AI Planning.
Planning-graph techniques.
Heuristic planning.

Eva Onaindia

Universitat Politècnica de València

Acknowledgements

Most of the slides used in this course are taken or are modifications from Dana Nau’s lecture slides for the textbook *Automated Planning,* licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License:

http://creativecommons.org/licenses/by-nc-sa/2.0/
http://creativecommons.org/licenses/by/3.0/es/

I would like to gratefully acknowledge Dana Nau’s contributions and thank him for generously permitting me to use aspects of his presentation material.

Outline

- History
- Motivation
- Reachability analysis
- The planning graph
  - Example
  - Mutual exclusion
  - Example (continued)
- The Graphplan algorithm
- Solution extraction
- Heuristics derived from planning graphs
- Discussion

History

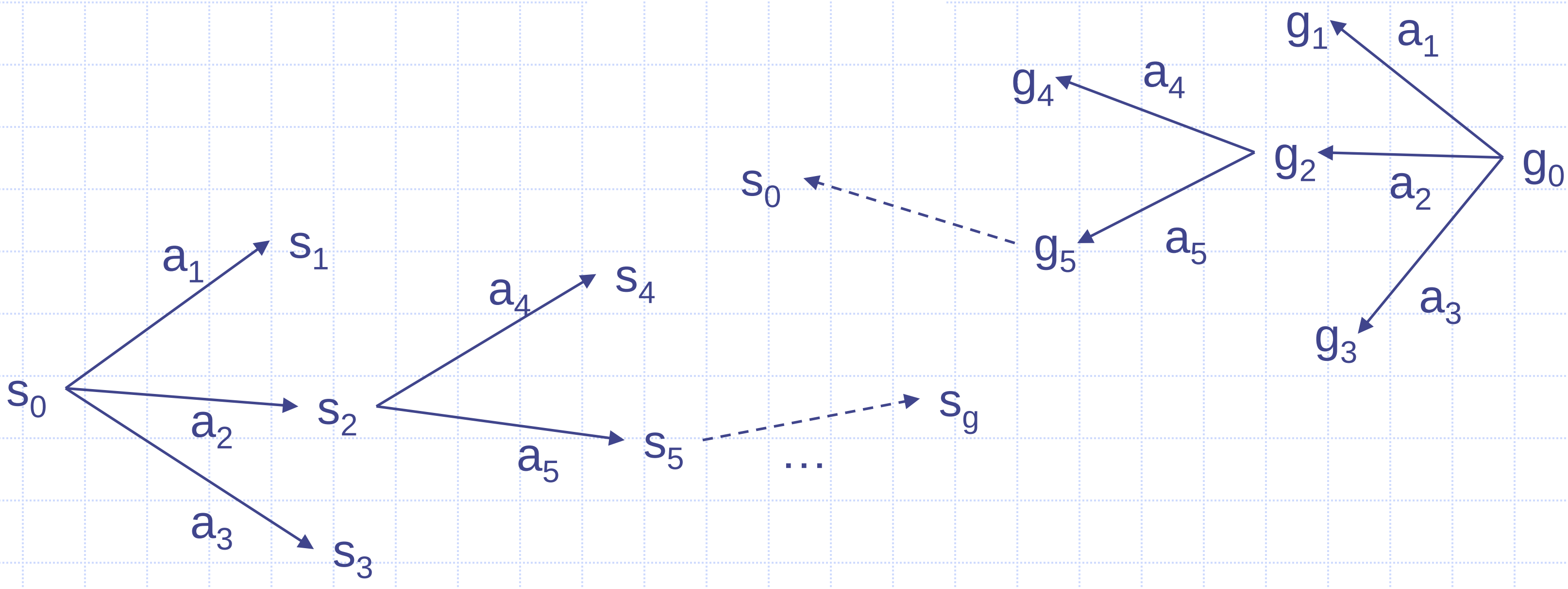
- Before Graphplan came out, most planning researchers were working on PSP-like planners
  - POP, SNLP, UCPOP, etc.

- Graphplan caused a sensation because it was so much faster

- Many subsequent planning systems have used ideas from it
  - IPP, STAN, GraphHTN, SGP, Blackbox, Medic, TGP, LPG
  - Many of them are much faster than the original Graphplan

Motivation

- A big source of inefficiency in search algorithms is the *branching factor*
  - the number of children of each node
- e.g., a backward search may try lots of actions that can't be reached from the initial state
- and forward search may try lots of actions that do not reach the goal state



Motivation

- One way to reduce branching factor:
- First create a *relaxed problem*
  - Remove some restrictions of the original problem
    - Want the relaxed problem to be easy to solve (polynomial time)
  - The solutions to the relaxed problem will include all solutions to the original problem
    - actions that occur in the solutions to the relaxed problem will also occur in all solutions to the original problem
- Then do a modified version of the original search
  - Restrict its search space to include only those actions that occur in solutions to the relaxed problem

Motivation

- State-space planners provide a plan as a sequence of actions
- Plan-space planning synthesize a plan as a partially ordered set of actions; any sequence that meets the constraints of the partial order is a valid plan
- Planning-graph approaches take a middle ground. Their output is a sequence of sets of actions, e.g. <{a1,a2},{a3,a4},{a5,a6,a7}>
- Plan-space planning: *least-commitment principle*

  Planning-graph approach: *strong commitment* while planning, actions are considered fully instantiated and at specific time steps; *least-commitment* with respect to the order of the actions at a time step
- Planning-graph approaches rely on the idea of ***relaxation of the reachability analysis***

Reachability analysis

- Analysis performed by planners to make sure the goals of the problem are reachable

- State reachability:
  - given a set **A** of actions, a state **s** is reachable from some initial state **s0** if there is a sequence of actions in **A** that defines a path from **s0** to **s**
  - Reachability analysis consists of analyzing which states can be reached from **s0** in some number of steps and how to reach them
  - Reachability can be computed *exactly* through a *reachability tree* => it cannot be computed in a tractable way
  - Reachability can be *approximated* through a *planning graph* => relaxation of the reachability analysis

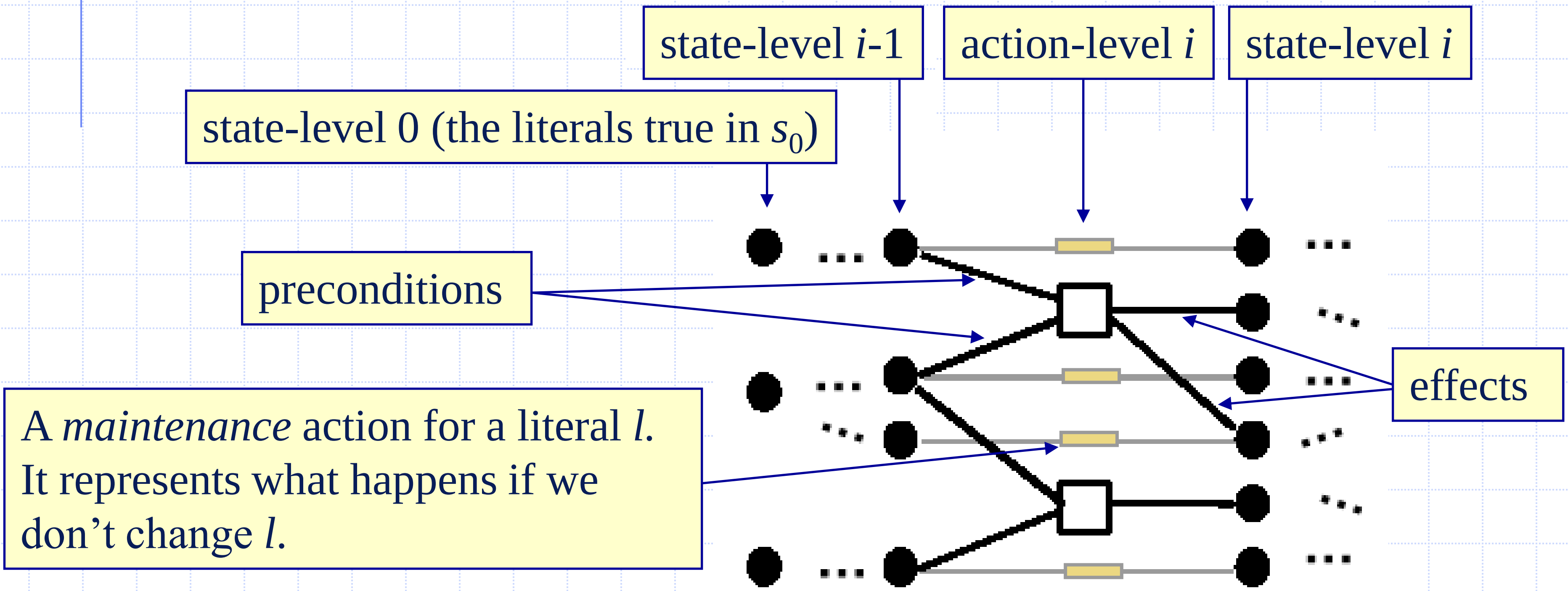
The planning graph (I)

