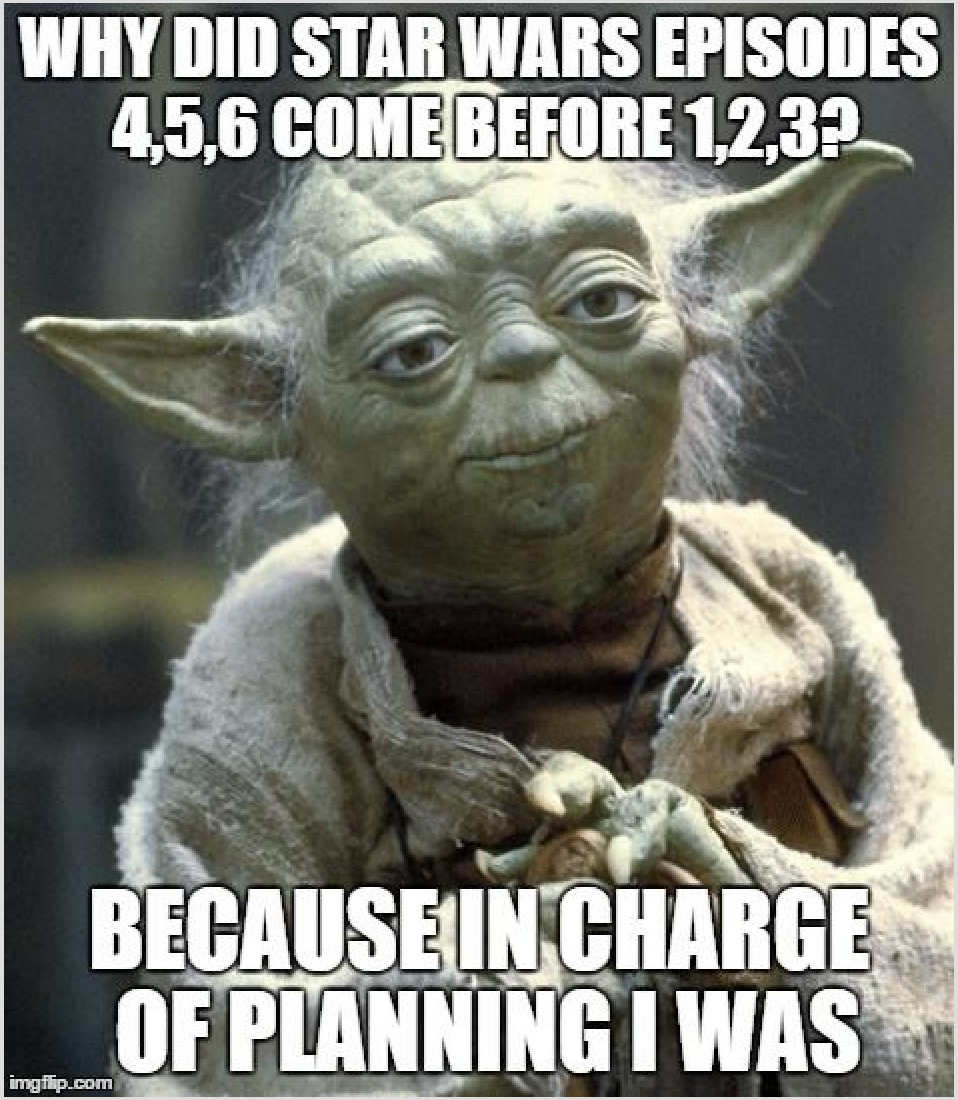
Planning (Ch. 10)



Forward search

Action($GoTo(x, y, z)$,
Precondition: $At(x, y) \wedge Mobile(x)$,
Effect: $\neg At(x, y) \wedge At(x, z)$)

Last time...

Initial: At(Truck, UPSD) ^ Package(UPSD, P1)
^ Package(UPSD, P2) ^ Mobile(Truck)

Goal: Package(H1, P1) ^ Package(H2, P2)

Action($Load(m, x, y)$,
Precondition: $At(m, y) \wedge Package(y, x)$,
Effect: $\neg Package(y, x) \wedge Package(m, x) \wedge At(m, y)$)

Action($Deliver(m, x, y)$,
Precondition: $At(m, y) \wedge Package(m, x)$,
Effect: $\neg Package(m, x) \wedge Package(y, x) \wedge At(m, y)$)

Heuristics for planning

Backwards search has a smaller branching factor in general, but it is hard to use heuristics

This is due to it looking at sets of states, and not a single state for the next action

For this reason, it is often better to apply a good heuristic to the dumb forward search

Heuristics for planning

Reformulate our grilling problem as actions:

$$\text{Action}(\ MakeSandwitch(x),$$
$$\text{Precondition: } Meat(x) \wedge Make(Bread, x, Bread))$$
$$\text{Effect: } \neg Meat(x) \wedge Sandwitch(Bread))$$

If our goal is just “Sandwitch(Bread)”, backtracking search would try to solve:

$$Meat(x) \wedge Make(Bread, x, Bread))$$

... but since “x” is still a variable, this represents a set of states rather than one

Heuristics for planning

In “search” we had no generalize-able heuristics as each problem could be different

Heuristics in planning are found the same way, we (1) relax the problem (2) solve it optimally

Two generic ways to always do this are:

1. Add more actions
2. Reduce number of states

Heuristics: add actions

Multiple ways to add actions (to goal faster):

1. Ignore preconditions completely - also ignore any effects not related to goals

   This becomes set-covering problem, which is NP-hard but has P approximations

2. Ignore any deletions in effects (i.e. anything with ¬), also NP-hard but P approximation

Ignore preconditions

By simply removing preconditions, we allow every action to happen at every state

$$\begin{array}{l} \quad\text{Action}(\ GoTo(x,y,z), \\ \text{Precondition: } At(x,y) \land Mobile(x), \\ \text{Effect: } \neg At(x,y) \land At(x,z)) \end{array}$$

↓

$$\begin{array}{l} \quad\text{Action}(\ GoTo(x,y,z), \\ \text{Precondition: } \\ \text{Effect: } \neg At(x,y) \land At(x,z)) \end{array}$$

Ignore preconditions

More importantly for the solution is how the Delivery action changes

The USPD can now just directly deliver to houses, so goal is: Deliver(USPD, P1, H1) and then Deliver(USPD, P2, H2)

$$\text{Action}(\ Deliver(m, x, y),$$
$$\text{Precondition:}$$
$$\text{Effect: } \neg Package(m, x) \wedge Package(y, x))$$

Ignore negative effects

To use this heuristic, the goal cannot have negative functions/literals (i.e. $\neg At(Truck, H2)$)

This can always be rewritten to something else (for above $At(Truck, USPD) \vee At(Truck, H1) \vee At(Truck, Truck)$)

$$\begin{array}{l} \text{Action}(\ GoTo(x, y, z), \\ \text{Precondition: } At(x, y) \wedge Mobile(x), \\ \text{Effect: } \boxed{\neg At(x, y)} \wedge At(x, z)) \end{array}$$

$\downarrow$

$$\begin{array}{l} \text{Action}(\ GoTo(x, y, z), \\ \text{Precondition: } At(x, y) \wedge Mobile(x), \\ \text{Effect: } At(x, z)) \end{array}$$

Ignore negative effects

For the UPS delivery example, it does not help us find a solution faster (min is 6 still)

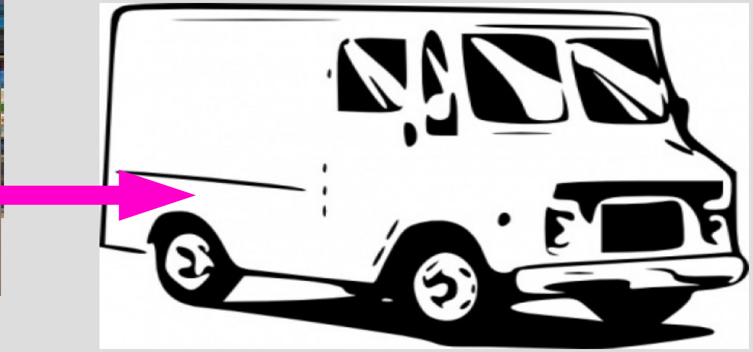
However, there are many more solutions as every action “copies” instead of “moves”

For example, a solution could be:
Move, Move, Load, Load, Deliver, Deliver

This is possible as truck exists at all 3 spots!

$$At(Truck,UPSD) \wedge At(Truck,H1) \wedge At(Truck,H2)$$
$$\wedge Package(UPSD,P1) \wedge Package(UPSD,P2) \wedge Mobile(Truck)$$

After 2 moves... then load...



$$At(Truck, UPSD) \land At(Truck, H1) \land At(Truck, H2) \land Package(Truck, P2)$$
$$\land Package(UPSD, P1) \land Package(UPSD, P2) \land Mobile(Truck)$$

After 2 moves... then load...

Heuristics: group states

Group similar states together into “super states” and solve the problem within “super states” separately (divide & conquer)

An admissible but bad heuristic would be the maximum of all “super states” individual solutions (but this is often poor)

A possibly non-admissible would be the sum of all “super states” (need independence)

Heuristics: group states

These “super states” can created in many ways

1. Delete relations/fluents (e.g. no more “At”)
2. Merge objects/literals (e.g. merge UPSD and Truck)

You then need to solve two problems:

1. Between the abstract “super states”
2. Within each “super state”

Heuristics: group states

Consider if there were 3 houses, but only two needed packages

We could remove all “At”s for this third house, as we can easily abstract it away

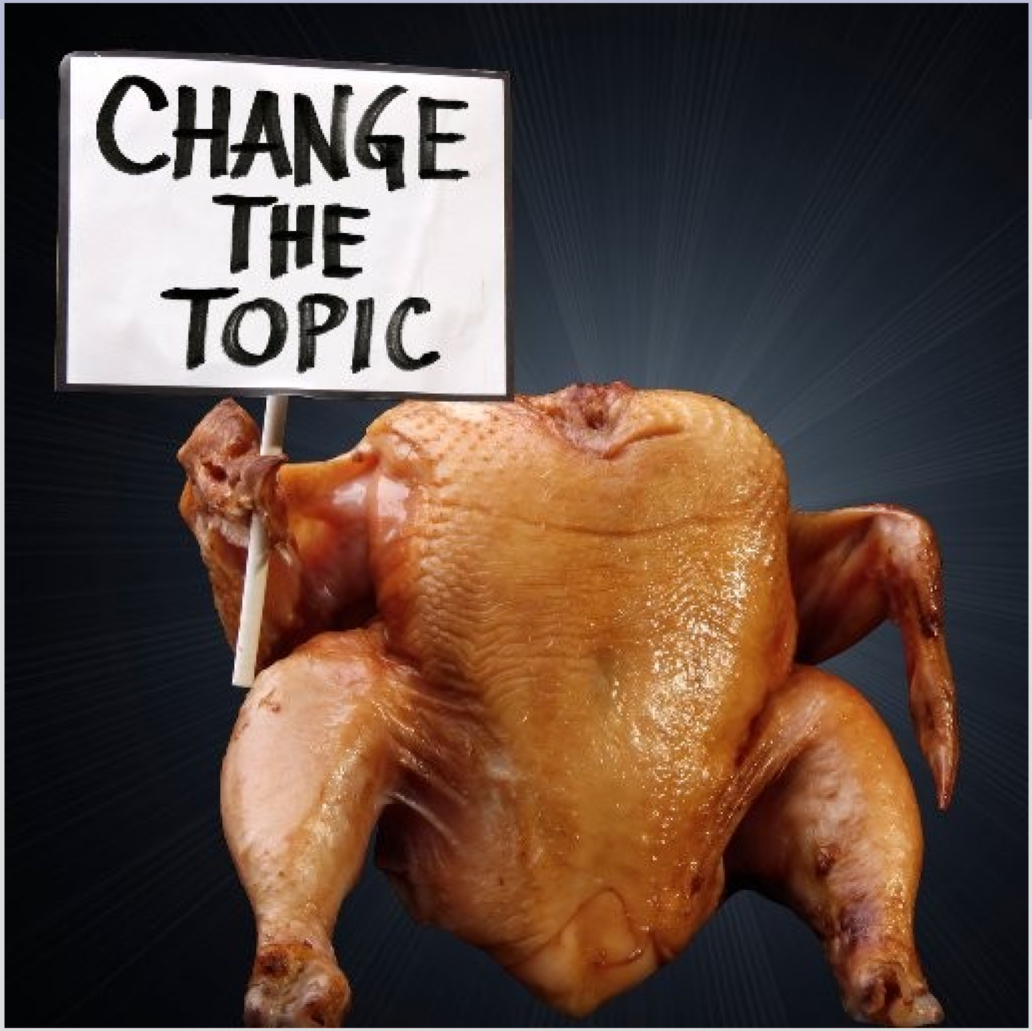
In this case the “super state” solution is the actual solution as there is no need to add back in a third house

Heuristics: group states

For example, if we were instead delivering
3 packages, 1 to H1 and 2 to H2...

We combine the two packages for H2 into a single “super package” with only one load and deliver (overall “super state” solution)

We then can simply see that each load/deliver corresponds to two individual loads/delivers (within super state solution)


CHANGE
THE
TOPIC

Graph Plan

A heuristic we will go over in detail is graph planning, which tries to do all possible actions at each step

The graph plan heuristic is nice because it is always admissible and computable in P time

The basic idea of graph plan is to track all the statements that could be true at any time

Graph Plan

Graph plan is an underestimate because once a relation/literal is added, it is never removed

Unlike the “remove negative effects” heuristic, we allow both negative and positive effects

But we can also use any preconditions that have been found anytime before (not quite as open as completely removing them)

Graph Plan

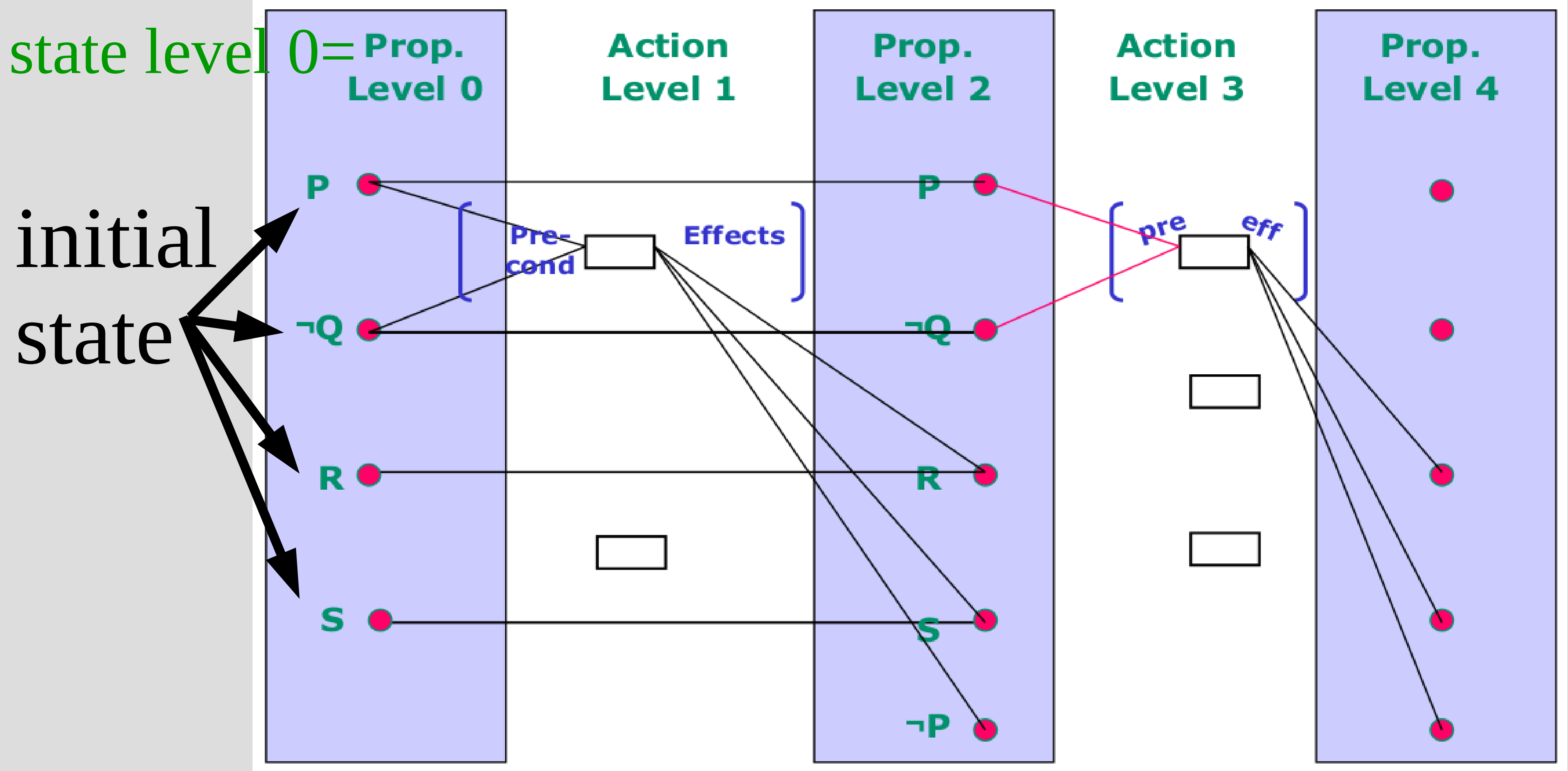
These simplifications/relaxations probably make the problem too easy

