example: (in the switches planning problem) choose plans where *tog*(1) does not occur after *tog*(3) Obvious solution: auxiliary predicate

```
#program dynamic.
moved3 :- o(tog(3)).
moved3 :- 'moved3.
:- o(tog(1)), moved3.
```

- Linear Temporal Logic can do the job requiring $\neg(\ o(tog(3)) \wedge\ \Diamond o(tog(1))\ )$

Linear-time Temporal Logic (LTL)

$\Box$ (forever), $\Diamond$ (eventually), $\circ$ (next), $\mathcal{U}$ (until)

- ✓ Decidable inference methods. Satisfiability: PSpace-complete
- ✓ Relation to other mathematical models: algebra, automata, formal languages
- ✓ Fragment of First-Order Logic: [Kamp 68] LTL = Monadic FO (<)
- ✓ Model checking and verification of reactive systems
- ✓ Many uses in AI: planning, ontologies, multi-agent systems, ...
- ✗ Monotonic: action domain representations manifest frame problem

Our proposal

Temporal Equilibrium Logic (TEL) [C_&Pérez 07]

TEL = ASP + LTL

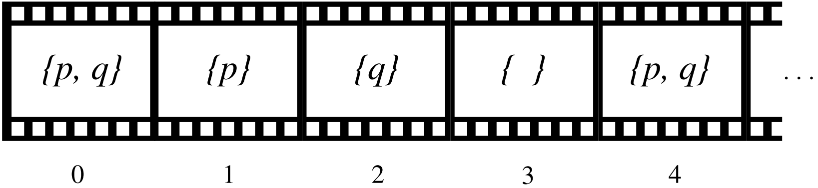
- ASP: logical characterisation Equilibrium Logic [Pearce 96]
- LTL: We add temporal operators $\square$, $\Diamond$, $\circ$, $\mathcal{U}$, $\mathcal{R}$ (+ past versions)
  Result: Temporal Stable Models for any arbitrary LTL theory.

(Linear) Temporal Equilibrium Logic

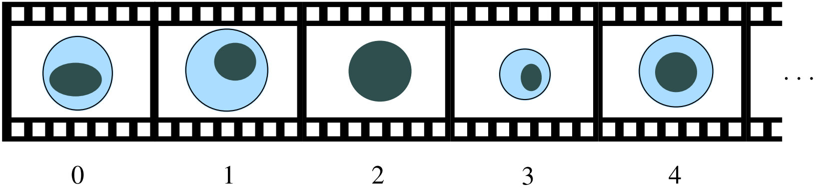
- Syntax = propositional plus
  - $\Box\alpha$ = "forever" α
  - $\Diamond\alpha$ = "eventually" α
  - $\circ\alpha$ = "next moment" α
  - $\alpha \ \mathcal{U} \ \beta$ = α "until eventually" β
  - $\alpha \ \mathcal{R} \ \beta$ = α "release" β
- As we had with Equilibrium Logic:
  1. A monotonic underlying logic: Temporal Here-and-There (THT)
  2. An ordering among models. Select minimal models.

Sequences

- In standard LTL, an interpretation is a (possibly ∞)-sequence of sets of atoms



- In THT we will have a (possibly ∞)-sequence of HT interpretations



Sequences

- We define an ordering among sequences $\mathbf{H} \leq {<}\mathbf{T}$ when

$$
\begin{array}{ccccccccccc}
T_0 & \longrightarrow & T_1 & \longrightarrow & T_2 & \longrightarrow & \ldots & \longrightarrow & T_i & \longrightarrow & \ldots \\
\cup\!| & & \cup\!| & & \cup\!|\!\cup & & & & \cup\!| & & \\
H_0 & \longrightarrow & H_1 & \longrightarrow & H_2 & \longrightarrow & \ldots & \longrightarrow & H_i & \longrightarrow & \ldots
\end{array}
$$

