planning by translation

translate:

planning problem ⇒ other problem

then solve the other problem

useful?

---

useful?

planning problem ⇒ other problem

viable solution if:

- the other problem can be solved efficiently
- the translation is not too expensive

which other problem?

planning problem ⇒ other problem

popular targets of translation:

- propositional satisfiability
- constraint satisfaction
- integer programming

this course: propositional satisfiability

---

plansat

planning problem ⇒ sat

sat = find a model of a propositional formula

how to translate

planning problem ⇒ sat

planning	sat
actions to execute, resulting states	value of variables

[note] The sequence of actions to execute can be chosen; the state changes as a consequence. Depending on them, the goal is reached or not.

In a satisfiability problem, the values of the variables can be chosen. Depending on them, the formula may be satisfied or not.

---

correspondence

planning problem ⇒ sat

planning	sat
actions to execute, resulting states	value of variables

correct translation:

- each sequence of actions/states correspond to a propositional interpretation and vice versa
- if the sequence leads to the goal, the propositional interpretation satisfies the formula and vice versa

---

variables

planning problem ⇒ sat

planning	sat
actions: `a`, `b`, `c` state variables: `x`, `y`	propositional variables
goal reached	formula satisfied

propositional variables represent:
state and action at each step

plan length

plans can be exponentially long
(exponential in the size of the domain)

fix a maximal length `n`
only `n` states and `n-1` actions

---

time

planning	sat
actions: `a`, `b`, `c` state variables: `x`, `y`	propositional variables for: action at time `0` and `1` state at time `0`, `1` and `2`
goal reached	formula satisfied

example: time `0`, `1`, `2`

---

all variables

planning	sat
actions: `a`, `b`, `c` state variables: `x`, `y`	propositional variables: a_0 b_0 c_0 a_1 b_1 c_1 x_0 y_0 x_1 y_1 x_2 y_2
goal reached	formula satisfied

a propositional variable for:

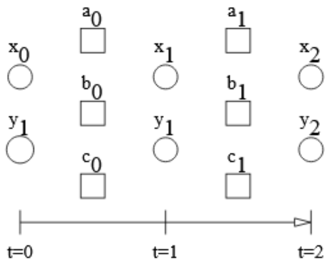
- each action at each time
- each state variable at each time

in this example: state variables `x` and `y` are Boolean

[note] If a state variable has more than two possible values, more propositional variables are needed to represent it.

The encoding of actions is redundant, since two propositional variables suffice to represent which of the three actions is executed a each time step. It however allow for a simpler propositional formula representing the planning problem.

all variables



if no formulae constrain the variables:

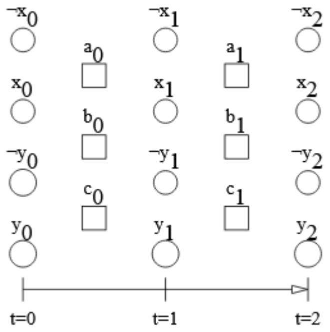
- every state is possible, at every time point
- every set of actions executable

instead:

- initial state fixed
- actions executable only if preconditions met
- no two actions at the same time
- next state from previous and action

---

negated variables



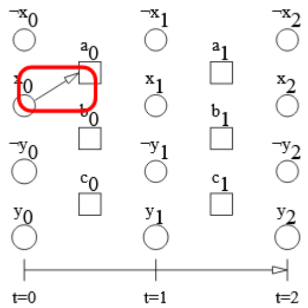
redundant (variable is negative if not positive)

simplifies connections
later generalized (mutex)

[note] The negated variables are not strictly necessary in this diagram, as a variable is negated exactly when not positive. However, this allows for a simple way of depicting negative preconditions and effects of an action.

Also, pairs of inconsistent literals are later generalized by the concept of mutex.

preconditions



example: x precondition to a

formula $x_0 \rightarrow a_0$?

WRONG!

[note] The diagram show how NOT to translate a precondition. Why it is wrong is explained next.