So we also track pairs of both actions and literals that are in conflict (called mutexes)

First, let's go over how to convert actions and relations into graph plan, then later we will add in the mutexes

Graph Plan

Graph plan will alternate between possible facts ("state level") and actions ("action level")



Graph Plan

You start with the relations of the initial state on the left (now explicitly stating negatives)

Then you add “no actions” which simply keep all the relationships the same but move them to the right

Then you add actions, which you do by linking preconditions on the left to resulting effects on the right (adding any new ones)

Graph Plan

Consider this problem:

Initial: $Sleepy(me) \wedge Hungry(me)$

Goal: $\neg Sleepy(me) \wedge \neg Hungry(me)$

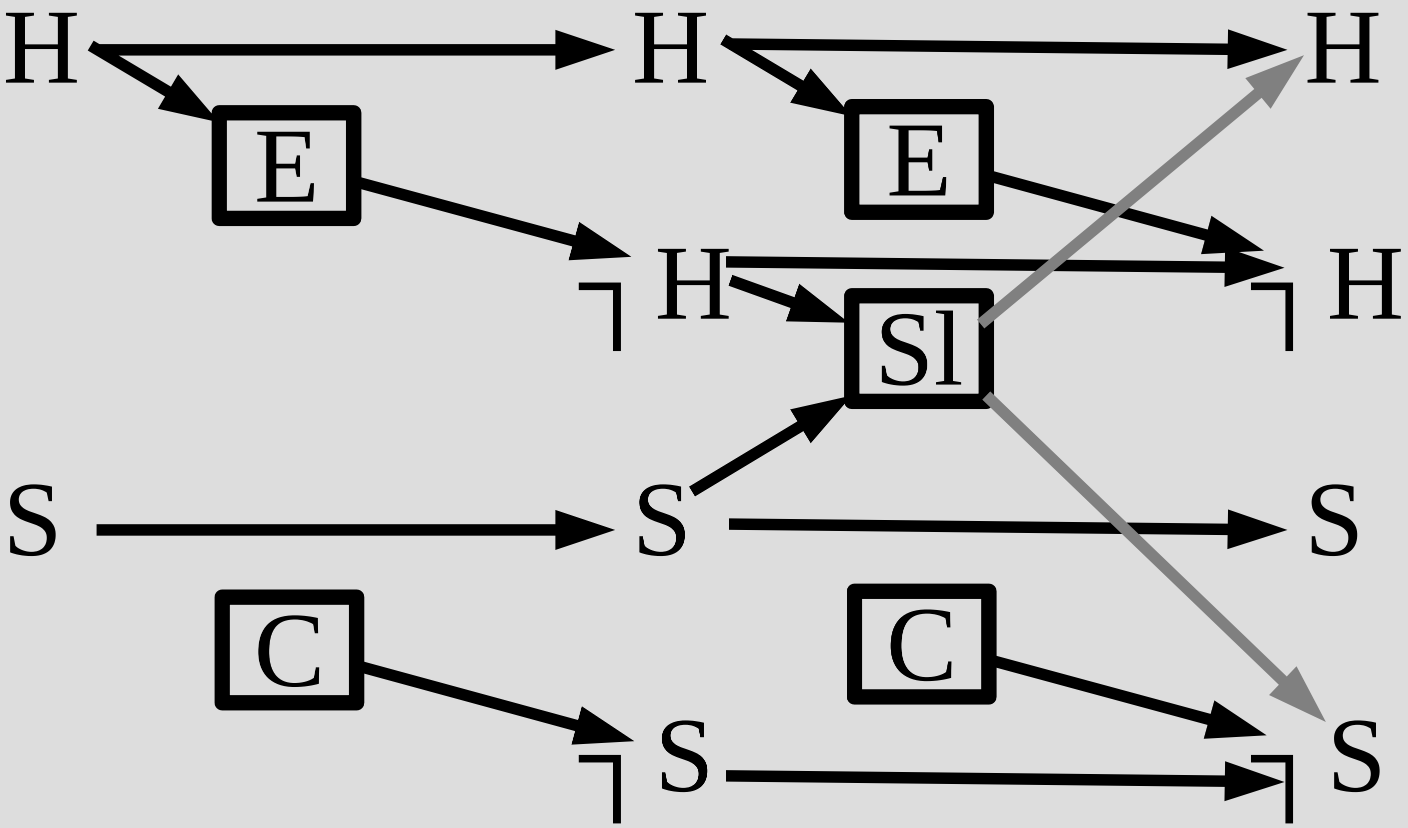
Action($Eat(x)$,
Precondition: $Hungry(x)$,
Effect: $\neg Hungry(x)$)

Action($Coffee(x)$,
Precondition: ,
Effect: $\neg Sleepy(x)$)

Action($Sleep(x)$,
Precondition: $Sleepy(x) \wedge \neg Hungr(x)$,
Effect: $\neg Sleepy(x) \wedge Hungry(x)$)

Graph Plan

Consider this problem:



Graph Plan

Each set of relations/literals are what we call levels of the graph plan, S = states, A = actions

State level 0 is $S_0 = \{H, S\}$

$A_0 =$ {C, E, all “no ops”}

$S_1 = \{H, \neg H, S, \neg S\}$

$A_1 =$ {C, E, Sl, all “no ops”}

$S_2 = \{H, \neg H, S, \neg S\}$

Graph Plan

You do it! (show 3 state and 2 action levels)

Initial: $\neg Money(me) \wedge \neg Smart(me) \wedge \neg Debt(me)$

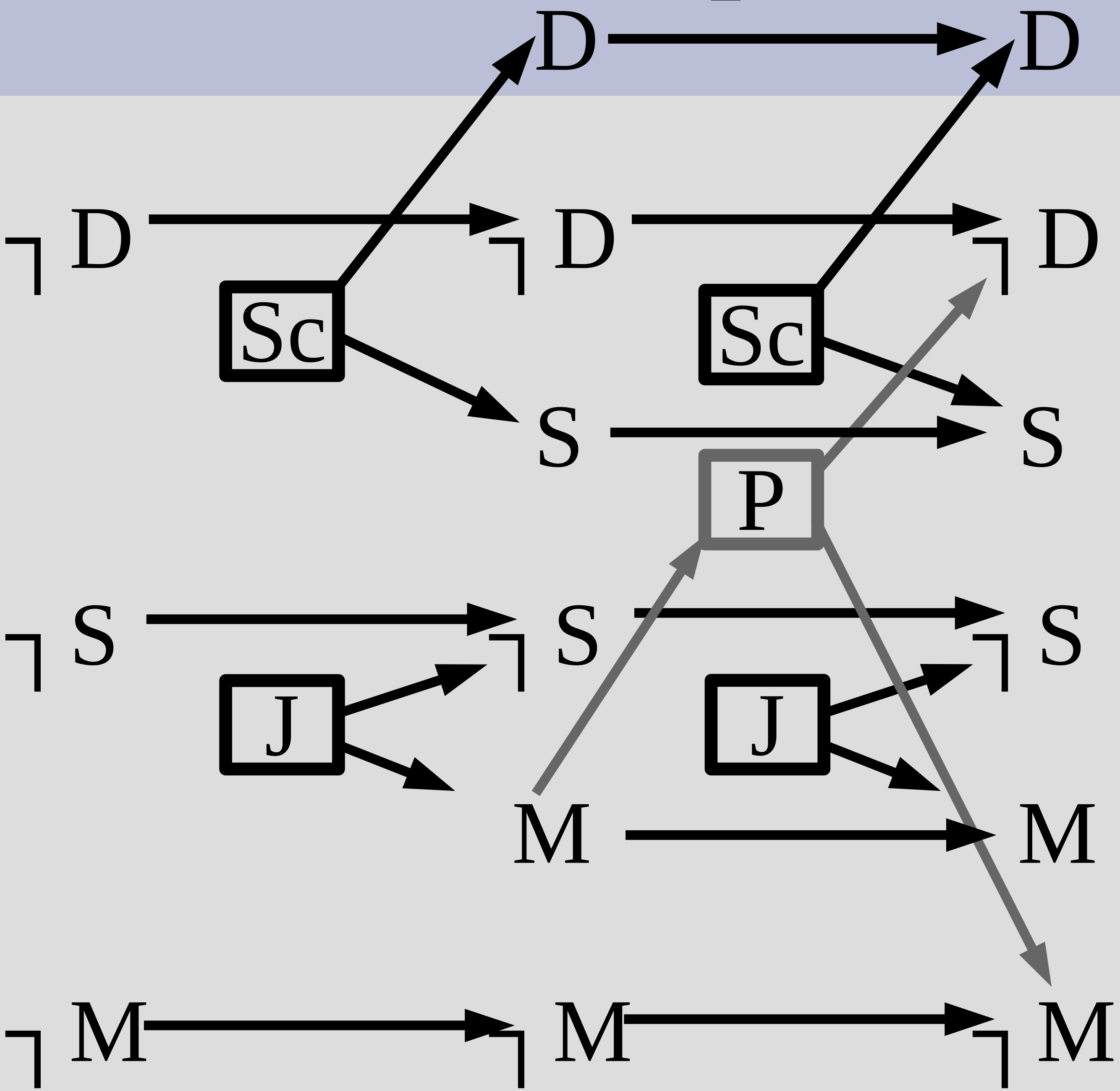
Goal: $Money(me) \wedge Smart(me) \wedge \neg Debt(me)$

Action($School(x)$,
Precondition: ,
Effect: $Debt(x) \wedge Smart(x)$)

Action($Job(x)$,
Precondition: ,
Effect: $Money(x) \wedge \neg Smart(x)$)

Action($Pay(x)$,
Precondition: $Money(x)$,
Effect: $\neg Money(x) \wedge \neg Debt(x)$)

Graph Plan



Graph Plan

The graph plan allows multiple actions to be done in a single turn, which is why S_1 has both ¬ Sleepy(me) and ¬ Hungry(me)

You keep building the graph until either:
(1) You find your goal (more on this later)
(2) The graph converges (i.e. states, actions and mutexes stop changing)

Mutexes

A mutex are two things that cannot be together (i.e. cannot happen or be true simultaneously)

You can put mutexes:
1. Between two relationships/literals
2. Between actions

There are different rules for doing mutexes between actions vs. relations

Mutexes: actions

For all of these cases I will assume actions two actions: A1 and A2

These actions have preconditions and effects: Pre(A1) and Effect(A1), respectively

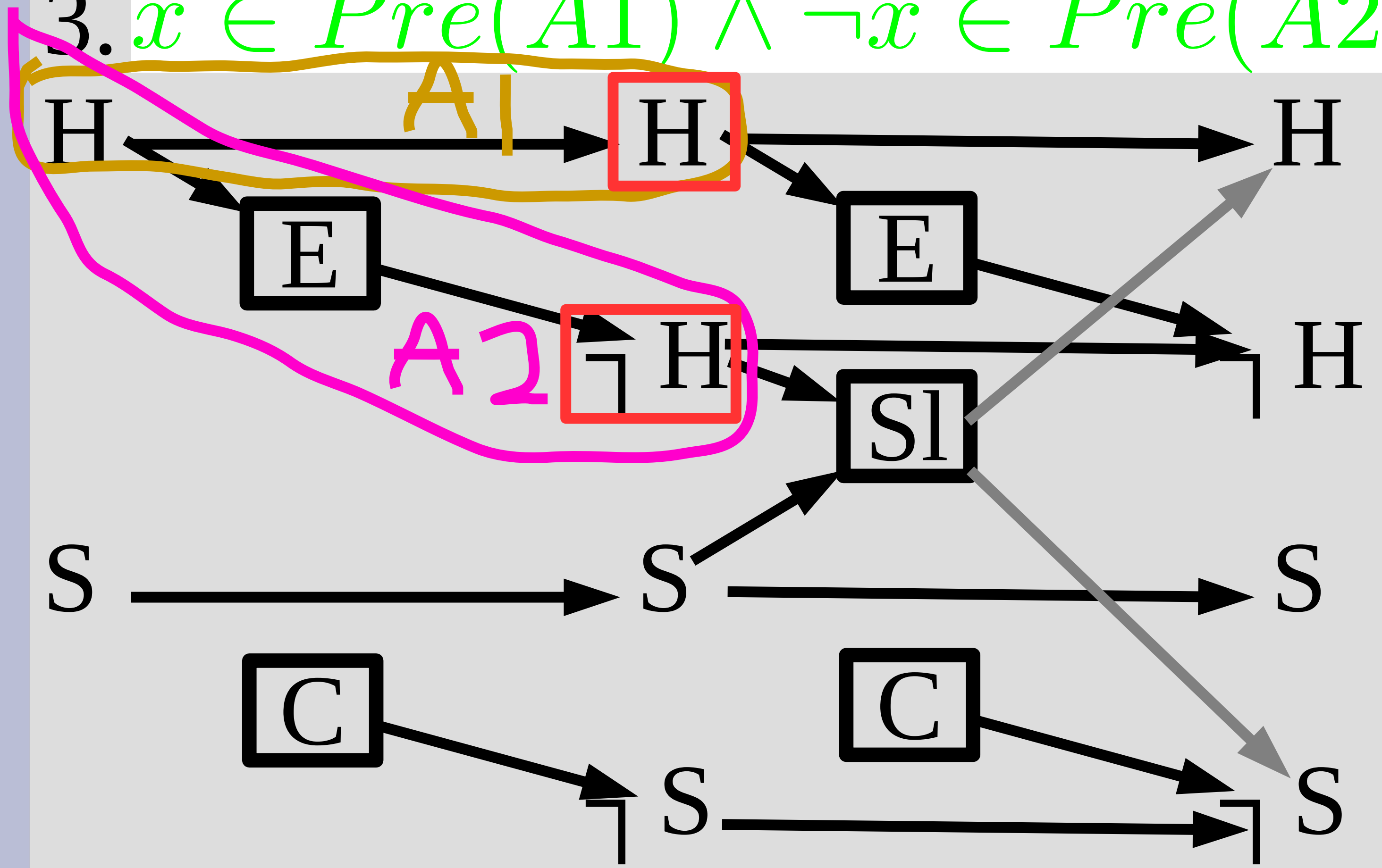
For example, I will abbreviate below as:

$$
\begin{array}{lcl}
\text{Action}(\ Eat(x), & & A1 = Eat \\
\text{Precondition: } Hungry(x), & \longrightarrow & H \in Pre(A1) \\
\text{Effect: } \neg Hungry(x)) & & \neg H \in Effect(A1)
\end{array}
$$

Mutexes: actions

Mutex Action rules:

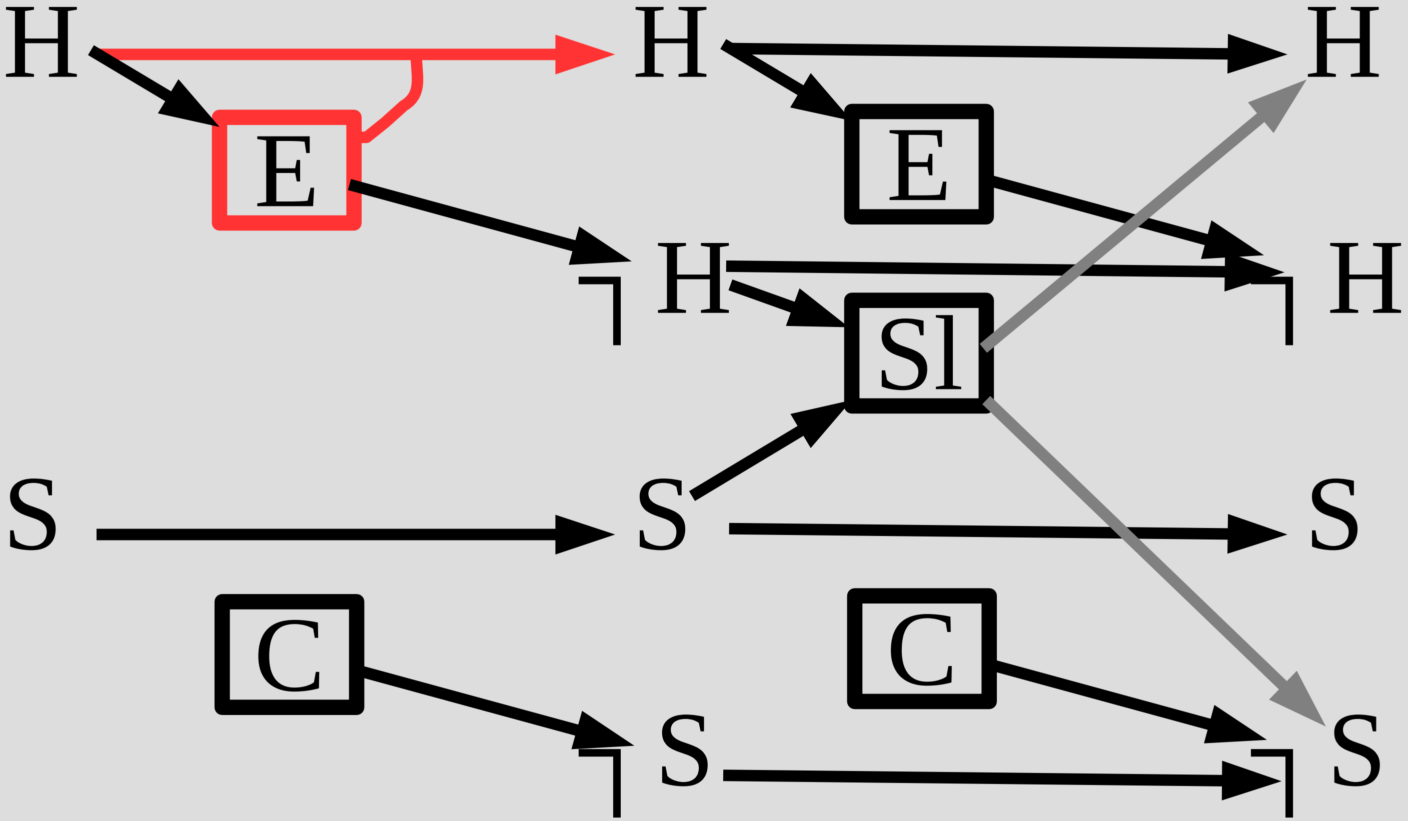
1. $x \in Effect(A1) \land \neg x \in Effect(A2)$
2. $x \in Pre(A1) \land \neg x \in Effect(A2)$
3. $x \in Pre(A1) \land \neg x \in Pre(A2)$



Mutexes: actions

Mutex Action rules:

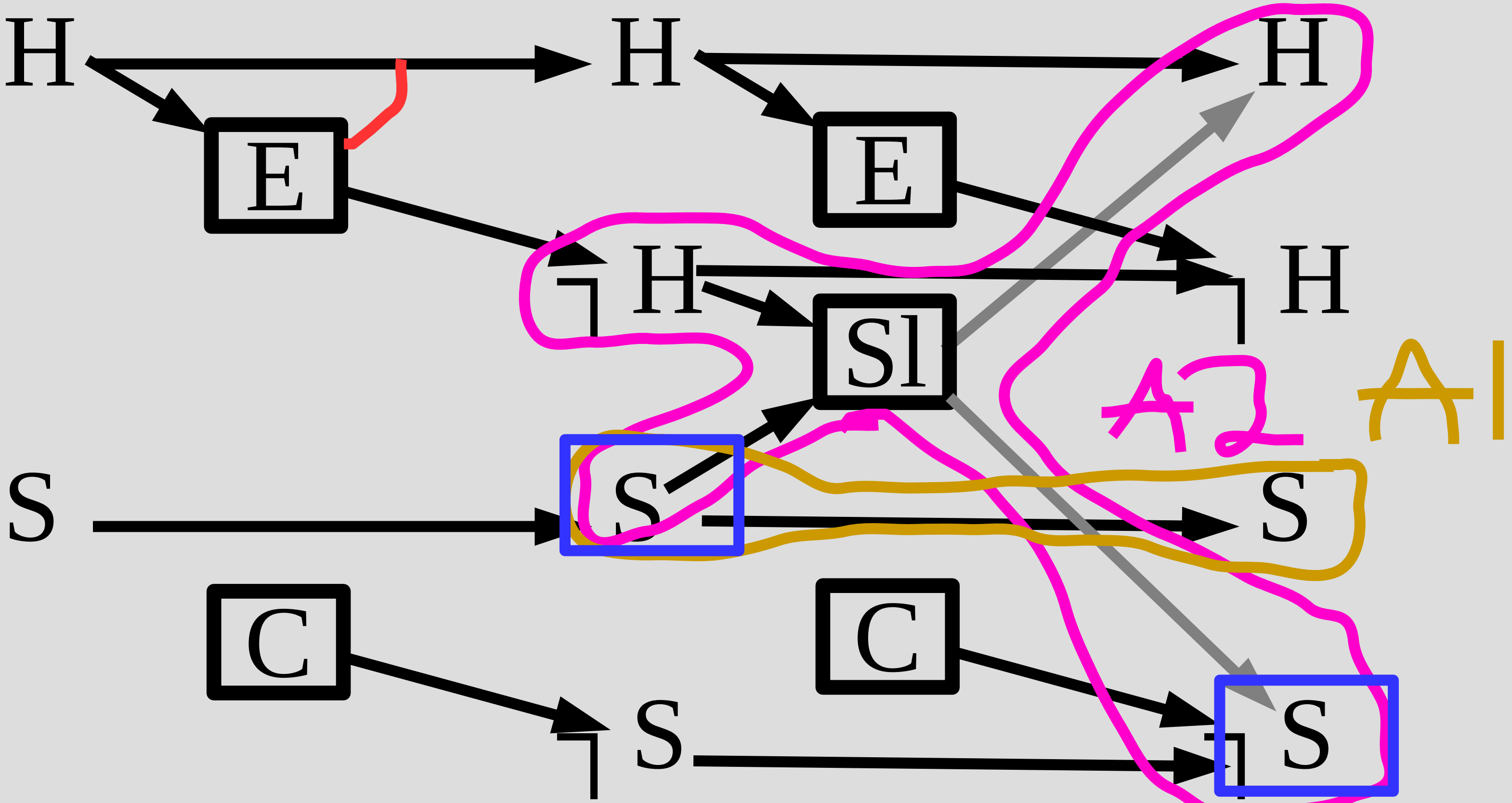
1. $x \in Effect(A1) \wedge \neg x \in Effect(A2)$
2. $x \in Pre(A1) \wedge \neg x \in Effect(A2)$
3. $x \in Pre(A1) \wedge \neg x \in Pre(A2)$



Mutexes: actions

Mutex Action rules:

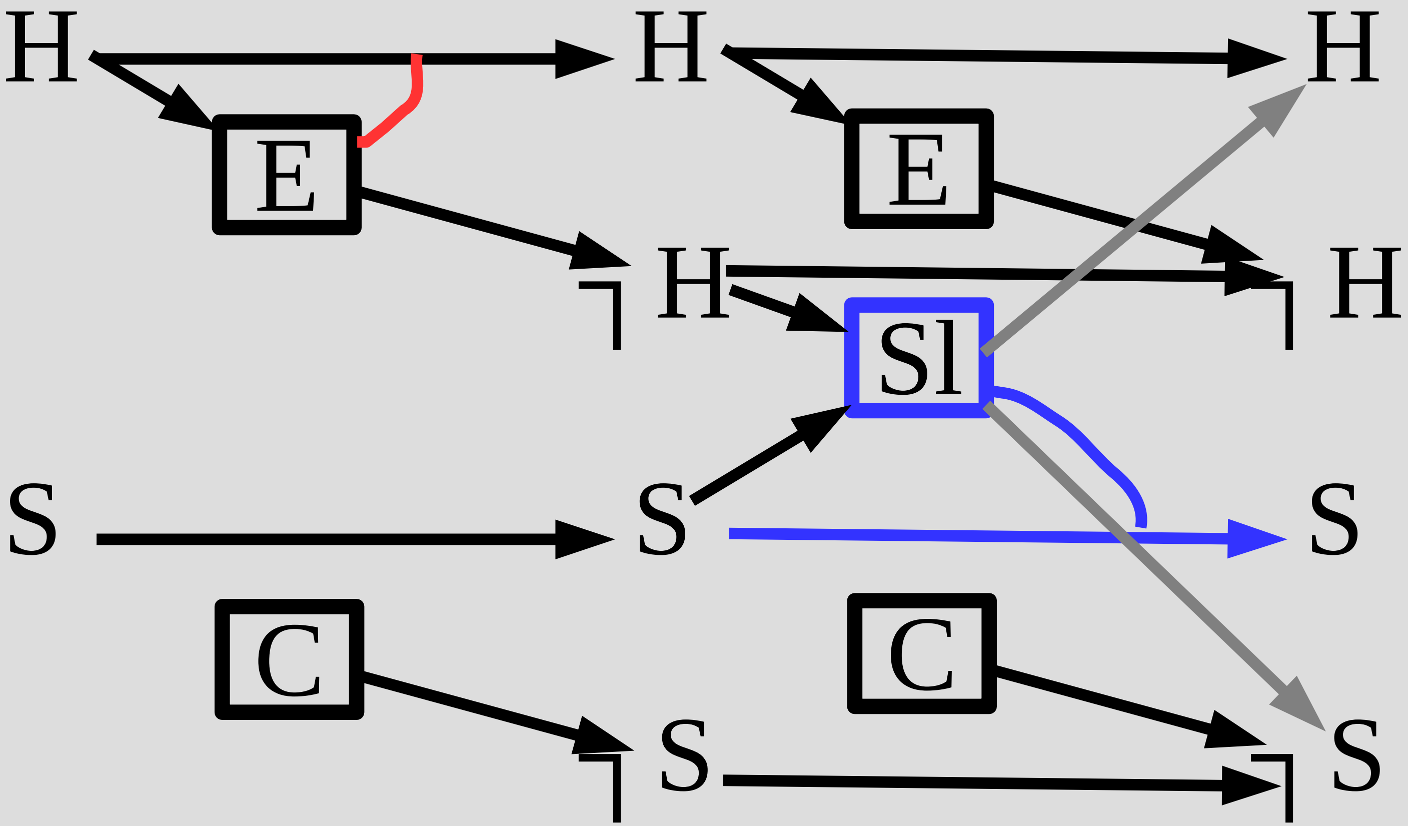
1. $x \in Effect(A1) \land \neg x \in Effect(A2)$
2. $x \in Pre(A1) \land \neg x \in Effect(A2)$
3. $x \in Pre(A1) \land \neg x \in Pre(A2)$



Mutexes: actions

Mutex Action rules:

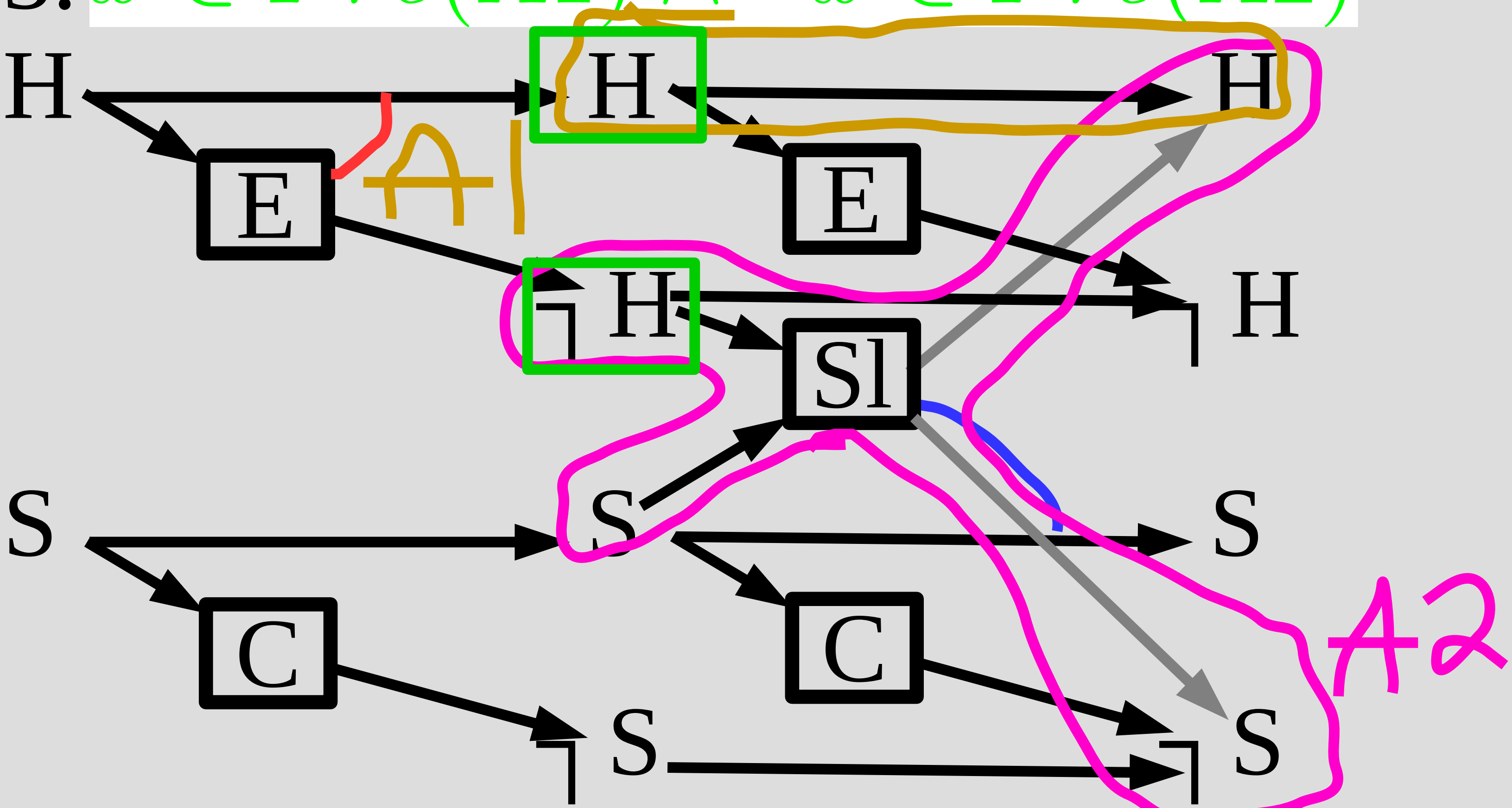
1. $x \in Effect(A1) \wedge \neg x \in Effect(A2)$
2. $x \in Pre(A1) \wedge \neg x \in Effect(A2)$
3. $x \in Pre(A1) \wedge \neg x \in Pre(A2)$



Mutexes: actions

Mutex Action rules:

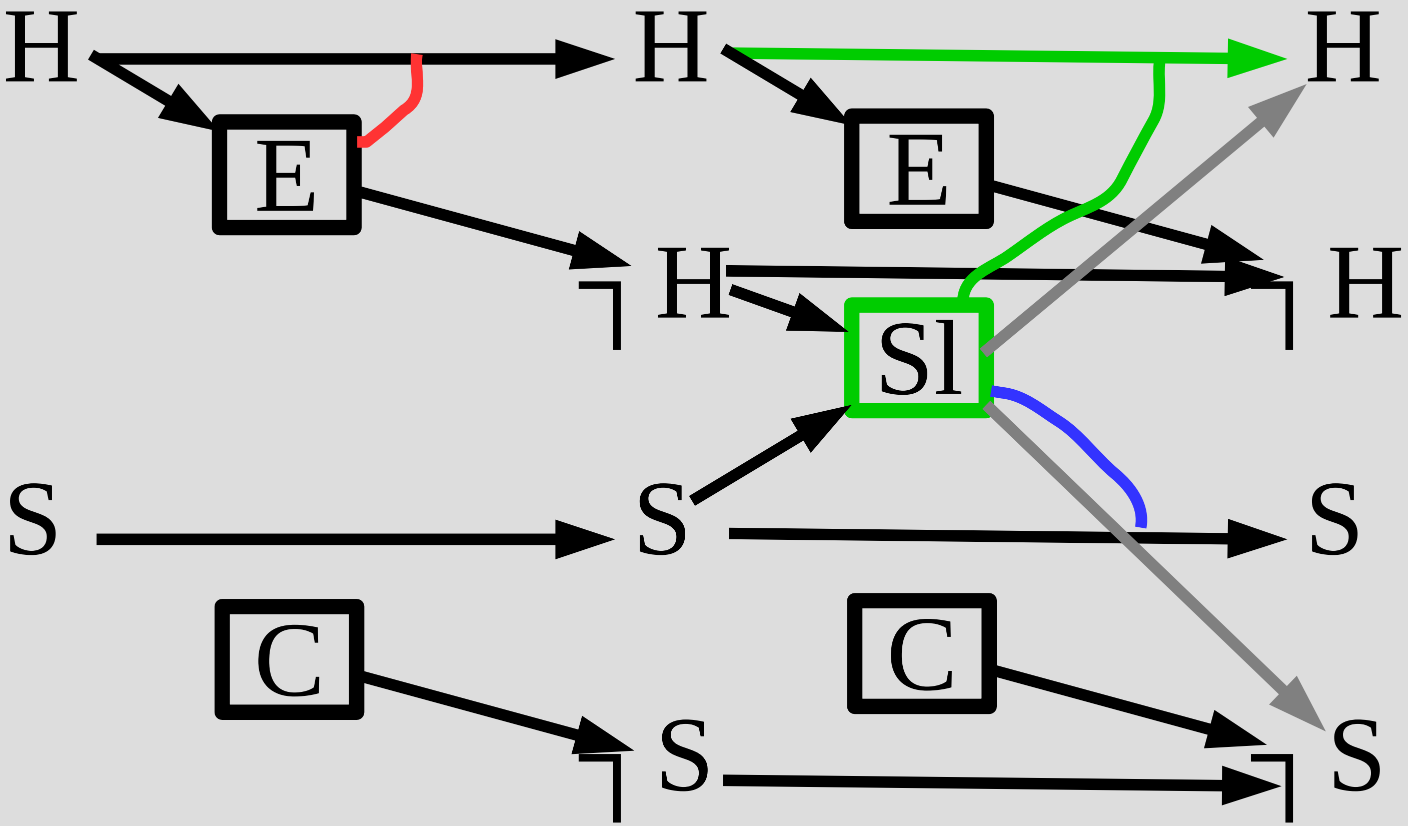
1. $x \in Effect(A1) \wedge \neg x \in Effect(A2)$
2. $x \in Pre(A1) \wedge \neg x \in Effect(A2)$
3. $x \in Pre(A1) \wedge \neg x \in Pre(A2)$



Mutexes: actions

Mutex Action rules:

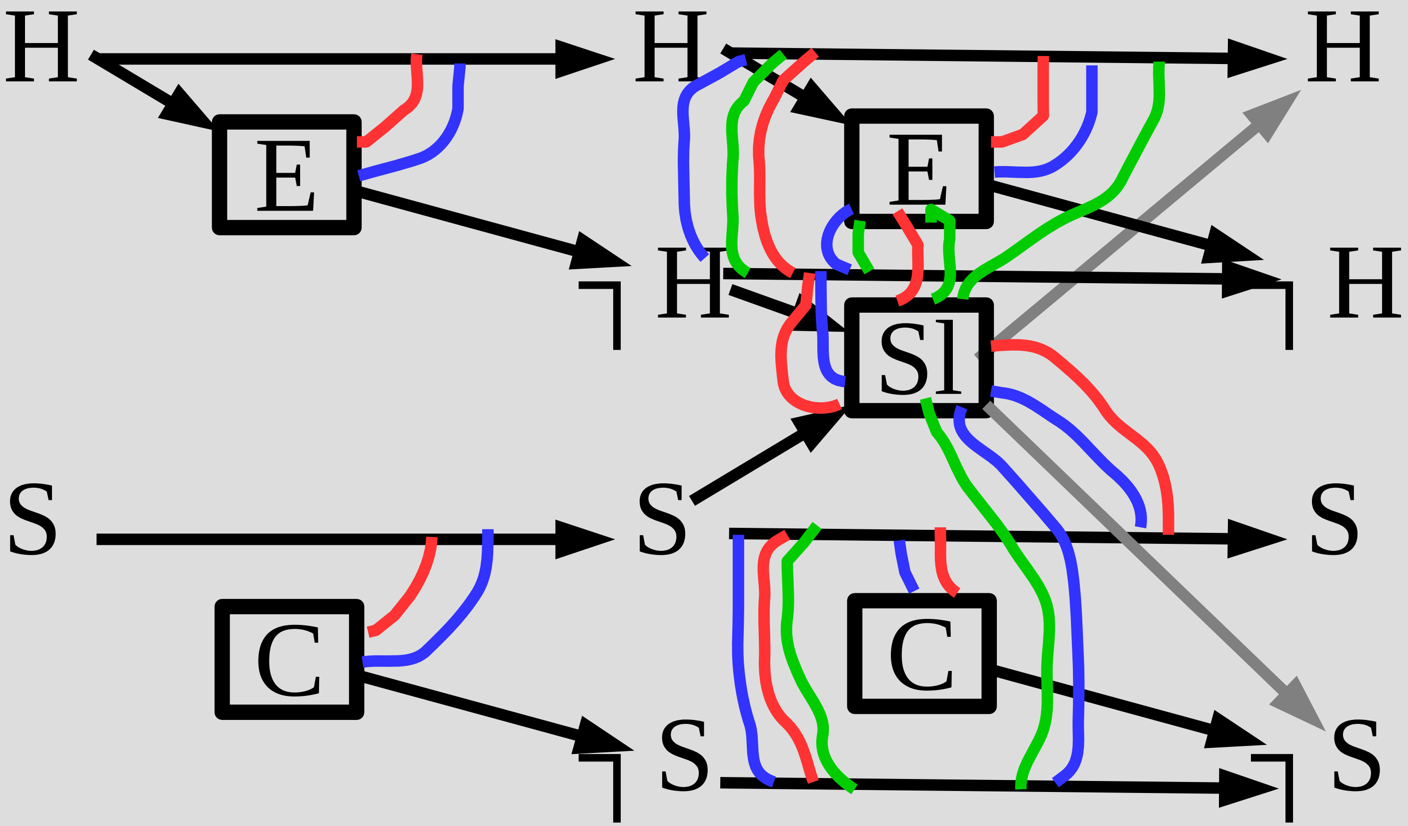
1. $x \in Effect(A1) \wedge \neg x \in Effect(A2)$
2. $x \in Pre(A1) \wedge \neg x \in Effect(A2)$
3. $x \in Pre(A1) \wedge \neg x \in Pre(A2)$



Mutexes: actions

Mutex Action rules:

1. $x \in Effect(A1) \wedge \neg x \in Effect(A2)$
2. $x \in Pre(A1) \wedge \neg x \in Effect(A2)$
3. $x \in Pre(A1) \wedge \neg x \in Pre(A2)$



Mutexes: states

There are 2 rules for states, but unlike action-mutexes they can change across levels

1. Opposite relations are mutexes (x and ¬ x)
2. If there are mutexes between all possible actions that “lead” to a pair of states...

Two ways that “leading” can be in mutex:
1. Actions are in mutex
2. Preconditions of action pair are in mutex

Mutexes: states

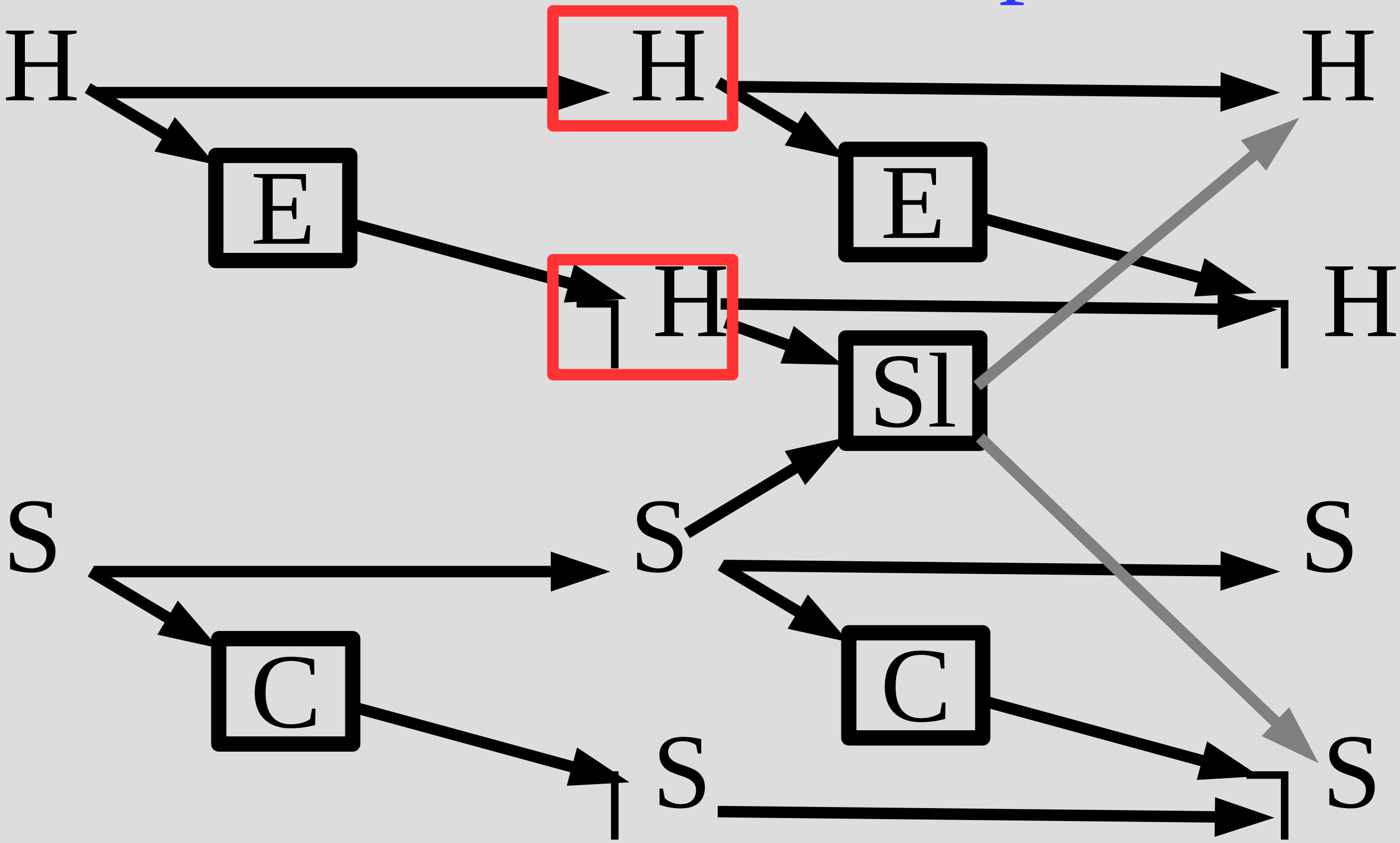
Another way to compute state mutexes:

(1) Add mutexes between all pairs in state
(2) If any pair of actions can lead to this pair of relationships, un-mutex them

Recap:
If any valid pair of actions = no mutex
All ways of reaching invalid = mutex

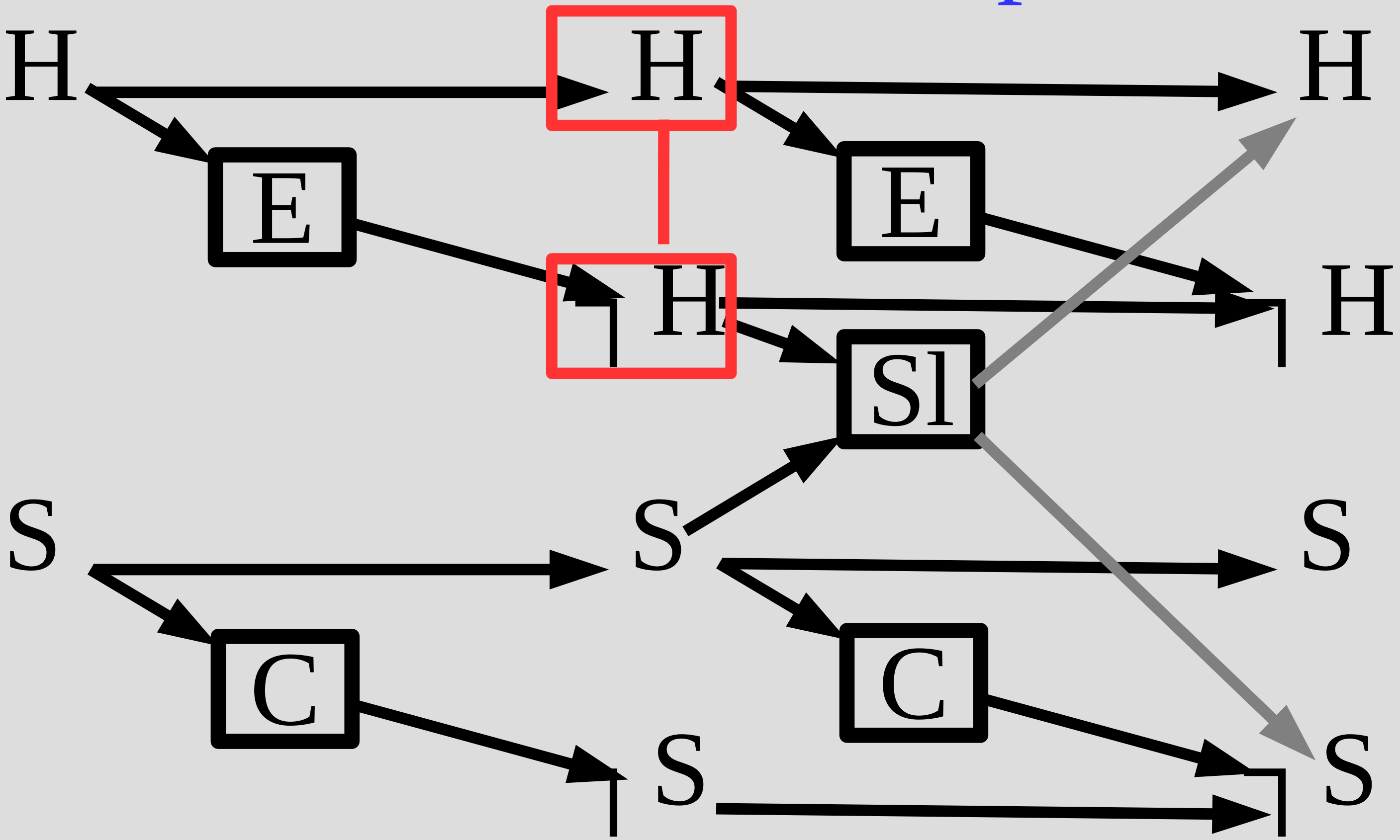
Mutexes: states

1. Opposite relations are mutexes (x and ┐x)
2. If there are mutexes between all possible actions that lead to a pair of states