- A layered graph
- Two kinds of layers alternate:
  - Literals (propositions) (shown with circles)
  - Actions (shown with squares)
- Every two layers correspond to a discrete time (time step)
- No variables as in action schemas

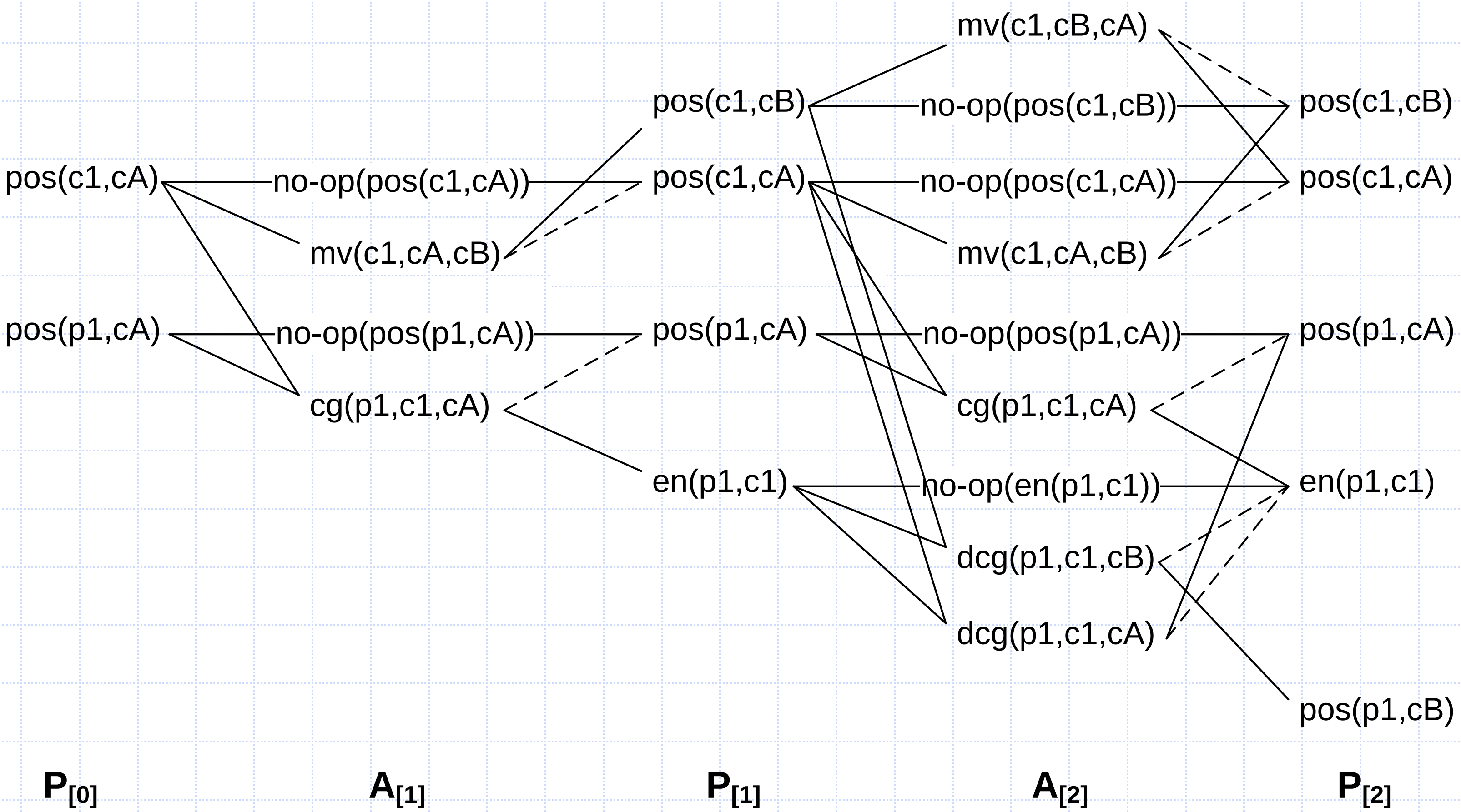
- The first layer is a literal layer which shows all the literals that are true in the initial state
- Every action has a link from each of its preconditions and a link to each of its effects.
- Straight lines between two literals at consecutive literal levels denote *NoOp*

The planning graph (II)

- Search space for a relaxed version of the planning problem
- Alternating layers of ground literals and actions
  - Nodes at action-level *i*: actions that might be possible to execute at time *i*
  - Nodes at state-level *i:* literals that might possibly be true at time *i*
  - Edges: preconditions and effects

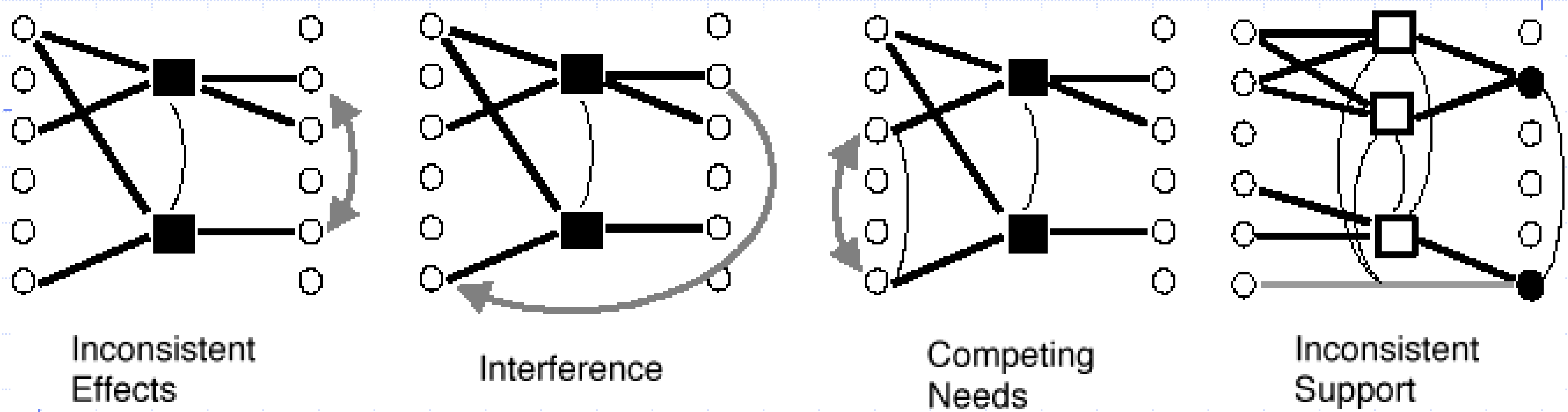


Example



mv(c1,cB,cA)
pos(c1,cB)
no-op(pos(c1,cB))
pos(c1,cB)
pos(c1,cA)
no-op(pos(c1,cA))
pos(c1,cA)
no-op(pos(c1,cA))
pos(c1,cA)
mv(c1,cA,cB)
mv(c1,cA,cB)
pos(p1,cA)
no-op(pos(p1,cA))
pos(p1,cA)
no-op(pos(p1,cA))
pos(p1,cA)
cg(p1,c1,cA)
cg(p1,c1,cA)
en(p1,c1)
no-op(en(p1,c1))
en(p1,c1)
dcg(p1,c1,cB)
dcg(p1,c1,cA)
pos(p1,cB)
P[0]
A[1]
P[1]
A[2]
P[2]