Definition (THT-interpretation)

is a pair of sequences of sets of atoms $\langle \mathbf{H}, \mathbf{T} \rangle$ with $\mathbf{H} \leq \mathbf{T}$. $\square$

Temporal Here-and-There (THT)

$\langle \mathbf{H}, \mathbf{T} \rangle, i \models \alpha \quad \Leftrightarrow \quad$ "α is proved at i"
$\langle \mathbf{T}, \mathbf{T} \rangle, i \models \alpha \quad \Leftrightarrow \quad$ "α assumed at i" $\quad \Leftrightarrow \quad \mathbf{T}, i \models \alpha$ in LTL

- An interpretation $M = \langle \mathbf{H}, \mathbf{T} \rangle$ satisfies α at situation i, written $M, i \models \alpha$

α	$M, i \models \alpha$ when ...
an atom p	$p \in H_i$
$\wedge, \vee$	as usual
$\alpha \rightarrow \beta$	$\mathbf{T}, i \models \alpha \rightarrow \beta$ in LTL and $\langle \mathbf{H}, \mathbf{T} \rangle, i \models \alpha$ implies $\langle \mathbf{H}, \mathbf{T} \rangle, i \models \beta$
$\circ, \Box, \Diamond, \mathcal{U}, \mathcal{R}$	as in LTL (just deal with timepoints)

(Linear) Temporal Equilibrium Logic

- $\circ\alpha$ satisfied in $i+1$

$$\bullet \longrightarrow \overset{\alpha}{\bullet} \longrightarrow \bullet \longrightarrow \ldots \longrightarrow \bullet \longrightarrow \ldots$$

- $\Box\alpha$ satisfied for all $j \geq i$

$$\overset{\alpha}{\bullet} \longrightarrow \overset{\alpha}{\bullet} \longrightarrow \overset{\alpha}{\bullet} \longrightarrow \ldots \longrightarrow \overset{\alpha}{\bullet} \longrightarrow \ldots$$

- $\Diamond\alpha$ satisfied for some $j \geq i$

$$\bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \ldots \longrightarrow \overset{\alpha}{\bullet} \longrightarrow \ldots$$

- $\alpha \ \mathcal{U} \ \beta$ = repeat α until (mandatorily) β

(Linear) Temporal Equilibrium Logic

$\alpha \ \mathcal{R} \ \beta$ = disjunction of two cases

- $\beta \ \mathcal{U} \ (\beta \wedge \alpha)$

$$\begin{array}{ccccccccccccc} \beta & & \beta & & \beta & & & & \beta \wedge \alpha & & & & \\ \bullet & \longrightarrow & \bullet & \longrightarrow & \bullet & \longrightarrow & \dots & \longrightarrow & \bullet & \longrightarrow & \bullet & \longrightarrow & \dots \end{array}$$

- $\Box \beta$

$$\begin{array}{ccccccccccccc} \beta & & \beta & & \beta & & & & \beta & & \beta & & \\ \bullet & \longrightarrow & \bullet & \longrightarrow & \bullet & \longrightarrow & \dots & \longrightarrow & \bullet & \longrightarrow & \bullet & \longrightarrow & \dots \end{array}$$

Temporal Equilibrium Models

Definition (Temporal Equilibrium Model)

of a theory Γ is a model $\mathbf{T}$ of Γ such that there is no $\mathbf{H} < \mathbf{T}$ satisfying $\langle \mathbf{H}, \mathbf{T} \rangle, 0 \models \Gamma$. □

- Temporal Equilibrium Logic (TEL) is the logic induced by temporal equilibrium models.

Theorem

Deciding whether a temporal theory has some THT-model is PSPACE*-complete.*

Theorem

Deciding whether a temporal theory has some temporal stable model is EXPSPACE*-complete.*

- Tool `abstem` allows computing temporal stable models for infinite traces
- Tool `telingo` focuses on finite traces, closer to practical problem solving with ASP
- Temporal formulas in telingo: we can use expressions inside `&tel{...}` with future-ops in heads, past-ops in bodies and any of them in constraints.