---

permissions and obligations

x_0 means that x is true at time 0
a_0 means that a is executed at time 0

x precondition to a
    means: if x is true, a *can* be executed
$x_0 \rightarrow a_0$
    means: if x is true at time 0, then a *must* be executed at time 0

---

evaluate the whole history

if a executed at time 0
then x must have been true at time 0

formula $a_0 \rightarrow x_0$

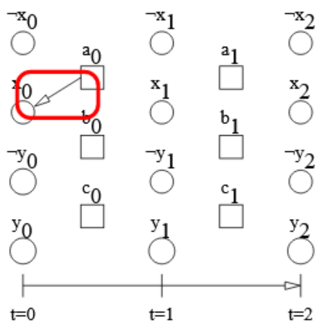
[note] Contrary to what the diagram may suggest, the plan will not be found by starting from the initial state and following some precondition-action links. It will be found by some method for propositional satisfiability, whose algorithm is not of interest at this point.

In order for this to work, the translation must be correct: an evaluation of the propositional variables satisfies the formula if and only if it represents a sequence of actions and the states that result from executing them.

A model represents a whole sequence of states and actions were known. The formula is what discriminates a random sequence of states and actions from one that is coherent with the particular domain under considerations.

An inefficient but correct solution for the satisfiability problem is to check every possible propositional interpretation in turn. The formula can be seen as something that takes one such interpretation and decides whether it is a plan. In terms of planning, it is as if the sequence of states and actions were known, and the only decision to take is whether it is an actual plan or not.

correctness of states-actions links



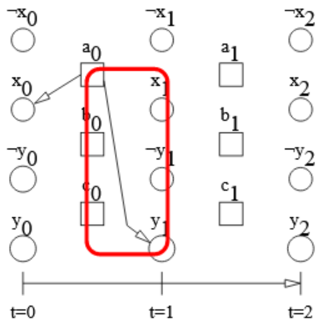
x is precondition to a:

if a is executed at time 0
then x was true at time 0

as a propositional formula: $a_0 \rightarrow x_0$

---

effects



a has effect y

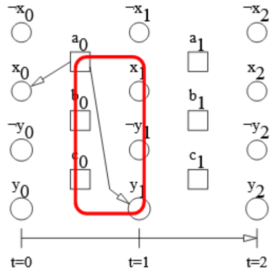
if a executed at time 0
y true at time 1

formula $a_0 \rightarrow y_1$

[note] This time, the formula is intutive: action implies effects.

It is however not sufficient.

inertia



what if a is *not* executed at time 0?

y_1 is true if y_0 was true

unless the action executed at time 0 has effect $\neg y$

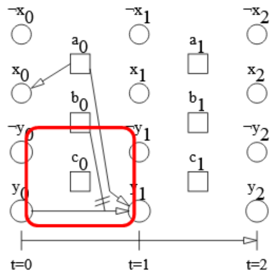
[note] Positive effects are easy to encode as a formula: if an action is executed, all its effects become true.

However, the formula is correct only if it also encode inertia: variables remain true if not changed by an action.

This in turns require considering all actions that change the value of the variable.

---

inertia and blocking actions



example: action b makes y false

not only $b_0 \rightarrow \neg y_1$

if y was true at time 0
then it is true at time 1
unless b is executed at time 0

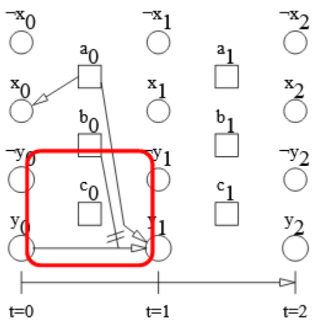
formula: $(y_0 \wedge \neg b_0) \rightarrow y_1$

not enough…

[note] The action b_0 has the usual effect of achieving its effect $\neg y_1$ but not only. It also blocks inertia on its negation, so that y_0 no longer produces y_1.

The formula $(y_0 \wedge \neg b_0) \rightarrow y_1$ is however not sufficient to correctly encode inertia.

missing bit

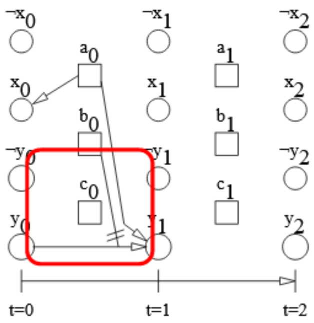


y_1 is true if:
a_0 is true, or
y_0 is true and b is false

otherwise, y_1 is false

[note] The condition about a_0, b_0 and y_0 tell when y_1 is true. The missing part is that whenever this condition is false then y_1 is false as well.

effect



y_1 is true *if and only if*:
a_0 is true, or
y_0 is true and b is false

formula:

$$y_1 \equiv a_0 \vee y_0 \wedge \neg b_0$$

[note] The formula is still simple also because the variable is only affected by two actions.