Mutual Exclusion



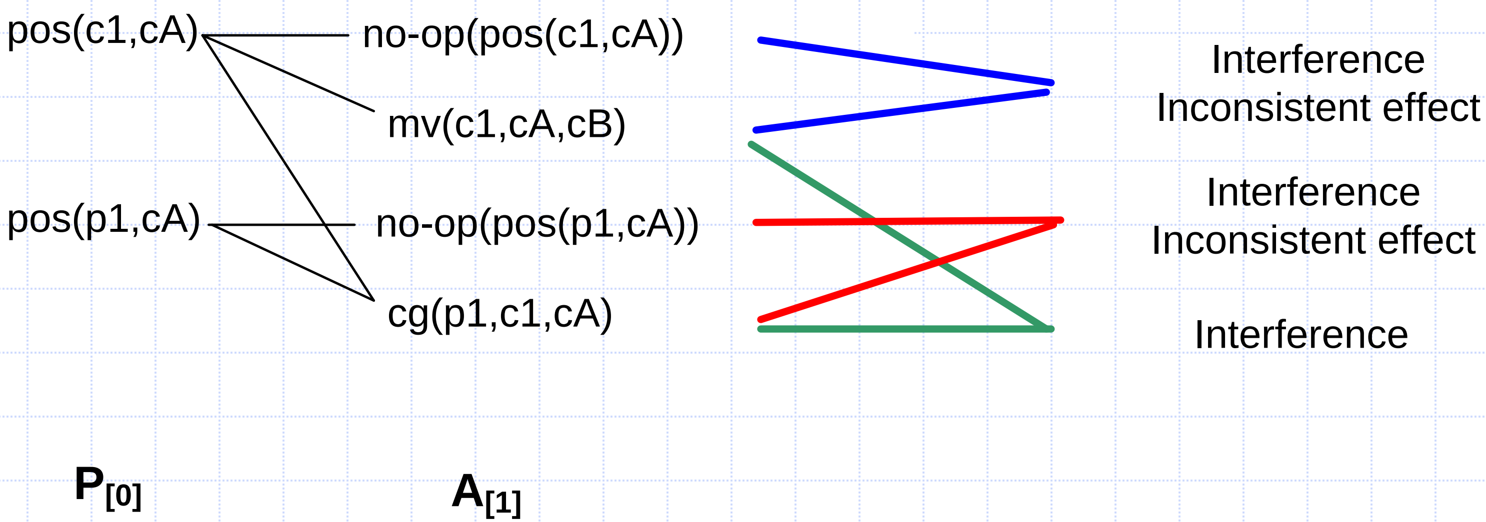
- Two actions at the same action-level are mutex if
  - *Inconsistent effects:* an effect of one negates an effect of the other
  - *Interference:* one deletes a precondition of the other
  - *Competing needs:* **they have mutually exclusive preconditions**
- Otherwise they don't interfere with each other
  - Both may appear in a solution plan
- Two literals at the same state-level are mutex if
  - *Inconsistent support:* one is the negation of the other, **or all ways of achieving them are pairwise mutex**

Recursive propagation of mutexes

Example (level A[1])

A[1]

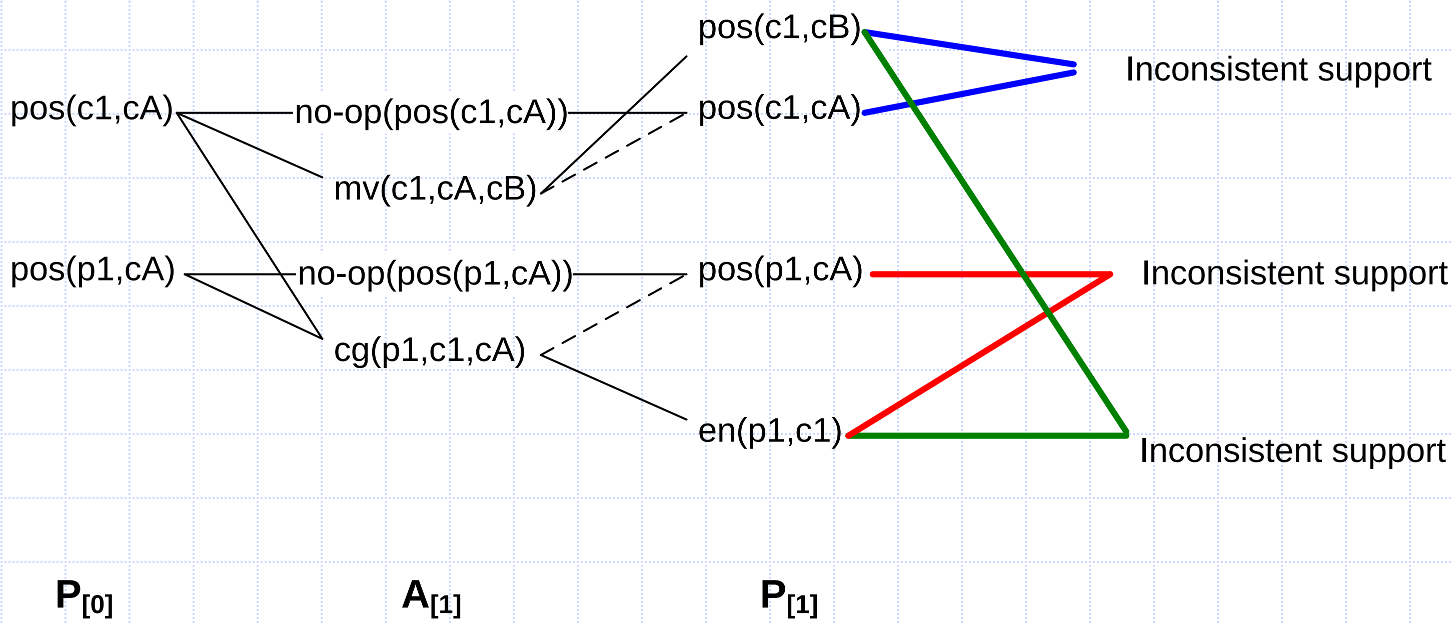
mv(c1,cA,CB) x {cg(p1,c1,cA), no-op(pos(c1,cA))}
cg(p1,c1,cA) x {no-op(pos(p1,cA)), mv(c1,cA,cB)}