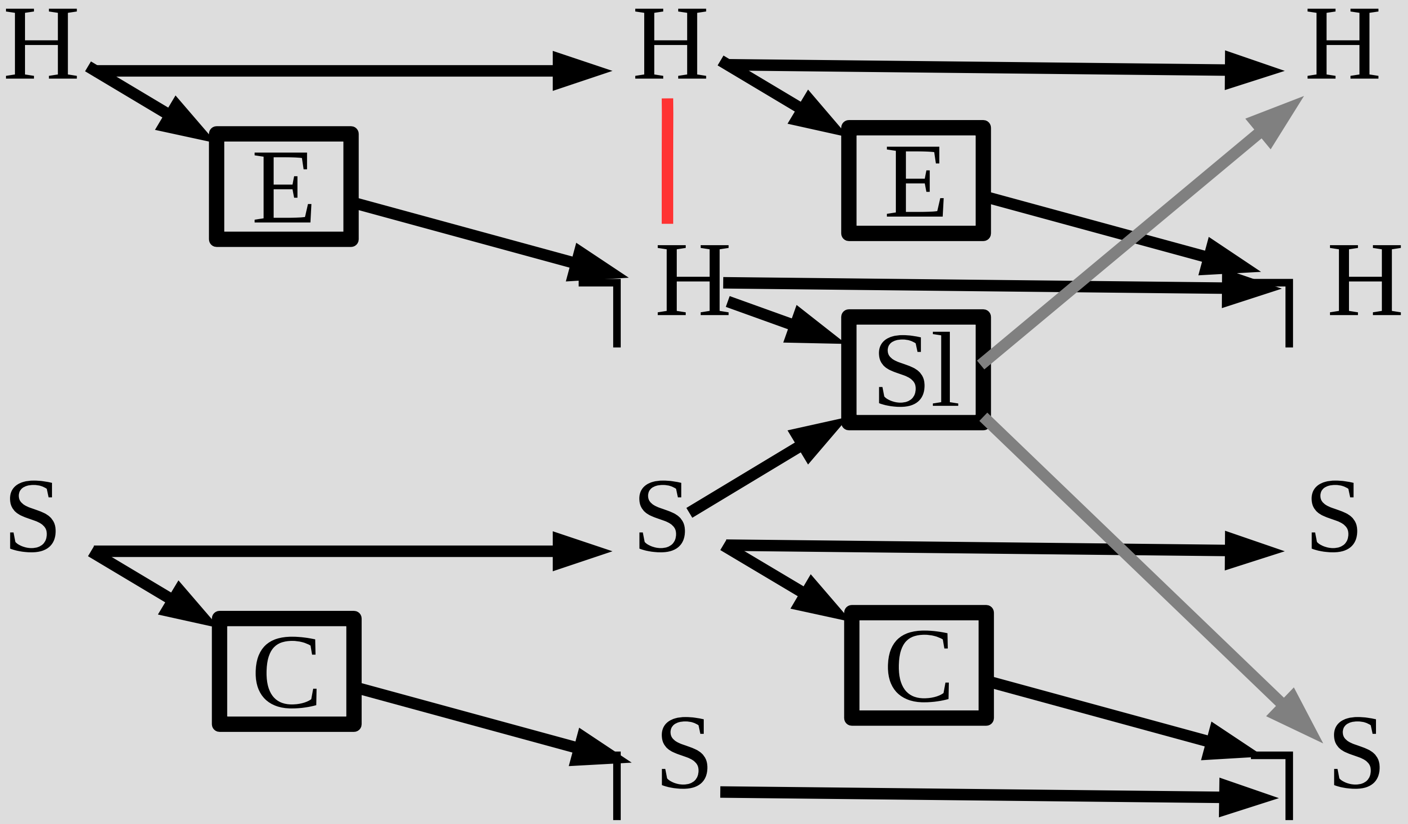
Mutexes: states

1. Opposite relations are mutexes (x and ┐x)
2. If there are mutexes between all possible actions that lead to a pair of states



Mutexes: states

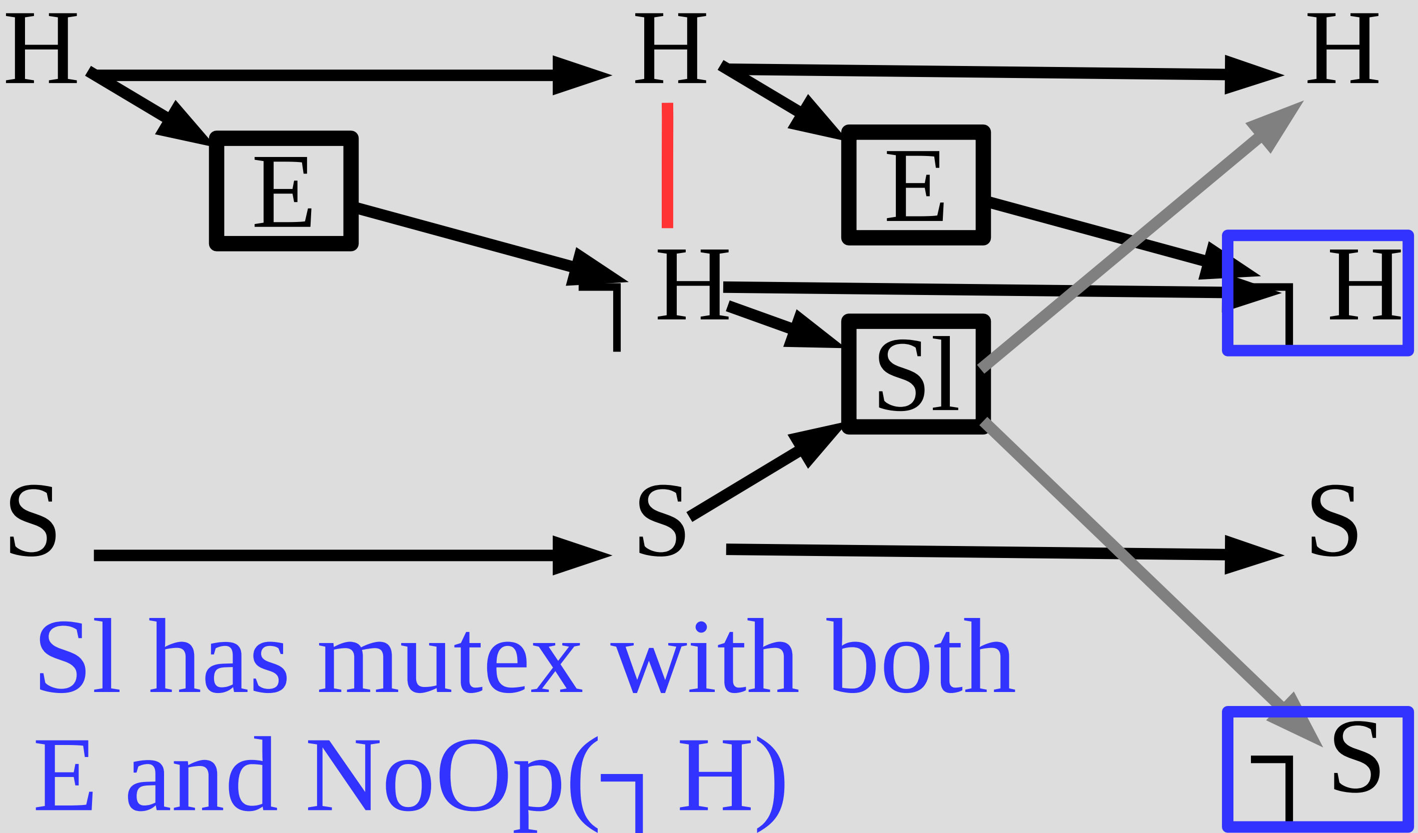
1. Opposite relations are mutexes (x and ¬ x)
2. If there are mutexes between all possible actions that lead to a pair of states



None... but if we remove coffee...

Mutexes: states

1. Opposite relations are mutexes (x and ¬ x)
2. If there are mutexes between all possible actions that lead to a pair of states

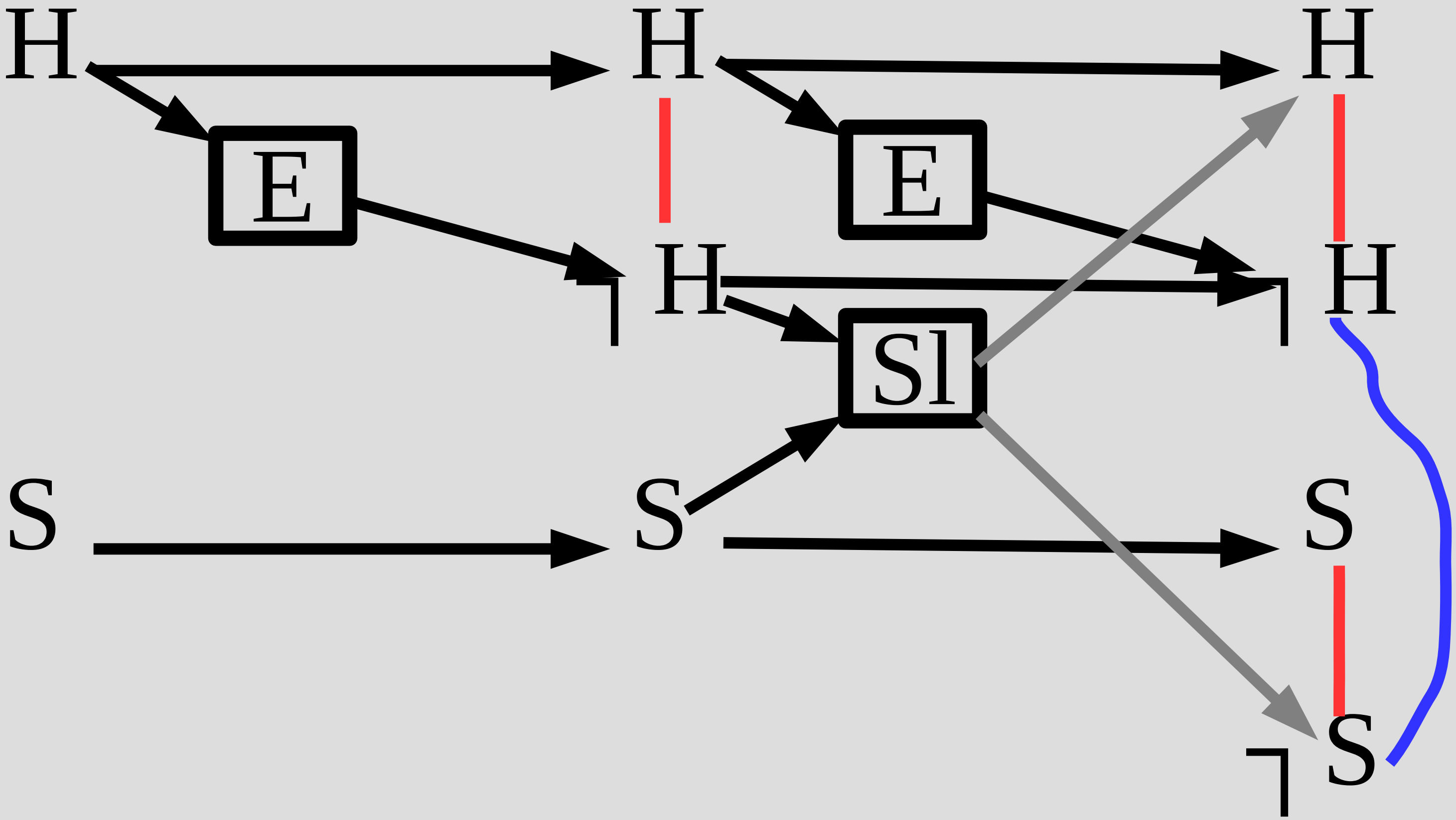


This mutex will be gone on the next level (as you can eat again)

Sl has mutex with both E and NoOp(¬ H)

Mutexes: states

1. Opposite relations are mutexes (x and ┐ x)
2. If there are mutexes between all possible actions that lead to a pair of states



Mutexes: actions

You do it!

Initial: $\neg Money(me) \wedge \neg Smart(me) \wedge \neg Debt(me)$

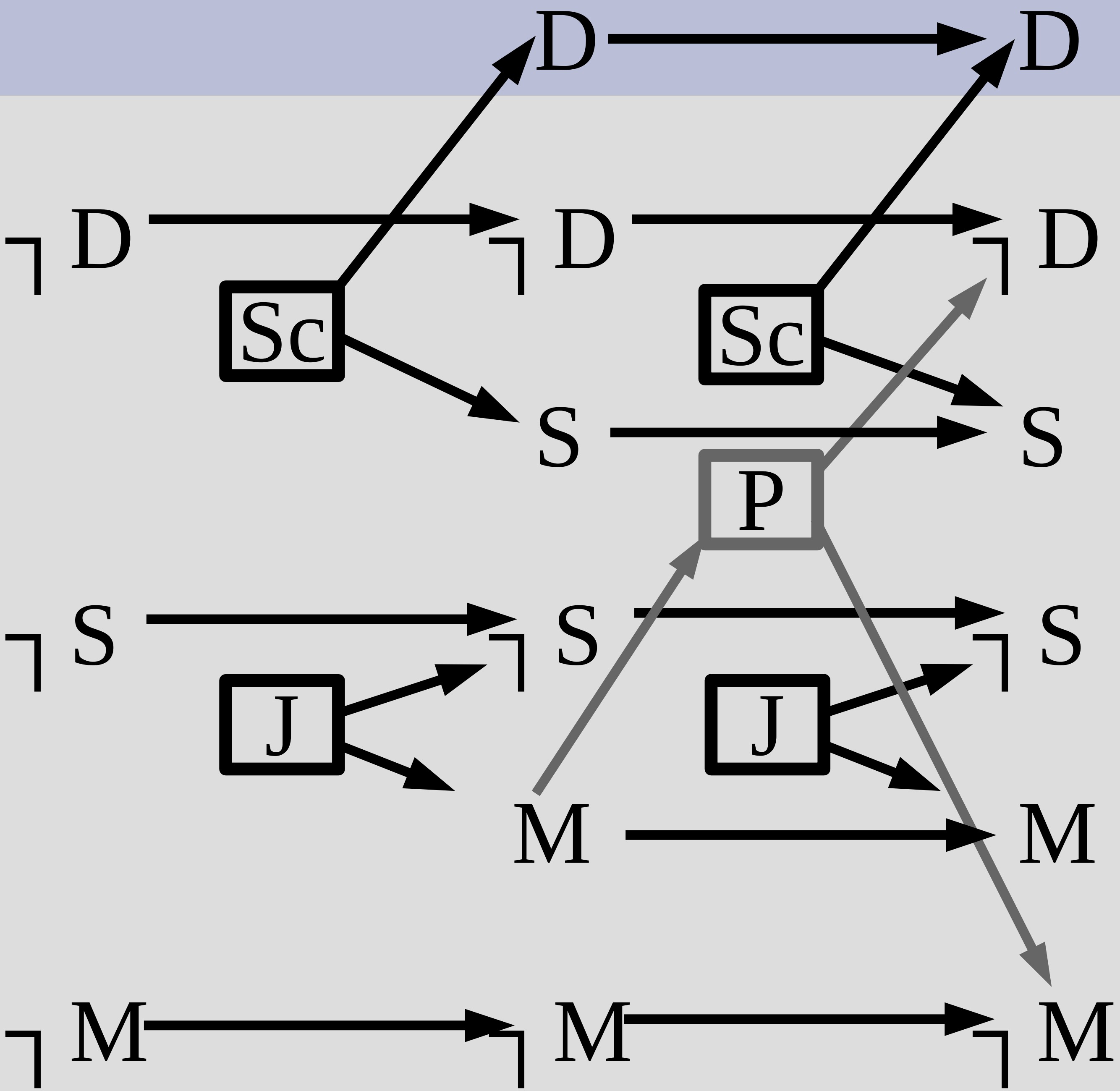
Goal: $(me) \wedge Smart(me) \wedge \neg Debt(me)$

Action($School(x)$,
Precondition: ,
Effect: $Debt(x) \wedge Smart(x)$)