The arrows in the figure represent the formula $y_1 \leftarrow a_0 \vee y_0 \wedge \neg b_0$, which is incomplete. The correct encoding has $\equiv$ instead of the implication.

multiple actions affecting a variabl

example: variable `y`

- made true by actions `a`, `d`, `e`
- made false by action `b`, `h`

formula:

$$y_1 \equiv a_0 \lor d_0 \lor e_0 \lor y_0 \land \neg b_0 \land \neg h_0$$

---

effect formula

for every variable `x` and time `t`

$$x_{t+1} \equiv \bigvee a_t \lor x_t \land \bigwedge d_t$$

where:

- `a` are the actions making `x` true
- `d` are the actions making `x` false

---

translated so far…

- states and actions into variables
- preconditions and effects of actions

missing: initial state, goal, single action

initial state

example: $x=\top$, $y=\bot$

translated: $x_0 \land \neg y_0$

---

goal

example: $y=\top$

translated: $y_0 \lor y_1 \lor y_2$

---

single action

an action for each time point
no more than one

encoded, for each time `t`:

- $a_t \lor b_t \lor c_t$
- $\neg(a_t \land b_t) \land \neg(a_t \land c_t) \land \neg(b_t \land c_t)$

---

parallel plans

the assumption of single action may not be necessary:

- remove the single action assumption
- obtain a plan
- if the plan contains `a` and `b` both executed at time `0`: linearize the plan as either `a,b` or `b,a`

not always possible

actions not runnable in parallel

initially, x true and y false

a has precondition x and effect ¬y
b has precondition ¬y and effect ¬x

both actions executable in the initial state
plan with both a_0=T and b_0=T

linearizations:

sequence b,a
    invalid
sequence a,b
    valid

---

allowed parallel actions

two possibilities:

- allow actions a and b at the same time only if

  either a,b or b,a executable in sequence

- allow actions a and b at the same time only if

  both a,b or b,a executable in sequence

previous example: a,b valid, b,a invalid
first condition met, second not

check the existence of linearization

encode as a propositional formula:
some linearization of the parallel actions is executable

requires: produce every linearization
check each

check all linearization simpler?

---

check validity of all linearizations

equivalent condition:

  no action falsifies the precondition of another

example:

- parallel actions a, b at time t
  imples: all preconditions met at time t
- sufficient condition holds
  implies: all preconditions remains valid after executing any sequence of the parallel actions

missing sequences

in the example: a and b conflict
because b falsifies a precondition of a

condition excludes a and b at the same time t

but a,b was a valid plan

still obtained:
a at time t and b at time t+1

[note] The requirement on the actions exclude a and b at the same time, while in fact these two actions have a valid linearization a,b. This is not a problem, as this plan will still be obtained with a only at the current time and b only at the next.

While no plan is lost this way, the requirement is much simpler to check than the existence of a valid linearization. It is also simple to encode as a propositional formula, as it is shown next.

---

encoding as a formula

> no action falsifies the precondition of another

only for the same time t

formula: if b falsifies the precondition of a
add formula $b_t \rightarrow \neg a_t$
equivalent to $\neg b_t \vee \neg a_t$

for every time t

the formula

$P \wedge E \wedge I \wedge G \wedge L$

where:

P
: preconditions of actions
  ($a_0 \rightarrow x_0$...)

E
: effects, negative effects and inertia
  ($y_1 \equiv a_0 \vee y_0 \wedge \neg b_0 \ldots$)

I
: initial state
  ($x_0 \wedge \neg y_0$)

G
: goal
  ($y_0 \wedge y_1 \wedge y_2$)

L
: linearizability of paralel actions
  ($\neg(a_0 \wedge b_0) \ldots$)

---

plansat

given: planning problem

- translate it into a formula $P \wedge E \wedge I \wedge G \wedge L$
- find a propositional model
  (how? next part of the course)
- variables a_0, b_0, ..., c_n tell the actions

why?

formula built so that:

a propositional interpretation satisfies the formula
if and only if
the values of the variables represent a plan (actions and states)

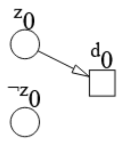
improvements

method works, but:

- many variables
  already 10 just for two Boolean variables and three actions over two steps
- some not really useful
  example: c not executable at time 0, variable c_0 useless
  variable false, made true by c: useless at time 0 and 1

---

impossible actions



z initially false

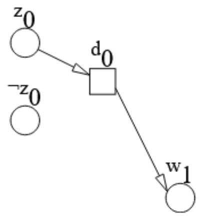
d cannot be executed

variable d_0 useless

also…

---

consequences of impossible actions



w initially false
w only made true by d

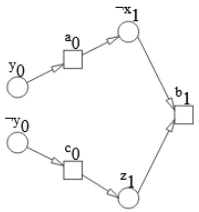
d_0 removed $\Rightarrow$ w_1 removed

may fire removal of other actions, etc.

[note] This is the first condition that allows the removal of variables: if an action cannot be executed at the initial time, its variable at time 0 can be removed. This implies the removal of all variables that are effects of actions that cannot be executed. In turn, this allows for the removal of actions at time 1 that have them as preconditions.

Other conditions will be shown that allow simplifying the formula.

mutex



y cannot be both true and false at the same time
⇒ a and c cannot be executed at the same time

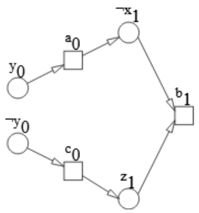
so?

[note] This is different than linearizability. Indeed, a and c could be executed in parallel, since a falsifies no precondition of c and c falsifies no precondition of

The reason why a and c cannot be executed at the same time is different: their preconditions cannot be achieved at the same time.

---

executable actions



variables a_0 and c_0 necessary
because action a alone can be executed at time 0
and action c alone can be executed at time 0

but: they cannot be *both* executed at the same time

consequence: $\neg x_1$ and z_1 cannot both be true

b not executable at time a

remove b_1
and its consequences, if not otherwise achievable,
etc.

---

what to remove at time t

remove a literal
only if *all* actions making it true removed from time $t-1$

remove an action
- one of its precondition at time t removed, or
- two of its preconditions cannot be both true at time t

mutex

both for literals and actions:

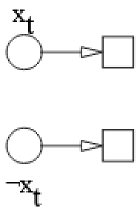
variables
two literals that cannot both be true at a certain time
actions
two actions that cannot both be executed at a certain time

related: if two literals are only made true by two actions in mutex they are in mutex as well

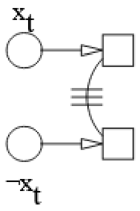
same for actions

---

actions in mutex: preconditions



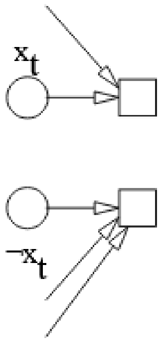
actions have conflicting preconditions



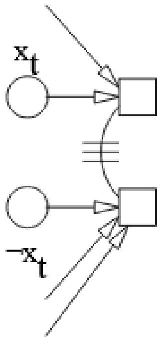
they cannot be executed at the same time

more generally…

actions in mutex: preconditions, more generall



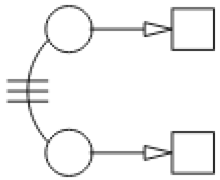
even if the actions have other preconditions



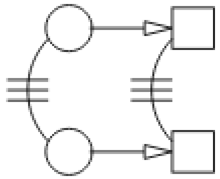
still they cannot be executed at the same time

even more generally…

actions in mutex: preconditions, generally

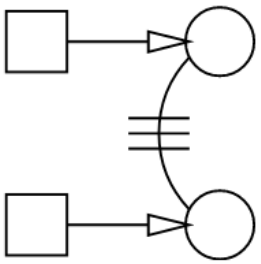


actions may have preconditions in mutex

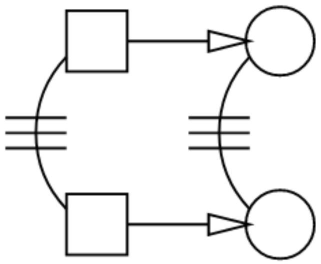


regardless of other preconditions, actions are in mutex

actions in mutex: effects



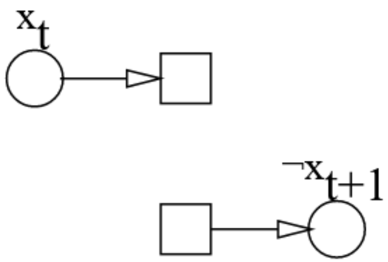
actions have conflicting effects
e.g., they are x_{t+1} and $\neg x_{t+1}$