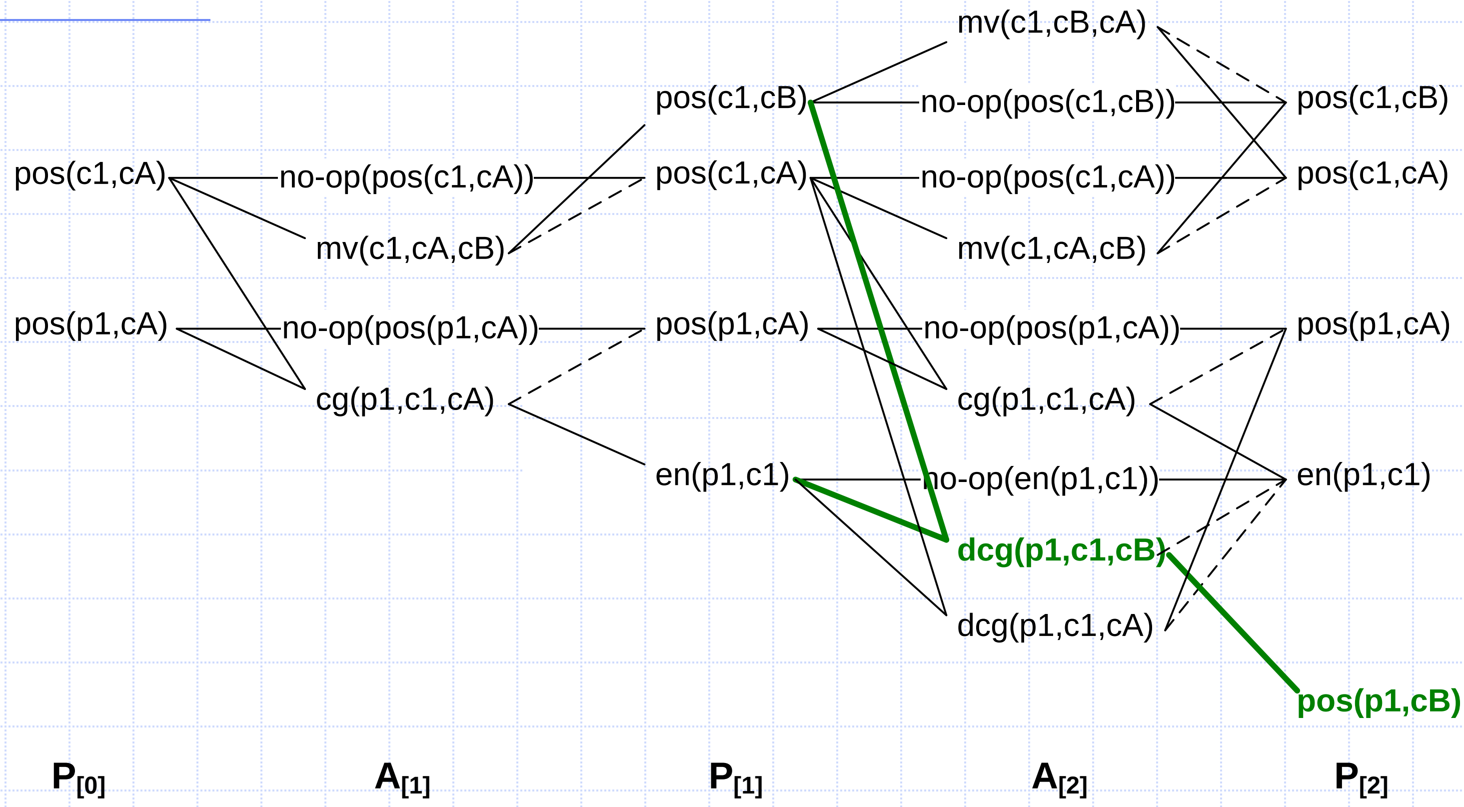
Example (level P[1])

P[1]

en(p1,c1) x {pos(p1,cA), pos(c1,cB)}
pos(c1,cA) x {pos(c1,cB)}
pos(c1,cB) x {en(p1,c1), pos(c1,cA)}



Example (level A[2])



Action **dcg(p1,c1,cB)** would not appear in **A[2]** because its preconditions are inconsistent. Consequently, literal pos(p1,cB) would not appear in **P[2],** which is an indication that the goal cannot be achieved at level **P[2]** and so the graph must be extended one more level (**A[3], P[3])**

Example (level A[2])

A[2]

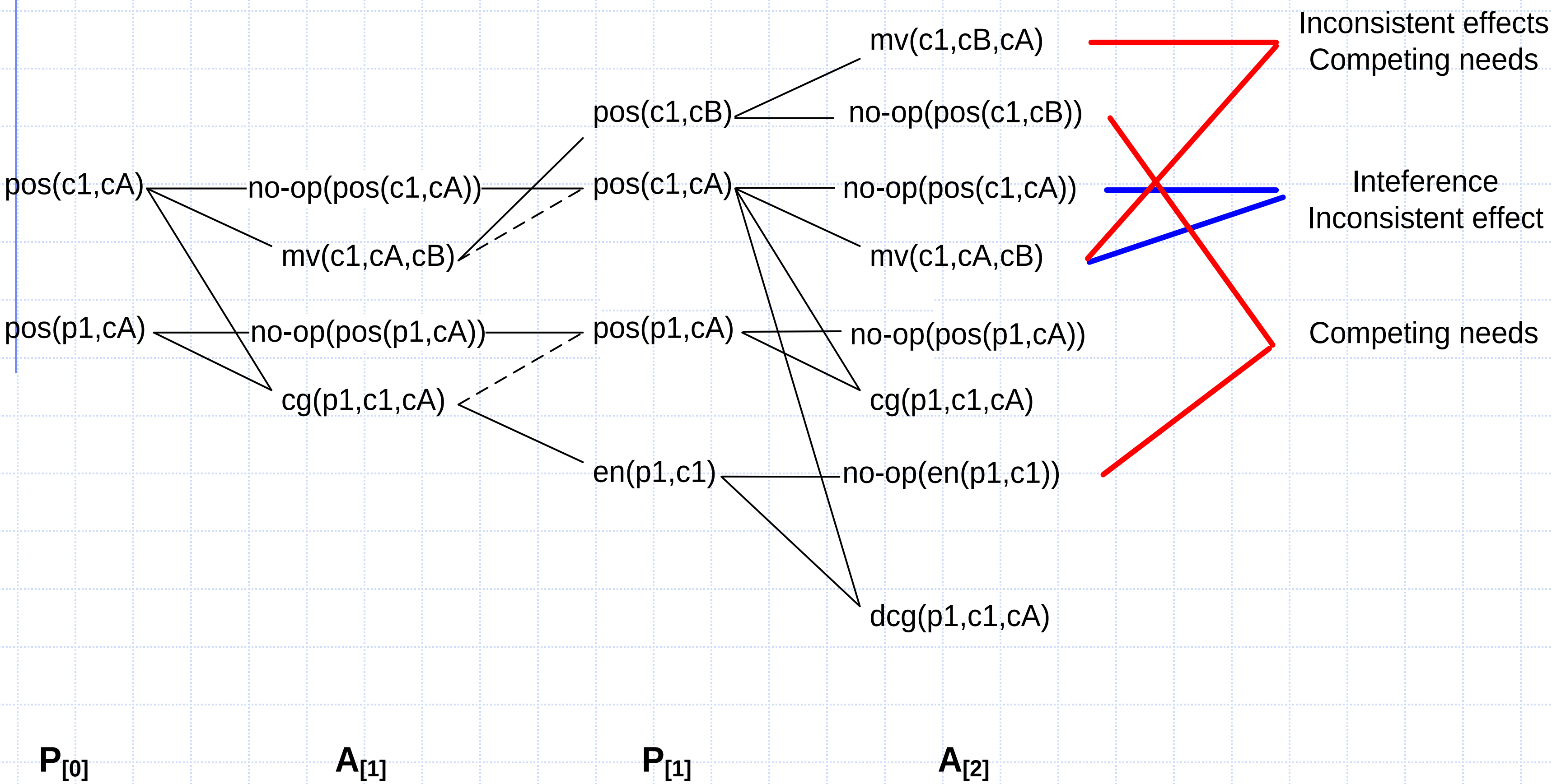
mv(c1,cA,cB) x
{mv(c1,cB,cA), cg(p1,c1,cA), dcg(p1,c1,cA), no-op(pos(c1,cA)), no-op(pos(c1,cB))}

mv(c1,cB,cA) x
{no-op(pos(c1,cB)), no-op(en(p1,c1)), no-op(pos(c1,cA)), cg(p1,c1,cA), dcg(p1,c1,cA), mv(c1,cA,cB)}

cg(p1,c1,cA) x
{no-op(pos(p1,cA)), mv(c1,cA,cB), mv(c1,cB,cA), no-op(pos(c1,cB)), dcg(p1,c1,cA), no/op(en(p1,c1))}

dcg(p1,c1,cA) x
{cg(p1,c1,cA), no-op(en(p1,c1)), mv(c1,cB,cA), no-op(pos(c1,cB)), mv(c1,cA,cB), no-op(pos(p1,cA))}