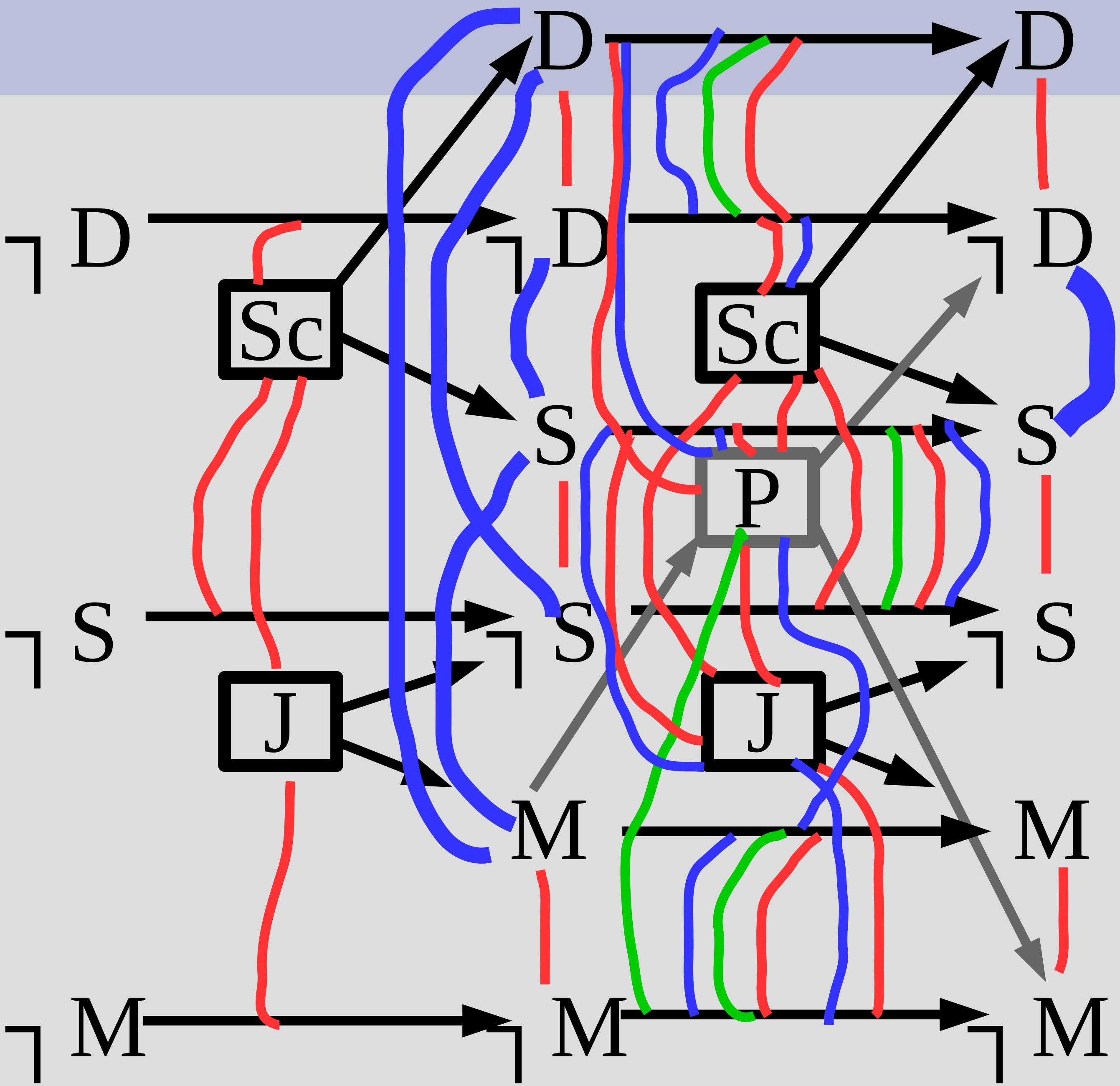
Action($Job(x)$,
Precondition: ,
Effect: $Money(x) \wedge \neg Smart(x)$)

Action($Pay(x)$,
Precondition: $Money(x)$,
Effect: $\neg Money(x) \wedge \neg Debt(x)$)

Mutexes: actions



Mutexes: actions



Non-trivial mutexes:
(SC, P),
(J, P),
(SC, J),
(P,¬ D&M),
(SC,¬ D&¬ S),
(J,¬ M&S)

GraphPlan

GraphPlan can be computed in $O(n(a+l)^2)$,
where n = levels before convergence
a = number of actions
l = number of relations/literals/states
(square is due to needing to check all pairs)

The original planning problem is PSPACE, which is known to be harder than NP

GraphPlan: states

Let's consider this problem:

Initial: $Clean \wedge Garbage \wedge Quiet$

Goal: $Food \wedge \neg Garbage \wedge Present$

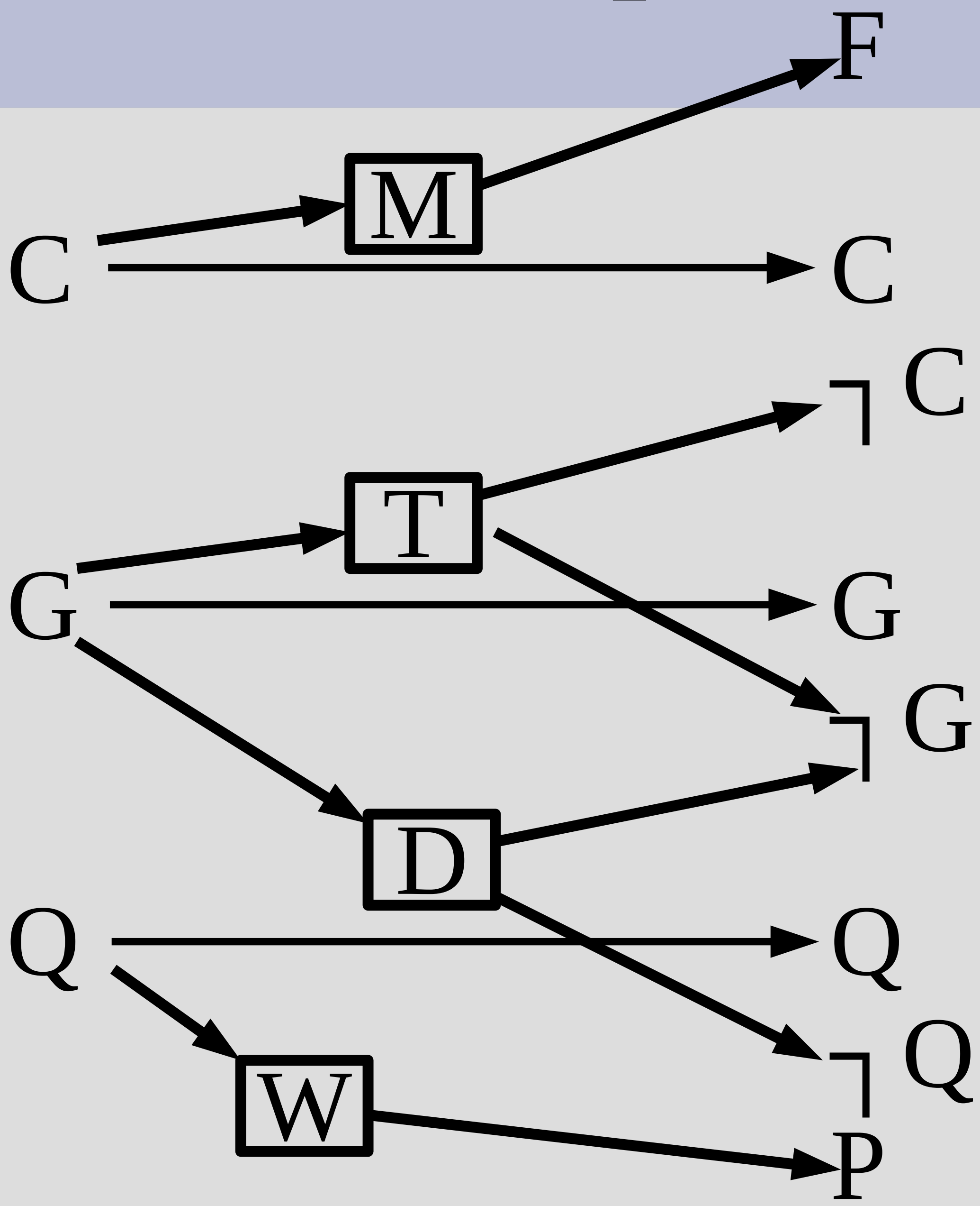
Action: ($MakeFood$,
Precondition: $Clean$,
Effects: $Food$)

Action: ($Takeout$,
Precondition: $Garbage$,
Effects: $\neg Garbage \wedge \neg Clean$)

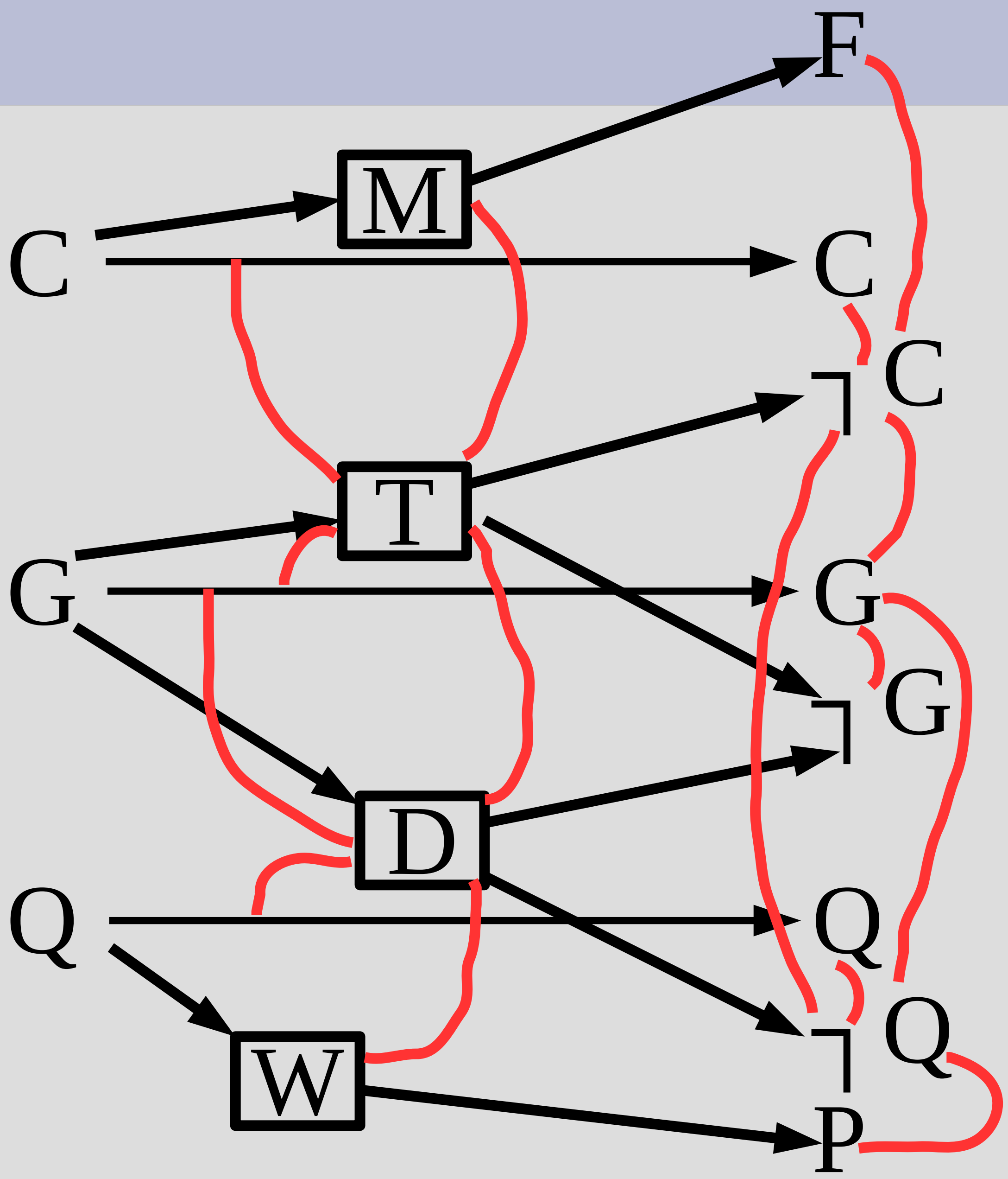
Action: ($Wrap$,
Precondition: $Quiet$,
Effects: $Present$)

Action: ($Dolly$,
Precondition: $Garbage$,
Effects: $\neg Garbage \wedge \neg Quiet$)

GraphPlan: states



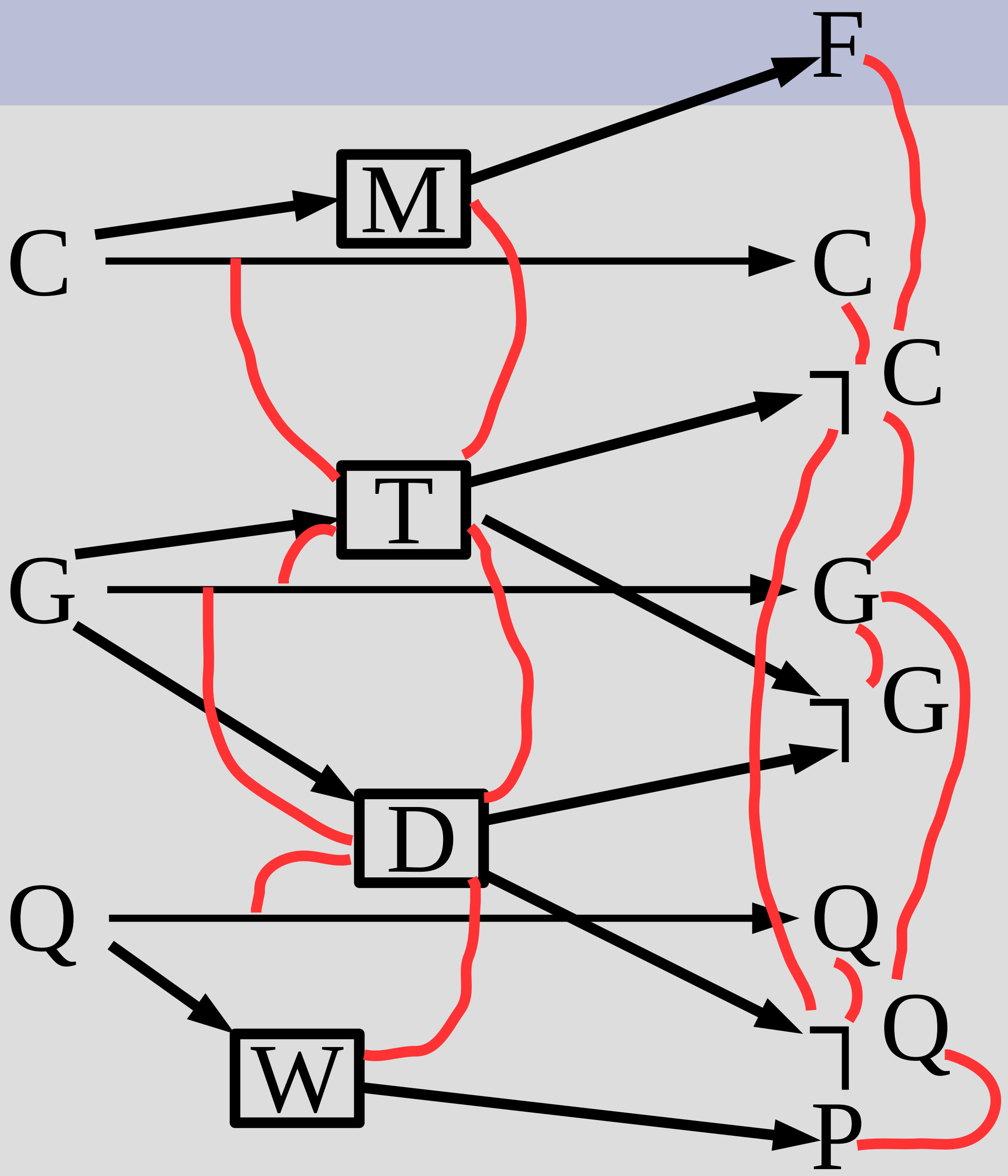
Mutexes



Possible state pairs:

- F, C
- ~~F, ¬C~~
- F, G
- F, ¬G
- F, Q
- F, ¬Q
- F, P
- ~~C, ¬C~~
- C, G
- C, ¬G
- C, Q
- C, ¬Q
- C, P
- ~~¬C, G~~
- ¬C, ¬G
- ¬C, Q
- ~~¬C, ¬Q~~
- ¬C, P
- … (more)