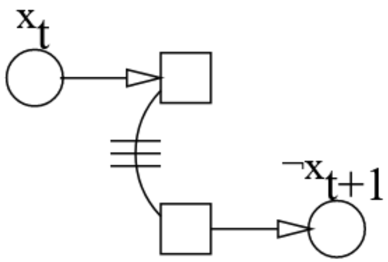
actions are in mutex

[note] In theory, a mutex on actions could derive from a mutex on effects. This is however not done because it requires going back and forth in the graph of variables, from a level of variables to the previous level of actions. Only opposite literals are checked, not the more general condition of literals in mutex.

actions in mutex: linearizability



one action makes false the precondition of the other



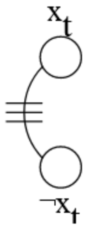
actions are in mutex

[note] This condition cannot be generalized to “x_t and y_{t+1} are in mutex”, not even theoretically.

A mutex is only possible between literals (or actions) at the same time point. A literal at time t cannot be in mutex with one at time $t+1$, like in this case. This is why actions are in mutex only if “one generate the opposite of the precondition of the other” and *not* “the effect of one and the precondition of the other are in mutex.”.

literals in mutex: base case

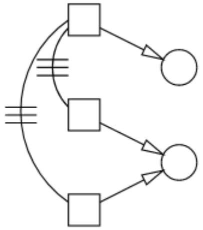
opposite literals at the same time point



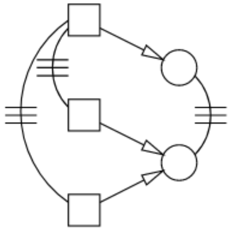
literals are in mutex

---

literals in mutex: conflicting actions



literal obtained by some actions
literal obtained by some other
each in conflict with some



literals are in mutex

[note] The first literal is the effect of executing one of a set of actions. The same for the second. But each action making the first true is in mutex with one making the second true.

inertia

x_1 and y_1 in mutex if (example):

- x_1 effect only of a_0 and b_0
- y_1 effect only of c_0
- c_0 in mutex with a_0
- c_0 in mutex with b_0

every action achieving x_1 in mutex with every action achieving y_1

but x_1 also achievable from x_0 by inertia

x_0 true, execute c_0 only
result: x_1 and y_1 both true
not a mutex

[note] This example shows that neglecting intertia may generate a mutex like the one on x_1 and y_1 while the two variables can in fact be true at the same time.

The solution is to include inertia in the computation of mutexes.

Since intertia may make some variables true, it also affect the computation of useless variables.

---

inertia as an action

action with x_t as precondition and x_{t+1} as effect
action with $\neg x_t$ as precondition and $\neg x_{t+1}$ as effect

too simple?

[note] The propositional formula used for encoding inertia was more complicated than this. It involved all actions making a variable true and all making it false. That was however done towards a different aim. The difference is explained next.

intertia-action vs. intertia-formula

when encoding the problem as a propositional formula:
complex formula $x_{t+1} \equiv \bigvee a_t \lor x_t \land \bigwedge d_t$

when checking mutexes and useless variables:
action with x_t as precondition and x_{t+1} as effect
action with $\neg x_t$ as precondition and $\neg x_{t+1}$ as effect

not enough in a formula
missing: mandatory action if no action falsifing its effect is executed

enough for checking mutexes and useless variables
only needs to know that variables could be true by intertia

[note] Such actions would not constraint enough the propositional interpretation, as they could just not be executed at some time point even if the action at that time point do not change x.

However, when checking mutexes and useless variables, the procedure of expansion only needs to know which variables could be true. It works like assuming that everything that could be made true might be, and excluding everything else.

As a result, intertia can be encoded as an action that can be executed if x_t is true and makes x_{t+1} true. Another action encodes intertia on the negated variable.