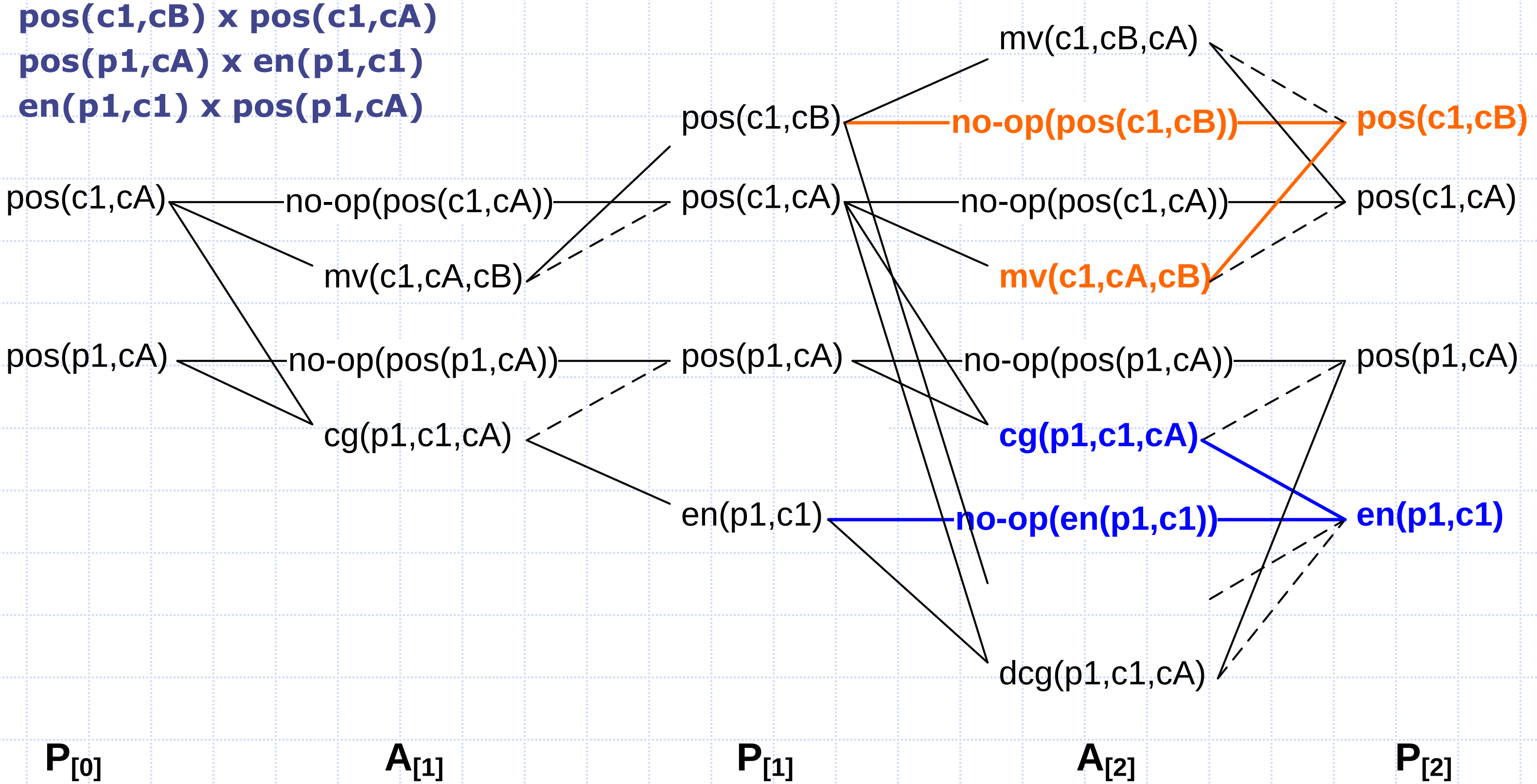
Example (level A[2])



Example (level P[2])

P[2]

pos(c1,cA) x pos(c1,cB)
pos(c1,cB) x pos(c1,cA)
pos(p1,cA) x en(p1,c1)
en(p1,c1) x pos(p1,cA)



pos(c1,cB) and **en(p1,c1)** are no longer mutex because now not all ways of achieving **pos(p1,cB)** are pairwise mutex with all ways of achieving **en(p1,c1)**

Example (level P[2])

Literals in P[2]

pos(c1,cB)

en(p1,c1)

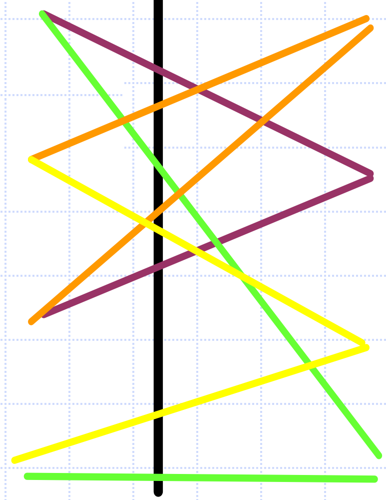
Actions in A[2] achieving the literals

no-op(pos(c1,cB))

mv(c1,cA,cB)

no-op(en(p1,c1))

cg(p1,c1,cA)



Mutex relations

Non-mutex

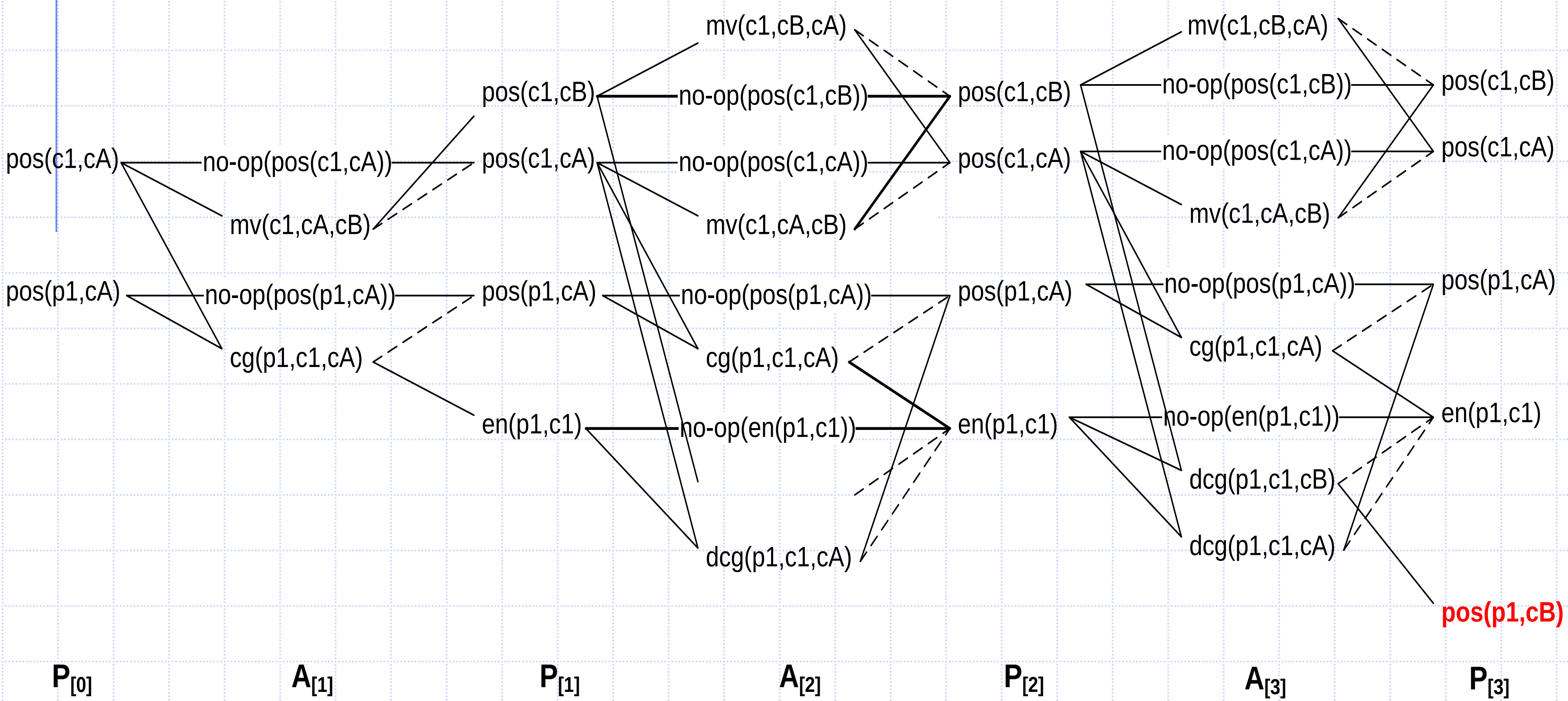
Competing needs in A[2]