LTL	future	past
$\circ p$	`> p`	`< p`
$\hat{\circ} p$	`>: p`	`<: p`
$\Diamond p$	`>? p`	`<? p`
$\Box p$	`>* p`	`<* p`
$p\,\mathcal{U}\,q$	`p >? q`	`p <? q`
$p\,\mathcal{R}\,q$	`p >* q`	`p <* q`
$p \wedge \circ q$	`p ;> q`	`p <; q`

plus Boolean operators `&, |, ~, &true, &false` ...

- We can fix the trace length n with `&tel{n > &true}`

- Back to our planning example, we forbid
  $\Diamond(o(tog(3)) \wedge \circ\Diamond o(tog(1)))$

```
#program initial.
h(light,on).
h(sw(X),up) :- switch(X).

#program final.
goal :- h(light,on),h(sw(1),down),
        h(sw(2),up),h(sw(3),down).
:- not goal.

#program initial.
:- &tel{ >? (o(tog(3)) ;> >? o(tog(1)) )}.
```

Or we can use instead past operators like:

```
#program dynamic.
:- o(tog(1)), &tel{ < <? o(tog(3))}.
```

- Temporal control constraints: they allow disregarding plans without changing the domain representation towards a goal
- Convenient in concurrent planning: some “non-critical" agents may fill the plan with erratic actions
- Example of control constraints:

$\neg(\neg p \, \mathcal{U} \, d)$	if you pick (p), do it before dropping (d)
$\neg\Diamond(p \wedge \circ\Diamond p)$	never pick twice

```
:- &tel { ~p >? d }.
:- &tel { >? (p ;> >? p)}.
```

Classical AI Planning

- **Knowing**: initial state + goal (partial description of final state)
- **Find out**: plan (sequence of actions) to reach the goal

Classical AI Planning adds these premises

- Discrete: fluents, actions, time points, everything discrete
- Deterministic: given a state and a (ground) action, only one possible outcome
- Static: the environment does not change while the agent is deliberating
- Fully observable domain: no missing information

Languages for Planning

- Languages for planning look for a balance between allowing efficient processing versus flexibility (elaboration tolerance).

- The most influential language has been STRIPS
  STanford Research Institute Problem Solver [Fikes & Nilson 1971]

- Based on triples with $\langle \mathtt{ACTION}, \mathtt{PRECON}, \mathtt{EFFECT} \rangle$
  where PRECON and EFFECT are lists of literals

$$
\begin{array}{ll}
\mathtt{ACTION}: & move(X, From, To) \\
\mathtt{PRECON}: & on(X, From), clear(X), clear(To) \\
\mathtt{EFFECT}: & on(X, To), clear(From), \neg on(X, From), \neg clear(To)\ on(X,
\end{array}
$$

```
(:action move
  :parameters (?block ?from ?to)
  :precondition (and
     (on ?block ?from) (clear ?ob) (clear ?to) )
  :effect (and
```

Languages for Planning

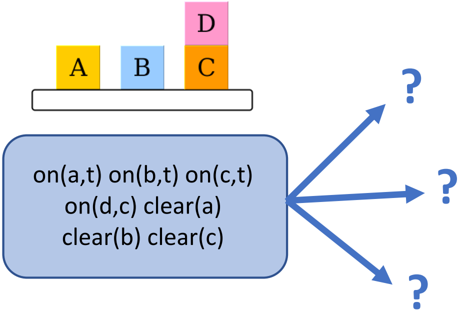
- Inertia is implicit: all the changes are listed (ADD/DEL lists)
- ⚠ STRIPS manifests ramification and qualification problems
- Existence of plan in propositional STRIPS is PSPACE-complete
- STRIPS has been carefully extended to add flexibility without harming planners efficiency ...