Mutexes

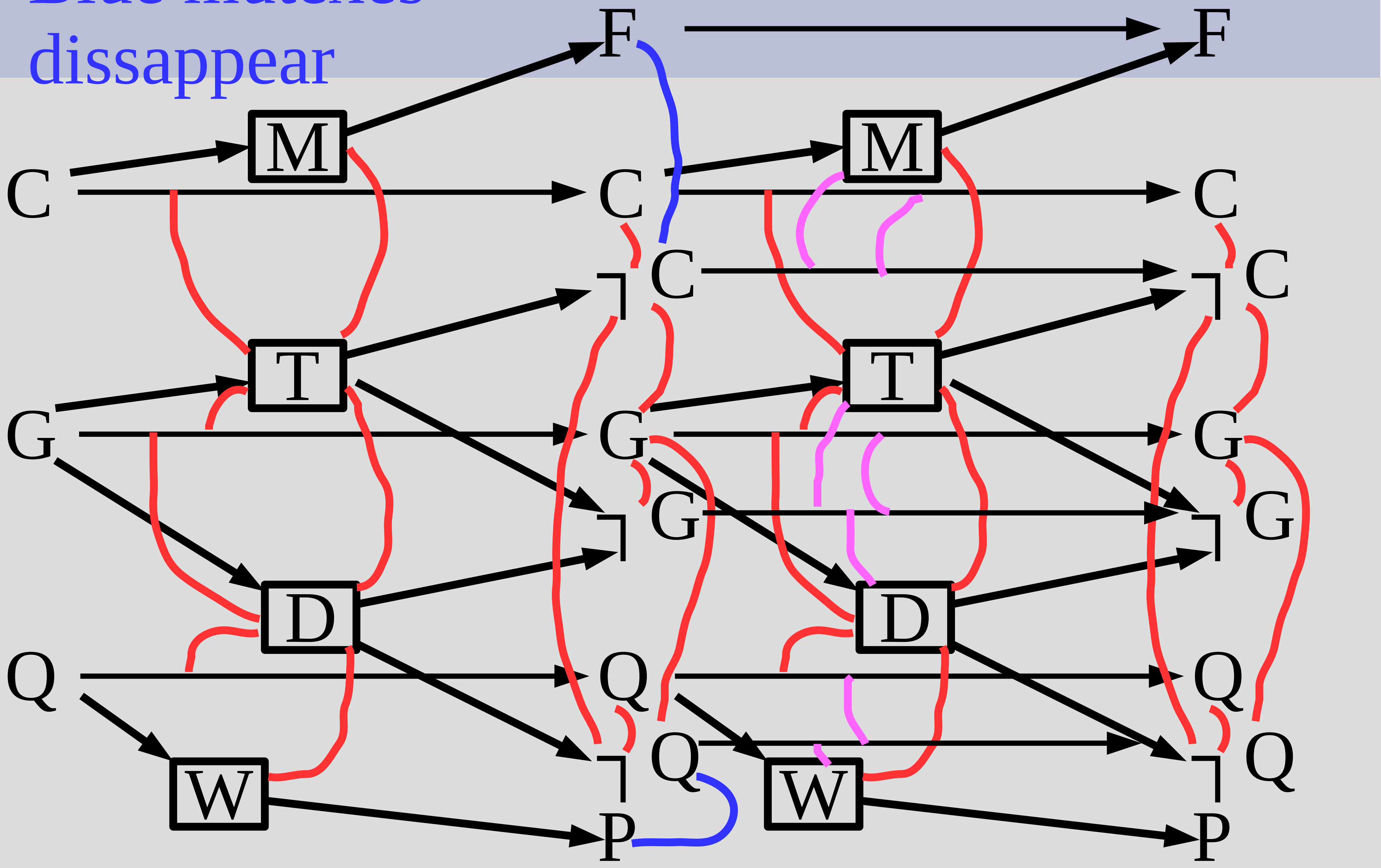


Make
one
more
level
here!

Mutexes

Blue mutexes dissappear

Pink = new mutex



GraphPlan as heuristic

GraphPlan is optimistic, so if any pair of goal states are in mutex, the goal is impossible

3 basic ways to use GraphPlan as heuristic:

(1) Maximum level of all goals
(2) Sum of level of all goals (not admissible)
(3) Level where no pair of goals is in mutex

(1) and (2) do not require any mutexes, but are less accurate (quick 'n' dirty)

GraphPlan as heuristic

For heuristics (1) and (2), we relax as such:

1. Multiple actions per step, so can only take fewer steps to reach same result
2. Never remove any states, so the number of possible states only increases

This is a valid simplification of the problem, but it is often too simplistic directly

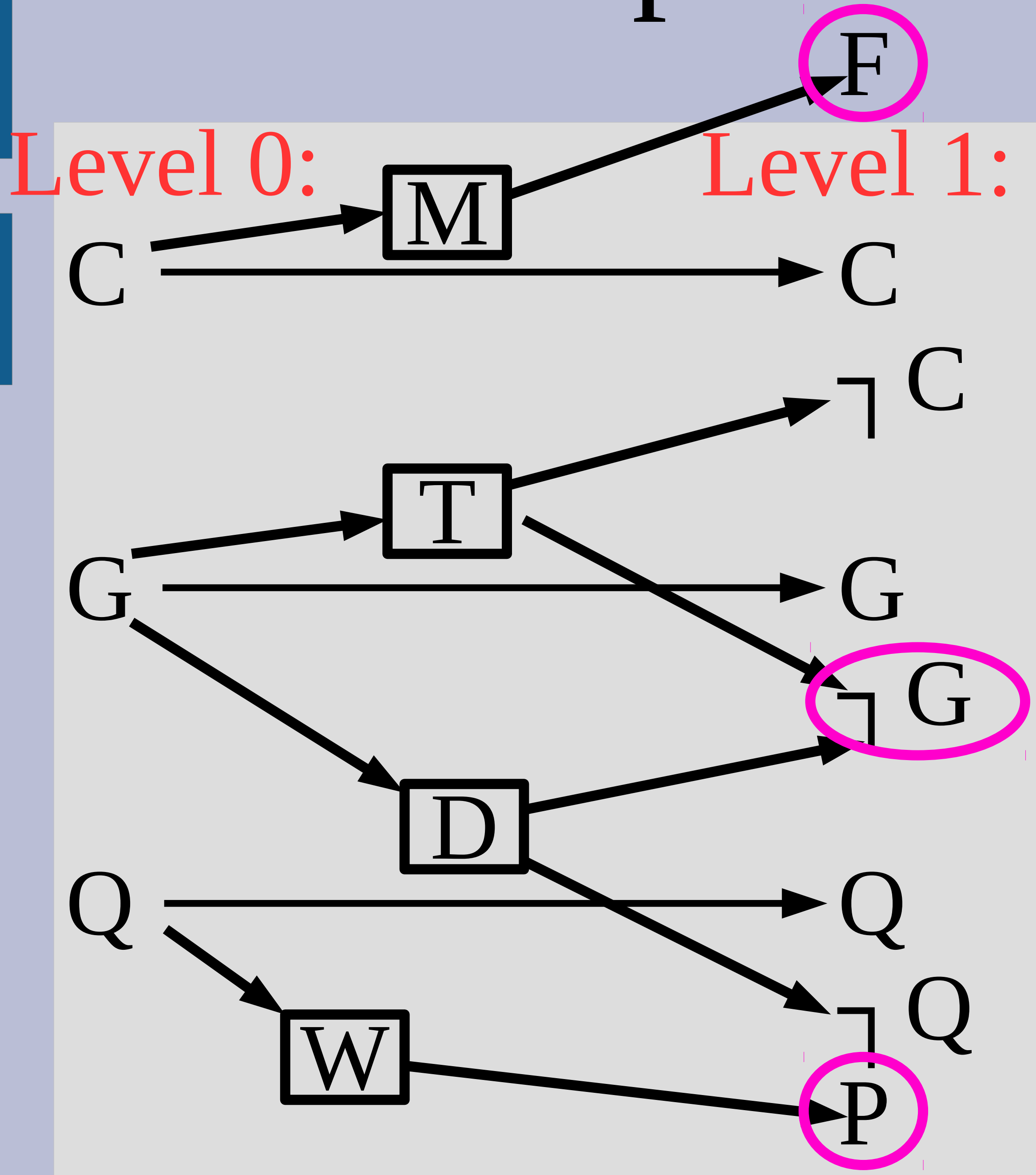
GraphPlan as heuristic

Heuristic (1) directly uses this relaxation and finds the first time when all 3 goals appear at a state level

(2) tries to sum the levels of each individual first appearance, which is not admissible (but works well if they are independent parts)

Our problem: goal={Food, ¬ Garbage, Present}
First appearance: F=1, ¬ G=1, P=1

GraphPlan: states



Heuristic (1):
Max(1,1,1) = 1

Heuristic (2):
1+1+1=3

GraphPlan as heuristic

Often the problem is too trivial with just those two simplifications

So we add in mutexes to keep track of invalid pairs of states/actions

This is still a simplification, as only impossible state/action pairs in the original problem are in mutex in the relaxation

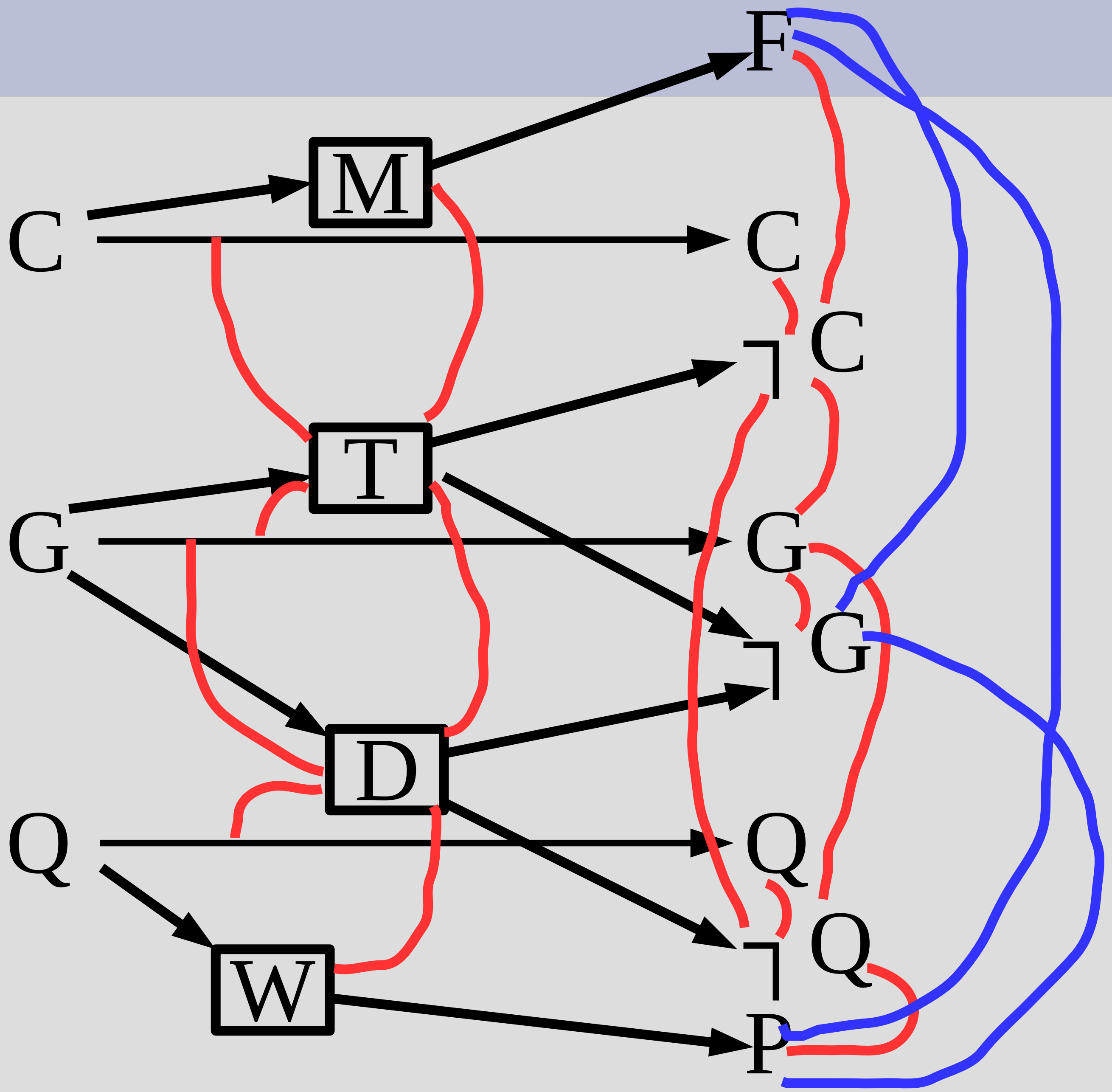
GraphPlan as heuristic

Heuristic (3) looks to find the first time none of the goal pairs are in mutex

For our problem, the goal states are:
(Food, ¬ Garbage, Present)

So all pairs that need to have no mutex:
(F, ¬ G), (F, P), (¬ G, P)

Mutexes



None of the pairs are in mutex at level 1

This is our heuristic estimate

Finding a solution

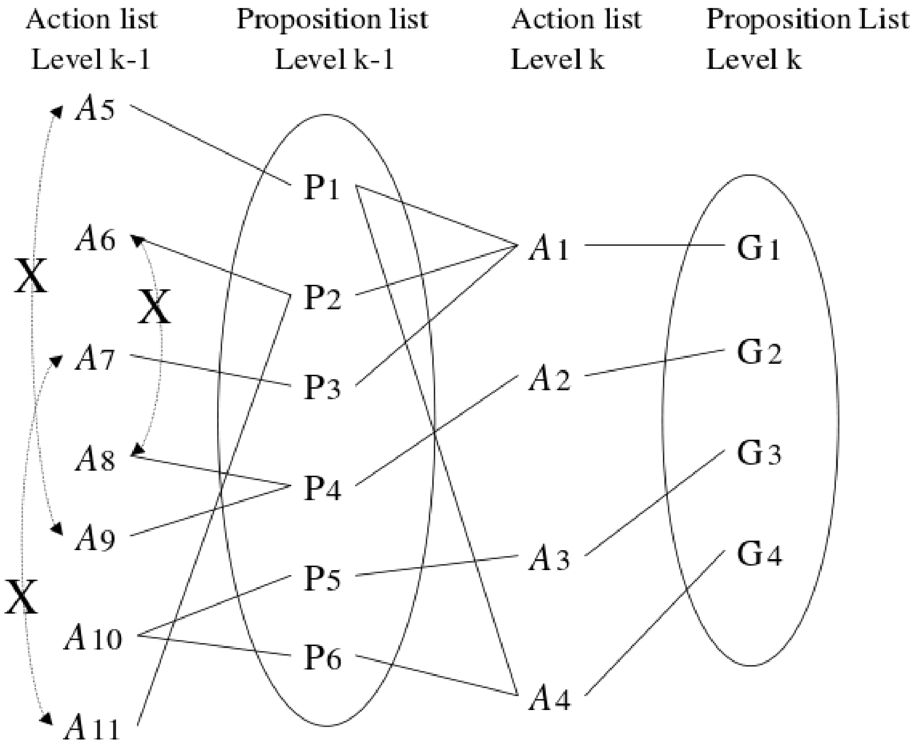
GraphPlan can also be used to find a solution:
(1) Converting to a Constraint Sat. Problem
(2) Backwards search

Both of these ways can be run once GraphPlan has all goal pairs not in mutex (or converges)

Additionally, you might need to extend it out a few more levels further to find a solution (as GraphPlan underestimates)

GraphPlan as CSP

Variables = states, Domains = actions out of
Constraints = mutexes & preconditions



(a) Planning Graph

Variables: $G_1, \cdots, G_4, P_1 \cdots P_6$

Domains: $G_1:\{A_1\}, G_2:\{A_2\} G_3:\{A_3\} G_4:\{A_4\}$
$P_1:\{A_5\} P_2:\{A_6, A_{11}\} P_3:\{A_7\} P_4:\{A_8, A_9\}$
$P_5:\{A_{10}\} P_6:\{A_{10}\}$

Constraints (normal): $P_1 = A_5 \Rightarrow P_4 \neq A_9$
$P_2 = A_6 \Rightarrow P_4 \neq A_8$
$P_2 = A_{11} \Rightarrow P_3 \neq A_7$

Constraints (Activity): $G_1 = A_1 \Rightarrow Active\{P_1, P_2, P_3\}$
$G_2 = A_2 \Rightarrow Active\{P_4\}$
$G_3 = A_3 \Rightarrow Active\{P_5\}$
$G_4 = A_4 \Rightarrow Active\{P_1, P_6\}$

Init State: $Active\{G_1, G_2, G_3, G_4\}$

(b) DCSP

from Do & Kambhampati

Finding a solution

For backward search, attempt to find arrows back to the initial state(without conflict/mutex)

This backwards search is similar to backward chaining in first-order logic (depth first search)

If this fails to find a solution, mark this level and all the goals not satisfied as: (level, goals)

(level, goals) stops changing, no solution

Graph Plan

Remember this...