Interferences in A[2]

Competing needs in A[2]

Example (layers A[3], P[3])

- The planning graph is extended level by level until the goals are achieved (necessary but not sufficient condition)

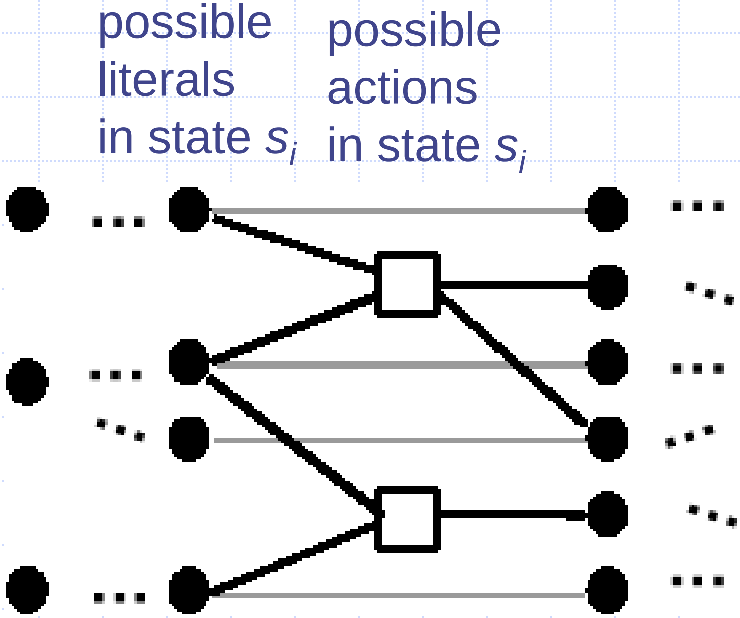


The Graphplan algorithm

procedure Graphplan:

- for $k = 0, 1, 2, \ldots$
  - *Graph expansion:*
    - create a "planning graph" that contains k "levels"
  - Check whether the planning graph satisfies a necessary (but insufficient) condition for plan existence
  - If it does, then
    - do *solution extraction:*
      - backward search, modified to consider only the actions in the planning graph
      - if we find a solution, then return it

relaxed problem



Solution extraction

- Check to see whether there's a possible solution
  - All of the goals appear at a proposition level
  - None are mutex with each other
- Thus, there's a chance that a plan exists

Solution Extraction

The set of goals we are trying to achieve

The level of the state s_j

A real action or a maintenance action

procedure Solution-extraction(g,j)

if $j=0$ then return the solution

for each literal l in g

nondeterministically choose an action to use in state s_{j-1} to achieve l

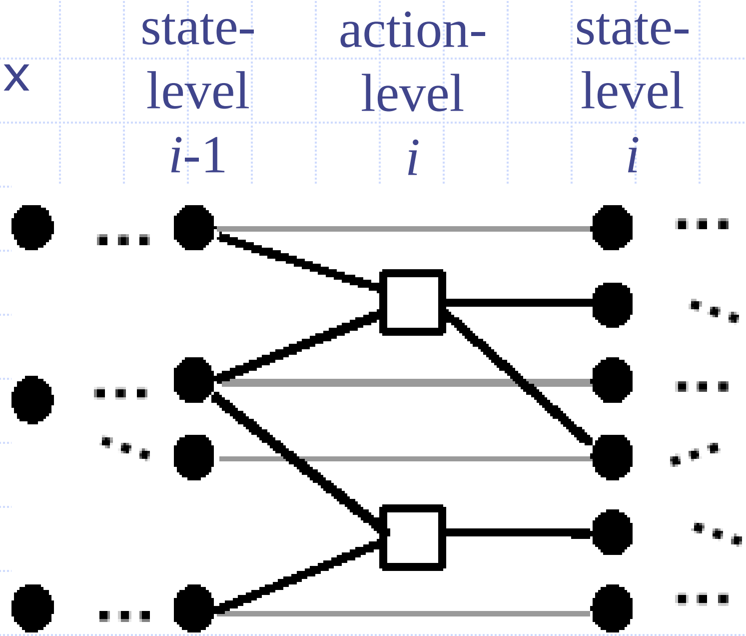
if any pair of chosen actions are mutex

then backtrack

g' := {the preconditions of the chosen actions}

Solution-extraction(g', $j-1$)

end Solution-extraction



Solution extraction

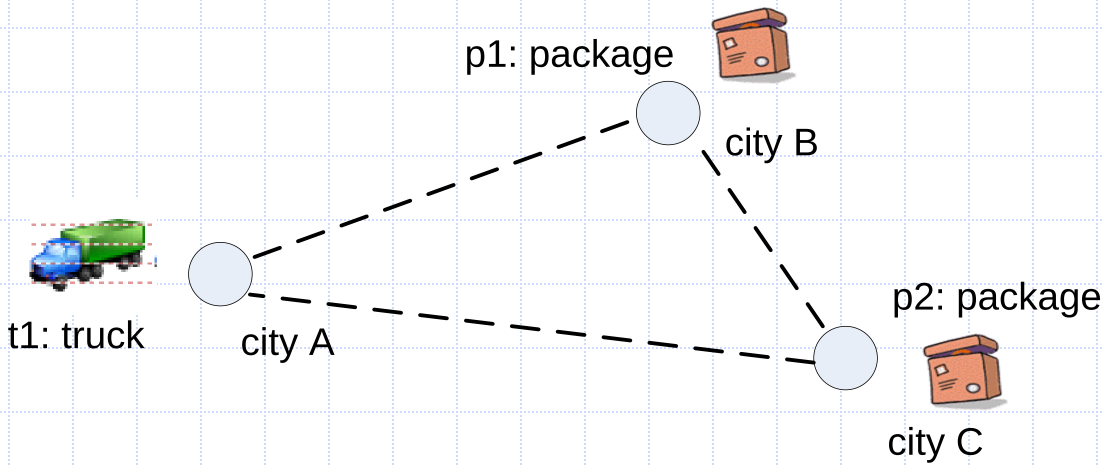
- If any pair of chosen actions are mutex then backtrack:
  - Choose another action for a literal so that the pair is not mutex
- If all sets of actions (combinations of actions) in level A_i for the literals at level P_i contain mutex actions then go back and do more graph expansion => generate another action level and another proposition level
- Termination of Graphplan:
  - Literals increase monotonically
  - Actions increase monotonically
  - Mutexes decrease monotonically
  - The graph levels off in a finite number of steps (two subsequent proposition levels are identical)

Solution extraction and mutex relations

- The mutual exclusion rules do not guarantee to find *all* mutual exclusion relationships, but usually find a large number of them (in fact, determining *all* mutual exclusion relationships can be as hard as finding a plan).