Languages for Planning

- PDDL (Planning Domain Description Language) [McDermott 1998]. Used for the International Planning Competition (IPC).
- Language versions:
  - 2.1: numeric fluents, plan metrics, actions with duration
  - 2.2: derived predicates (ramifications), timed exogenous events
  - 3.0: state-trajectory constraints (temporal logic), preferences
  - 3.0: object fluents (non-numeric multivalued)

Algorithms: Forward Planning

- State-space: 1 search node = 1 state
- Start with initial state, end when goal reached
- Expanding a node means looking for applicable ground actions



$move(X, From, To)$
throws 7 cases

$X = a, From = t, To = b$
$X = a, From = t, To = d$
$X = b, From = t, To = a$
$X = b, From = t, To = d$
$X = d, From = c, To = a$
$X = d, From = c, To = b$
$X = d, From = c, To = t$

- Branching factor = maximum size of one expand

Algorithms: Forward Planning

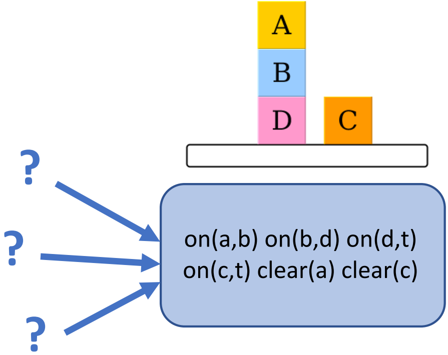
- Pros = simple!
  - We can use standard search algorithms
  - There are good (domain independent) heuristics
- Contras
  - Branching factor can be too large
- Many modern planners are based on Forward Planning
- A good admissible heuristic that underestimates the plan length is ignoring the delete list [Bonet & Geffner 2001]

Algorithms: Backward Planning

- Search space: 1 search node = 1 sub-goal = set of states
- We start with goal state, end when initial state reached
- Expanding a node means looking for relevant actions from effects to preconditions and jumping to a new sub-goal.
  E.g. where did each block come from?

$move(X, From, To)$
only 3 possible cases

$X = a, From = t, To = b$
$X = a, From = c, To = b$
$X = c, From = a, To = t$

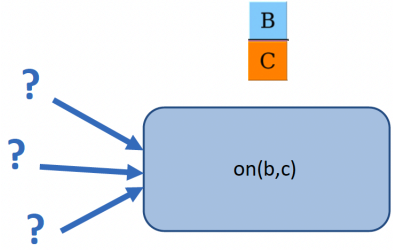


Algorithms: Backward Planning

- Pros
  - Goal-directed: explore relevant part of the search space
  - Branching factor much lower than Forward Planning
- Contras
  - Requires dealing with non-ground sub-goals
  - Hard to get good heuristics
- Goal can be a partial description. E.g. just get $on(b, c)$

$move(X, From, To)$
no hint to ground *From*

$X = b, From, To = c$



Algorithms: Bounded Horizon

- Horizon h = maximum plan length to explore
  ☞ Idea: find plans of fixed length h. If no one found, try with $h+1$
- First introduced in SAT planning [Kautz & Selman 92] with the `SATPLAN` planner.
- Ground fluent f becomes $h+1$ propositional atoms $f_0, \dots, f_h$
  Planning domain becomes a propositional formula in CNF
  A SAT solver is used to obtain plans
- CSP planning: domain becomes a constraint satisfaction problem (CSP). Actions and fluents can be integer variables
- ASP planning: domain becomes a logic program and an ASP solver is used instead. See translator from PDDL to ASP:
  ☞ `https://github.com/potassco/plasp`

Algorithms: Bounded Horizon

- In general, bounded horizon algorithms are incomplete: they cannot decide non-existence of plan
- However, in some cases, upper bounds for h can be obtained and completeness can be guaranteed.
- Example: if a Rubik Cube problem has a solution, the maximum number of quarter turns required is 26. Thus, try for all $h \leq 26$.