[italian] mandatory = obbligatorio

---

overall algorithm

- start from variables at time 0

  remove false literals
  add mutex between opposite literals

- check actions at time 0

  remove unexecutable actions
  add mutexes between actions

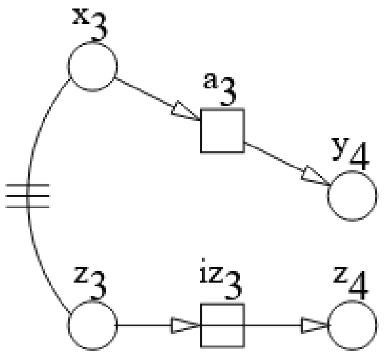
- check variables at time 1

  remove unachiavable literals
  add mutexes between variables

- …

[note] The algorithm alternates between variables at time t, actions at time t and variables at time $t+1$ etc. Each time, variables can be removed and mutexes added based on what obtained from the previous step.

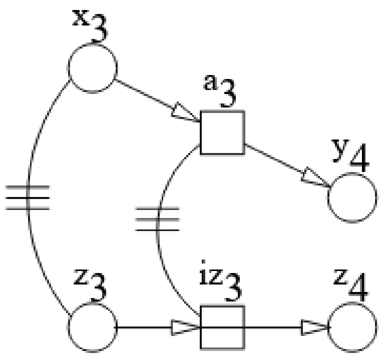
example, with inertia



x_3 and z_3 in mutex

---

action mutex

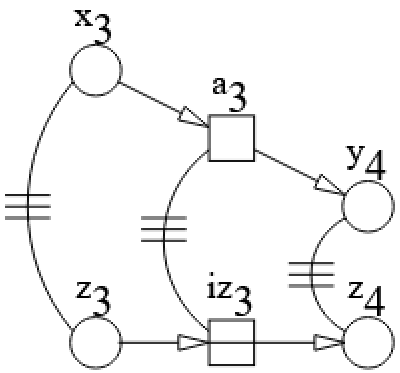


x_3 and z_3 in mutex

x_3 precondition to a
z_3 precondition to inertia action

therefore: mutex between a_3 and inertia on z_3

literal mutex



only way to achieve y_4 is by a
only way to achieve z_4 is by inertia

therefore: mutex between y_4 and z_4

goal

goal x and y:
both variables at time t
not in mutex

necessary condition
not checked: triple mutexes, for example
(actions a, b and c not all executable at the same time, but each two of them are)

[note] The algorithm for finding mutexes and useless variables is implcitely based on over-optimistic assumptions: if an action could be executed, its effects are achievable, and if actions are not in mutex they can all be executed. It only checks these conditions locally, neglecting the other variables and actions.

---

time steps

if goals not at level t, or in mutex
expand another level

otherwise: translate into a formula and check satisfiability

---

mutexes as formulae

a_t and b_t in mutex
add formula $\neg a_t \vee \neg b_t$

same for literals

formula increase?

[note] This addition may look counterproductive: the algorithm looks for mutexes and useless variables in order to simplify the formula resulting from the conversion, but not new parts are added to it.

---

formula increase

$\neg a_t \vee \neg b_t$
$\neg x_t \vee \neg y_t$

formulae encoding mutexes

enlarge the formula obtained by translation
but: make it simpler to check for satisfiability

[note] For example, if a_t is true then the first formula allows immediately concluding that b_t is false. This depdends on the specific satisfiability algorithm, but short subformulae like this may prove useful for directing the search for a satisfying interpretation.

summary

1. planning ⇒ satisfiability of a formula
2. planning graph makes the formula simpler

historically:
graphplan in 1995,
the planning graph then used in plansat (1996)

---

graphplan

build the graph

go back from the goal to the initial state

---

graphplan, necessary conditions

- build the graph with `n` time points
- if not all goal literals in the graph:
  `n++` and goto 1
- some goals in mutex:
  `n++` and goto 1
- search for a plan (to be continued)

graphplan, search for a plan

goals not in mutex: *maybe* a plan exists
necessary but not sufficient condition

- …
- to achieve a set of goals:
  choose a set of non-mutex actions that achieve all goal
- try to achieve its preconditions
  same way: choose a set of non-mutex actions, etc.
- otherwise, `n++` and expand the graph

first point already hard
use heuristics

---

innovativity of graphplan

before: search for a path in the graph of states

graphplan: introduced the graph of variables-actions-variables-…

then: used in plansat
then: heuristics based on the graph of variables-actions-variables-…

e.g., FF heuristics, etc.