- Mutual exclusion rules only find binary mutex but there also exists other higher-order mutex, eg. *ternary* mutex …

- Example:



Goals: (at p1 cityA) and (at p2 cityA)

Solution extraction and mutex relations

- Shortest solution plan involves six steps:
  - {move (t1,cityA,cityB)}
  - {load (t1,p1,cityB)}
  - {move (t1,cityB,cityC)}
  - {load (t1,p2,cityC)}
  - {move (t1,cityC,cityA)}
  - {unload (t1,p1,cityA), unload (t1,p2,cityA)}

- However, Graphplan infers that the goals can be achieved together by level 5 of the graph. This is because **(in p1 t1) (in p2 t1)** and **(at t1 cityA)** are achieved by level 4 of the graph and Graphaplan will not find any mutex relation between any pair of literals:

Solution extraction and mutex relations

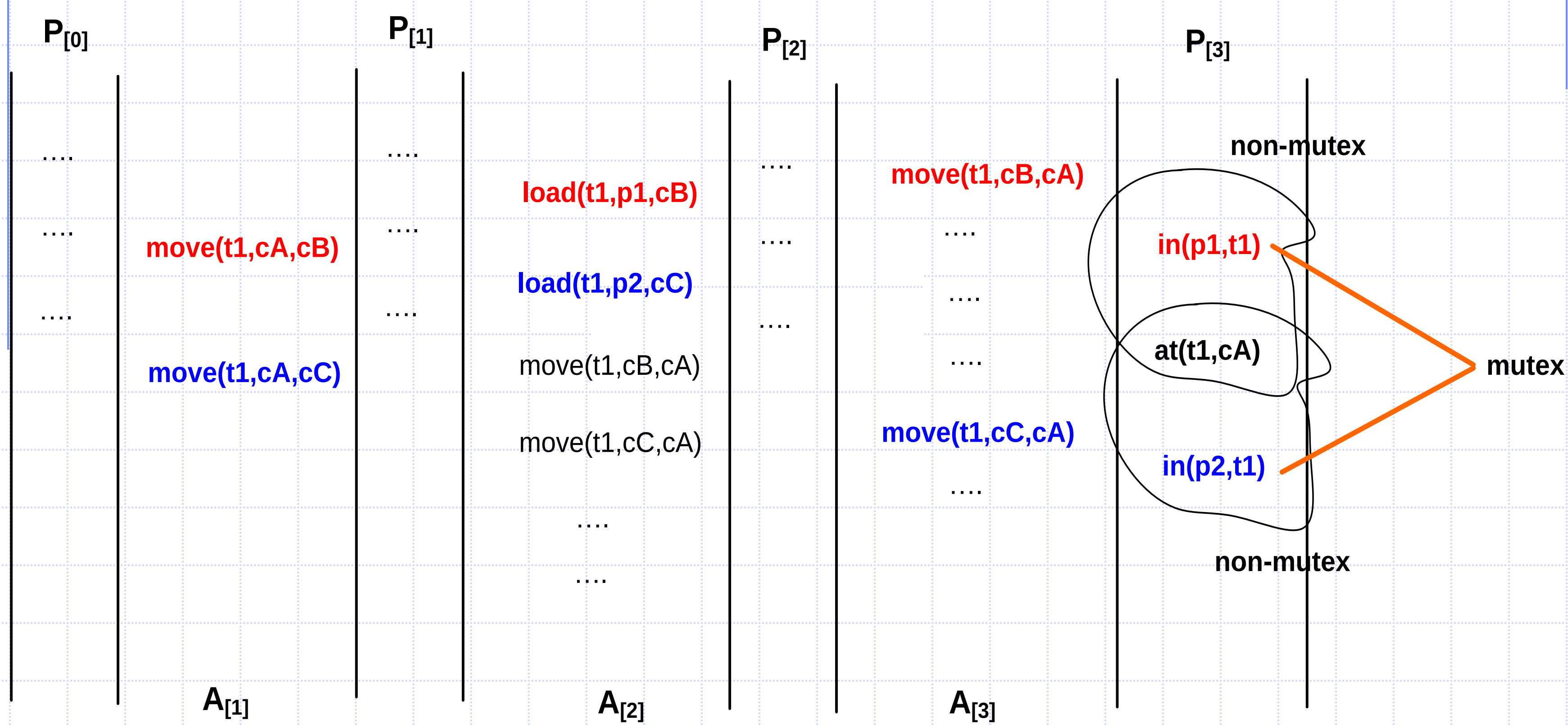
- the pairs **{(in p1 t1),(at t1 cityA)}** and **{(in p2 t1),(at t1 cityA)}** are not mutex at level 4 because there is a 3-step plan which will get the truck loaded with either package to the destination; and when two literals are found non-mutex, they can never subsequently become mutex since if they can occur in the same state then they will always be able to appear in the same state

- the pair **{(in p1 t1),(in p2 t1)}** is also not mutex at level 5 because there is a 4-step plan which will have the truck loaded with both packages although not at the destination

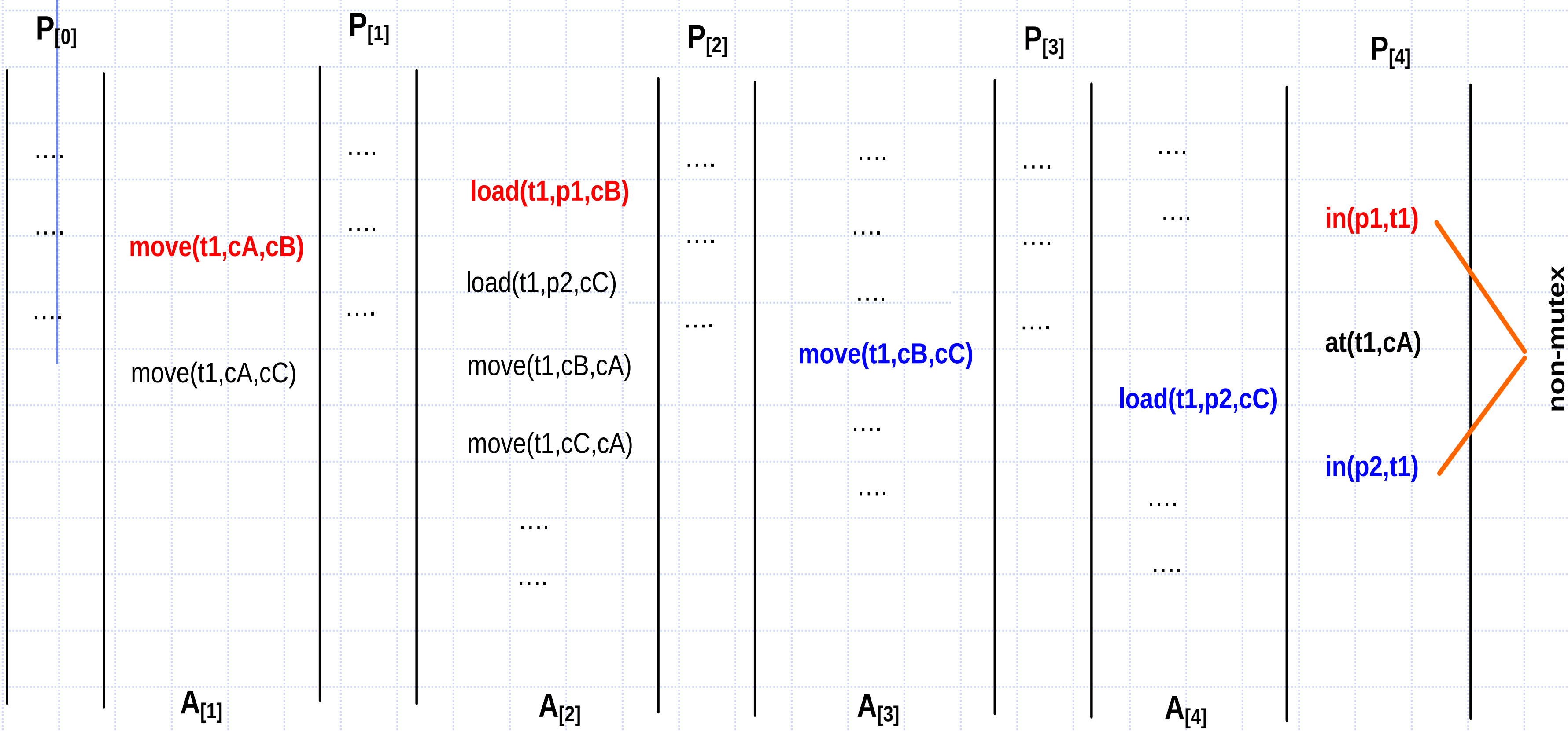
- then, no *binary* mutex relation is found with the three literals by level 4 although there is a mutex among the three literals

- Consequently, Graphplan begins searching for a plan from level 5 when it is impossible for one to be found until level 6 has been constructed.