Other Planning Techniques

- GRAPHPLAN [Blum & Furst 1995] graph search based on a (layered) planning graph
  - Even layers: 1 node = 1 (ground) fluent fact
  - Odd layers: 1 node = 1 (ground) action
  - Edges of type: precondition, effect, mutex (mutual exclusion)
- Partial-order Planning avoids fixing an ordering among actions, when it is irrelevant. Example of plan:

$$\begin{tikzcd} & \mathit{get}(\mathit{flour}) \arrow[dr] & \\ \mathit{go}(\mathit{store}) \arrow[ur] \arrow[r] \arrow[dr] & \mathit{get}(\mathit{milk}) \arrow[r] & \mathit{pay} \\ & \mathit{get}(\mathit{eggs}) \arrow[ur] & \end{tikzcd}$$

any of the 3!=6 permutations for getting items is a valid plan

Other Planning Techniques

- Using Temporal Logic expressions of control knowledge.
  Introduced with TLPLAN [Bacchus & Kabanza 2000]
  Formulas in LTL like $\square(pick \rightarrow \Diamond drop)$ (as seen in `telingo`)

- Hierarchical Task Networks (HTN) planning
  Different levels: first high-level actions
  1. Land-travel from Ourense to Santiago (SCQ)
  2. Fly from SCQ to GCN (Arizona)
  3. Land-travel from GCN airport to Great Canyon

  Then, get a refinement
  Land-travel from Ourense to Santiago (SCQ) =
  1. Walk to Ourense train station
  2. Take train 04175 to Santiago
  3. Walk to bus station
  4. Take bus XG802 to SCQ

Other Planning Techniques

Combining Planning and Machine Learning

- Learning control rules using Inductive Logic Programming (ILP) [Leckie & Sukerman 1991] `Grasshopper`
- Learning macro actions, i.e. fixed sequences of actions that simplify the search. Example in 8-puzzle: push a row to the right Using Reinforcement learning [Randløv 1999]
- Learning the domain description from set of execution traces. Very recent example using ASP [Rodríguez, Bonet, Romero & Geffner 2021]

Beyond Classical Planning

- Conformant planning: domain is only partially observable
  - **Knowing**: partial initial state + goal description
  - **Find out**: plan (linear sequence of actions) that always reaches the goal
- Non-deterministic actions can also be covered: reduction to an exogenous variable unknown at the initial state
- Complexity raises from PSPACE to EXPSPACE

Beyond Classical Planning

- Contingency planning: domain is only partially observable, but we have sensing actions (always non-deterministic)
  - **Knowing**: partial initial state + goal description
  - **Find out**: plan = nested conditional sequences of actions that guarantee reaching the goal
- Example of plan: the phone is at the kitchen or at the bedroom

$$\begin{array}{l} \textit{go}(\textit{kitchen}); \textit{turn}(\textit{light}, \textit{on}); \textit{watch}; \\ \textbf{if } \textit{at}(\textit{phone}, \textit{kitchen}) \textbf{ then } \textit{walk}; \textit{pick}(\textit{phone}) \\ \textbf{else } \textit{go}(\textit{bedroom}); \textit{pick}(\textit{phone}) \end{array}$$

- We represent the agent's beliefs (epistemic reasoning)

Beyond Classical Planning

- Probabilistic Planning: extends non-deterministic actions with probabilistic information
  - Markov Decision Process (MDP): world is fully-observable, transition only depends on the previous state, not previous history
  - Partially Observable MDP (POMDP): world not fully observable, deal with agent's beliefs (undecidable in the general case)
- Online planning: the environment changes during deliberation or plan execution ☞ Requires monitoring the plan execution and detecting the need for replanning
- Scheduling: actions may have durations and require consuming resources. Related to Operations Research techniques such as Critical Path Method (CPM)