Initial: $\neg Money(me) \wedge \neg Smart(me) \wedge \neg Debt(me)$

Goal: $(me) \wedge Smart(me) \wedge \neg Debt(me)$

Action($School(x)$,

Precondition: ,

Effect: $Debt(x) \wedge Smart(x)$)

Action($Job(x)$,

Precondition: ,

Effect: $Money(x) \wedge \neg Smart(x)$)

Action($Pay(x)$,

Precondition: $Money(x)$,

Effect: $\neg Money(x) \wedge \neg Debt(x)$)

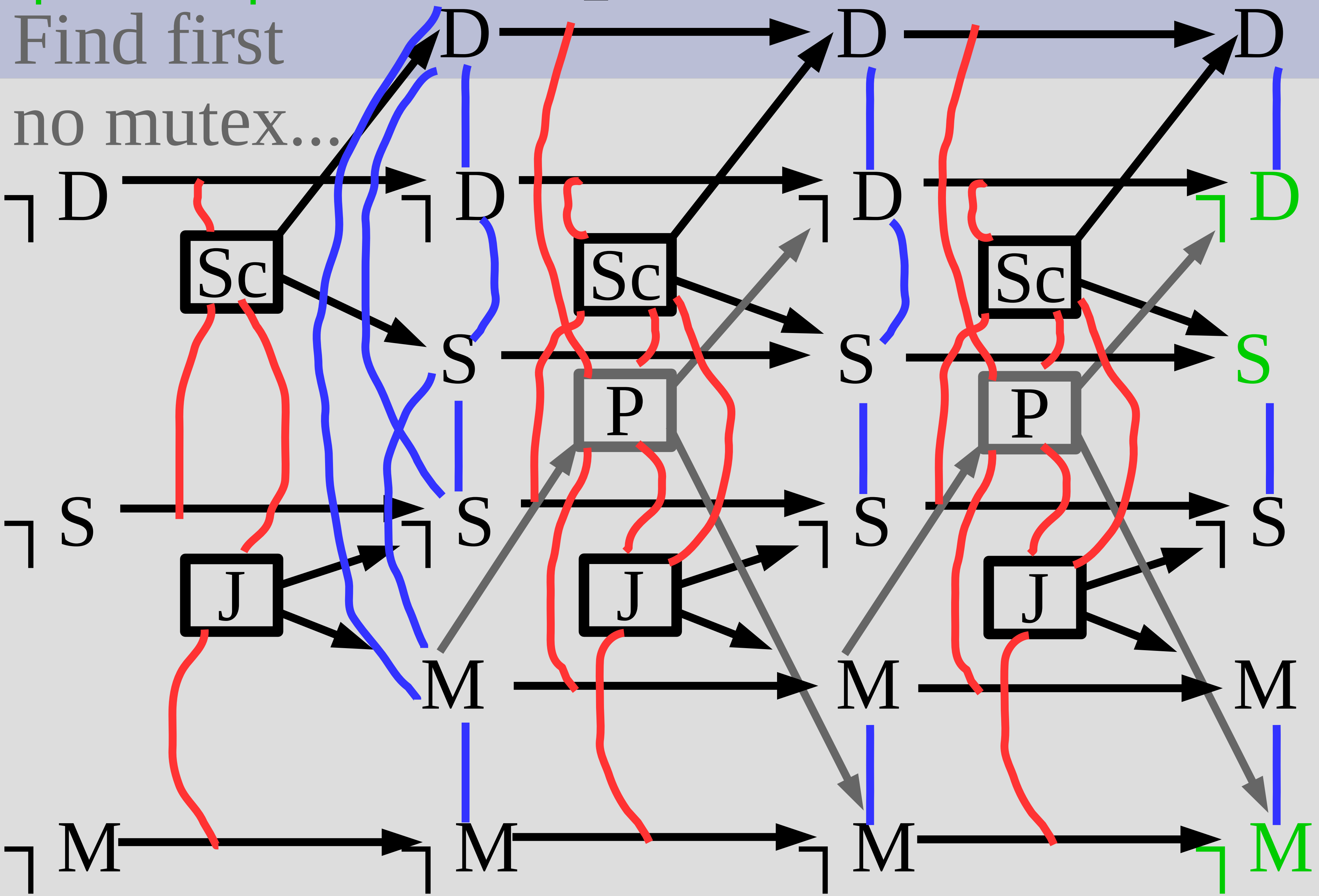
Graph Plan

Ask:

¬D^S^¬M

Find first

no mutex...



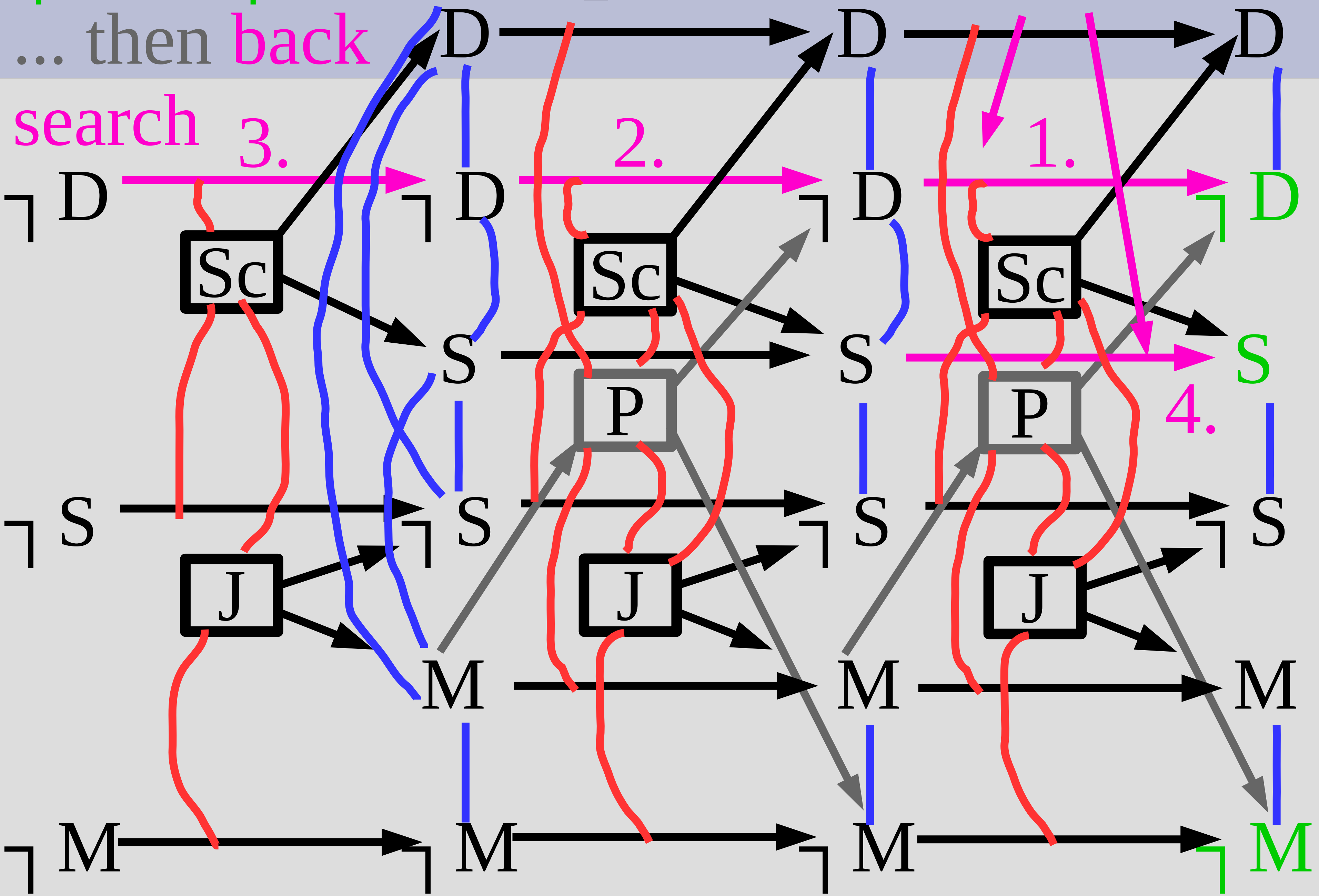
Ask:

┐D^S^┐M

... then back search

Graph Plan

Error! Actions 1&4 in mutex



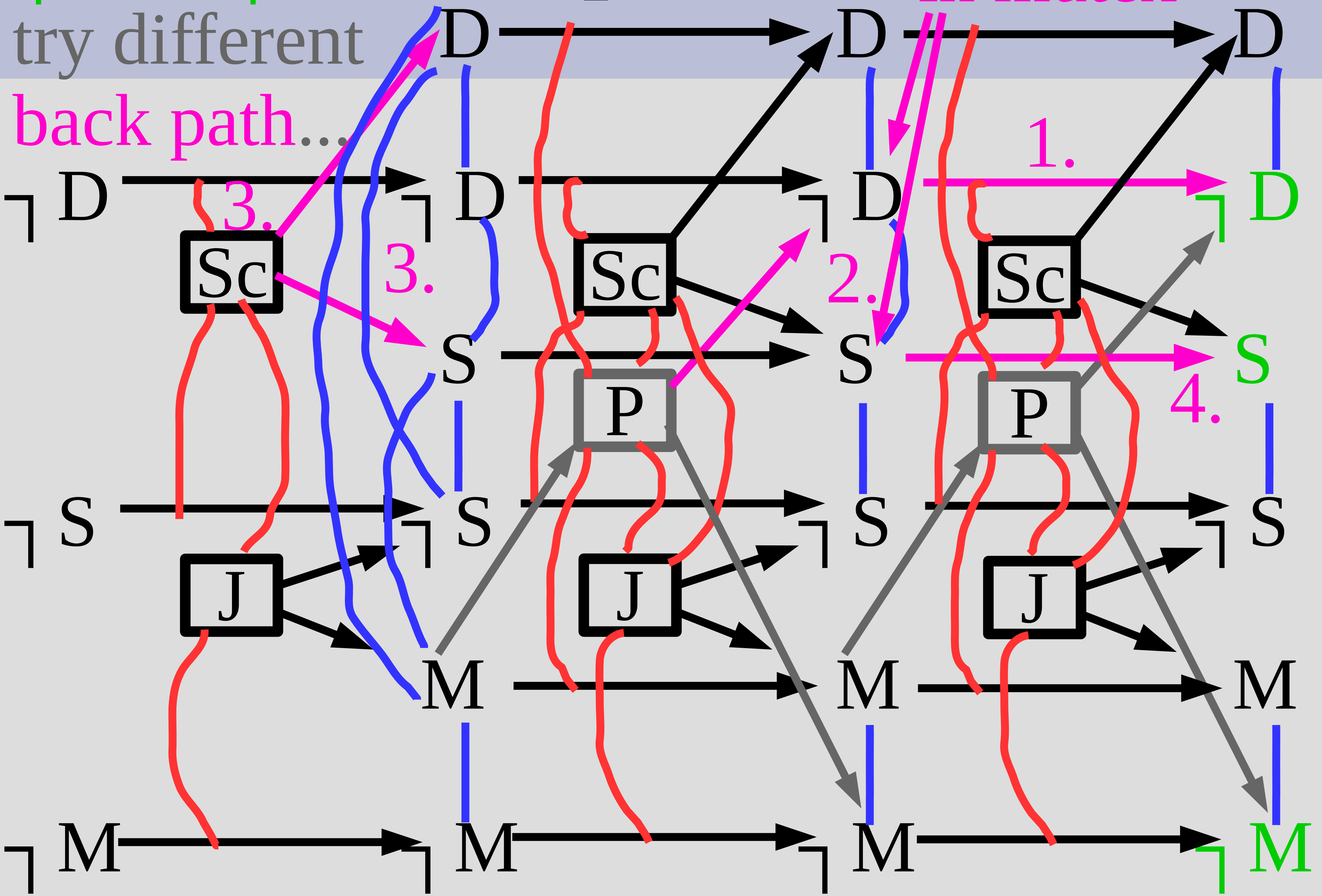
Graph Plan

Ask:

¬D^S^¬M

try different back path...

Error states in mutex



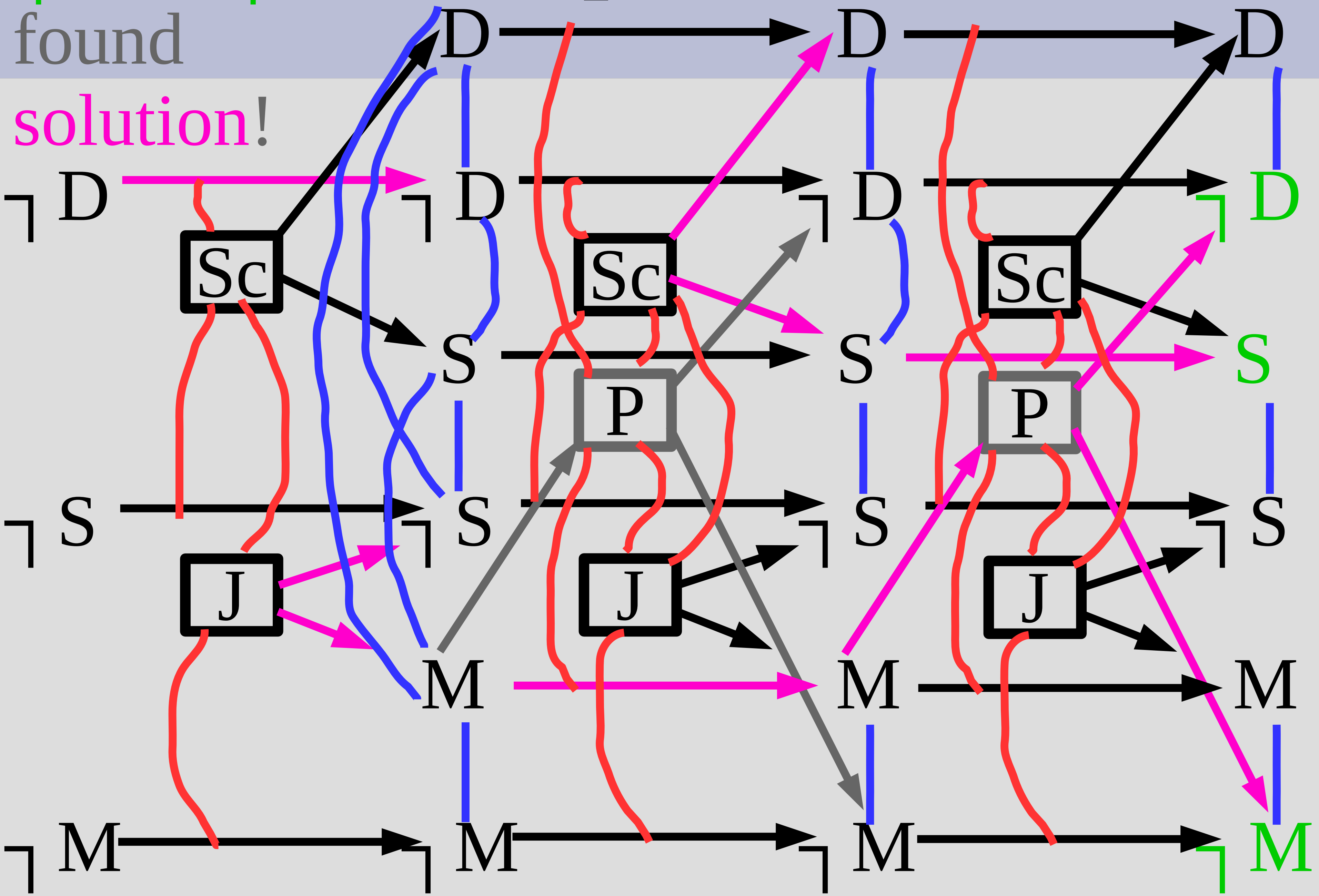
Graph Plan

Ask:

¬D^S^¬M

found

solution!



Finding a solution

Formally, the algorithm is:

```
graph = initial
noGoods = empty table (hash)
for level = 0 to infinity
    if all goal pairs not in mutex
        solution = DFS with noGoods
        if success, return paths
    if graph & noGoods converged, return fail
    graaph = expand graph
```

Mutexes

You try it!