Solution extraction and mutex relations



Solution extraction and mutex relations



Discussion

- Advantage:
  - The backward-search part of Graphplan—which is the hard part—will only look at the actions in the planning graph
  - smaller search space than PSP; thus faster
- Disadvantage:
  - To generate the planning graph, Graphplan creates a huge number of ground atoms
  - Many of them may be irrelevant
- Can alleviate (but not eliminate) this problem by assigning data types to the variables and constants
  - Only instantiate variables to terms of the same data type
- For classical planning, the advantage outweighs the disadvantage
  - GraphPlan solves classical planning problems much faster than PSP

Heuristics derived from planning graphs

- Recall how GraphPlan works:

loop

Graph expansion:

this takes polynomial time

extend a "planning graph" forward from the initial state

until we have achieved a necessary (but insufficient) condition for plan existence

Solution extraction:

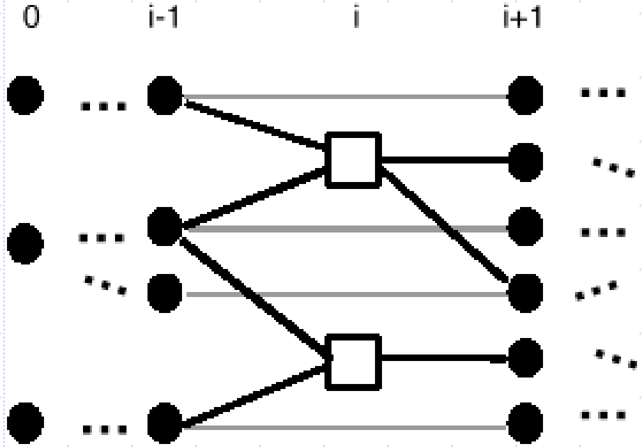
this takes exponential time

search backward from the goal, looking for a correct plan

if we find one, then return it

repeat

Heuristics derived from planning graphs



- In the graph, there are alternating layers of ground literals and actions
- The number of "action" layers is a lower bound on the number of actions in the plan
- Construct a planning graph, starting at ***s***
- **$h(g_i)$:** estimate to achieve **g_i** from **s**
- **$h(g_i)$=** level of the first layer that "possibly achieves" **g_i**

Heuristics derived from planning graphs

- Heuristics for a set of goal **G**={pos(p1,cB),pos(c1,cA)}:
  - The ***max level heuristic*** takes the maximum level cost of any of the goals (admissible, not very accurate).
    $hmax(G)=max_{g \in G}\ h(g)$ // $hmax(G)=max(0,3)=3$
  - The sum ***level heuristic*** returns the sum of the level costs of the goals (inadmissible, works well in practice)
    $hsum(G)=\sum g \in G\ h(g)$ // $hsum(G)=0+3=3$
  - The ***max$_2$ level heuristic*** takes the maximum level at which coexist any pair of goals.
    $hmax_2(G)=_{max\{g1,g2\} \in G}\ h(g1 \wedge g2)$ // $hmax_2(G)= max(4)=4$
  - The ***max$_k$ level heuristic*** takes the maximum level at which coexist any *k* goals. $hmax_k(G)=_{max\{g1,g2,\ ..,\ gk\} \in G}\ h(g1 \wedge g2 \wedge \ldots \wedge gk)$

Heuristics derived from planning graphs

- Heuristics used by forward state-space planners (like FF)
- Application of the heuristic over each state in the tree
- Planning graph + mutex calculation over each state => very costly process
- Computing heuristics on a **relaxed planning graph (RPG):**
  - Ignoring negated effects
  - No mutex calculation
  - Slide 11 shows an example on how the RPG will look like
- hmax(G)=max(0,2)=2
- hsum(G)=0+2=2
- $hmax_2$(G)= max(2)=2

Heuristics derived from planning graphs

- **Relaxed Planning Graph**:
  - No delete effects
  - No mutex
  - No no-op actions because we are only interested in the first appearance of any action/proposition
- Relaxed plan heuristic: extract a plan from a **RPG**
- See examples of the blocks-world domain

The FastForward Planner

- Use a heuristic function similar to $h(s) = \Delta^g(s,g)$
  - Some ways to improve it (I'll skip the details)
- Don't want an A*-style search (takes too much memory)
- Instead, use a greedy procedure:

until we have a solution, do
    expand the current state *s*
    s := the child of *s* for which *h(s)* is smallest
        (i.e., the child we think is closest to a solution)

The FastForward Planner

- Use a heuristic function *h*(*s*) = relaxed plan
- Don't want an A*-style search (takes too much memory)
- Instead, use a greedy procedure:

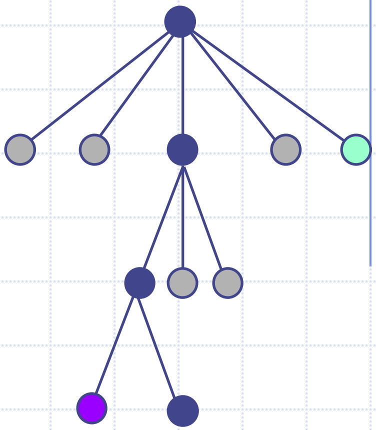
until we have a solution, do

expand the current state *s*

s := the child of *s* for which *h(s)* is smallest

(i.e., the child we think is closest to a solution)

The FastForward Planner



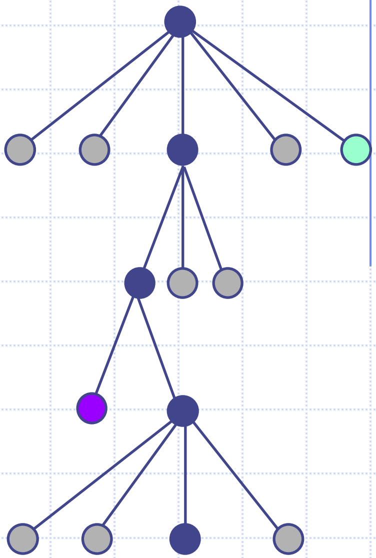
- Use a heuristic function $h(s)$ = relaxed plan
- Don't want an A*-style search (takes too much memory)
- Instead, use a greedy procedure:

until we have a solution, do
    expand the current state *s*
    s := the child of *s* for which *h(s)* is smallest
        (i.e., the child we think is closest to a solution)

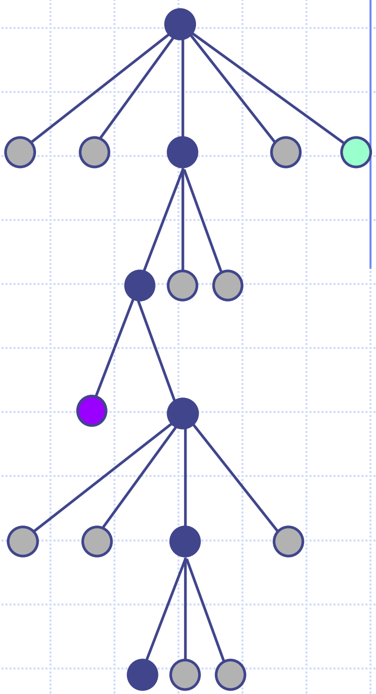
The FastForward Planner

- Use a heuristic function *h*(*s*) = relaxed plan
- Don’t want an A*-style search (takes too much memory)
- Instead, use a greedy procedure:

until we have a solution, do
    expand the current state *s*
    s := the child of *s* for which *h(s)* is smallest
        (i.e., the child we think is closest to a solution)



The FastForward Planner



- Use a heuristic function similar to $h(s)$ = relaxed plan
- Don't want an A*-style search (takes too much memory)
- Instead, use a greedy procedure:

  until we have a solution, do

  expand the current state *s*

  s := the child of *s* for which *h(s)* is smallest

  (i.e., the child we think is closest to a solution)

- There are some ways FF improves on this
  - e.g. a way to escape from local minima
    - breadth-first search, stopping when a node with lower cost is found

- Can't guarantee how fast it will find a solution, or how good a solution it will find
  - However, it works pretty well on many problems