Graphplan

UNIVERSITÄT DES SAARLANDES

GRAPHPLAN

- Many Planning Systems now use these Ideas:
  - GraphPlan
  - IPP
  - STAN
  - SGP
  - Blackbox
  - Medic
- Runs orders of magnitude faster than partial order planners (as POP, SNLP, UFPOP, etc.)

GraphPlan: THE BASIC IDEA

1. Construct the (initial) Planning Graph

2. Extract Solution (if possible)
   with fast Graph-Search-Algorithms

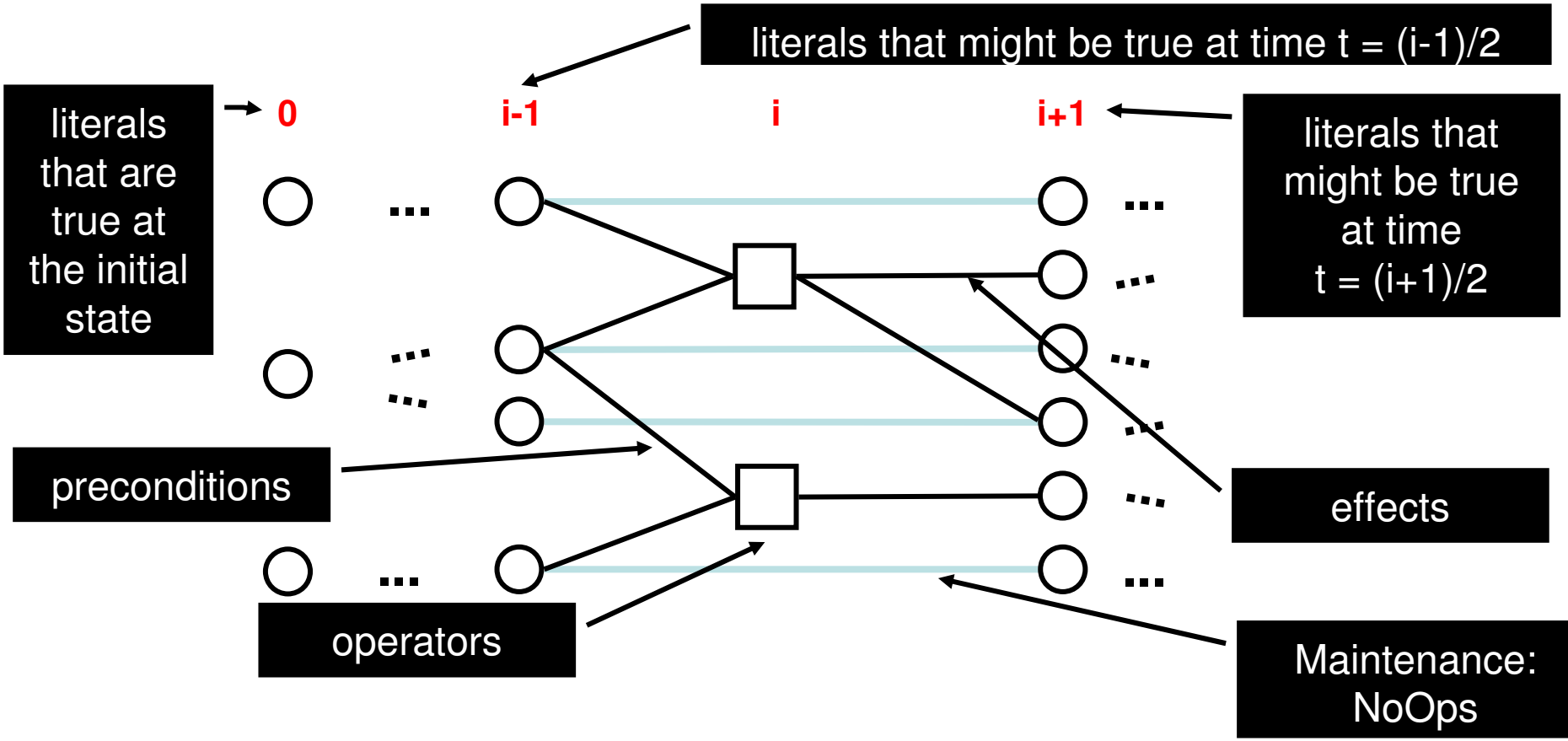
3. Else expand Graph and goto 2.

UNIVERSITÄT DES SAARLANDES

DFKI

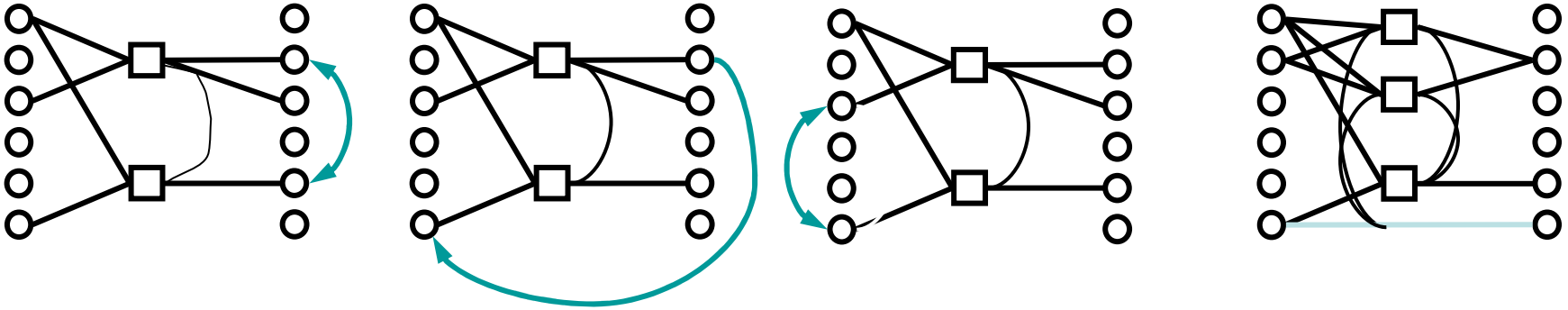
The Planning Graph

Alternating layers of ground literals and actions (ground instances of operators) representing the literals and actions that might occur at each time step $0 < i < N$



UNIVERSITÄT DES SAARLANDES

Mutual Exclusion



Inconsistent Effects

Interference

Competing Needs

Inconsistent Support

- Two actions are mutex if

  Inconsistent effects: an effect of one negates an effect of the other

  Interference: one effect deletes a precondition of the other

  Competing needs: they have mutually exclusive preconditions

- Two literals are mutex if

  Inconsistent support: one is the negation of the other, or all ways of achieving them are pairwise mutex

UNIVERSITÄT DES SAARLANDES

The Algorithm

```
function GRAPHPLAN(problem) returns solution or failure

  graph ← INITIAL-PLANNING-GRAPH(problem)
  goals ← GOALS[problem]

  loop do
    if goals all non-mutex in last level of graph then do
     solution ← EXTRACT-SOLUTION (graph, goals, LENGTH(graph))

        if solution ≠ failure then return solution

        else if NO-SOLUTION-POSSIBLE(graph) then return failure

    graph ← EXPAND-GRAPH(graph, problem)
```

The Sweetheart Example :

Suppose you want to prepare dinner, clean the garbage and give a present as a surprise to your sweetheart, who is asleep.

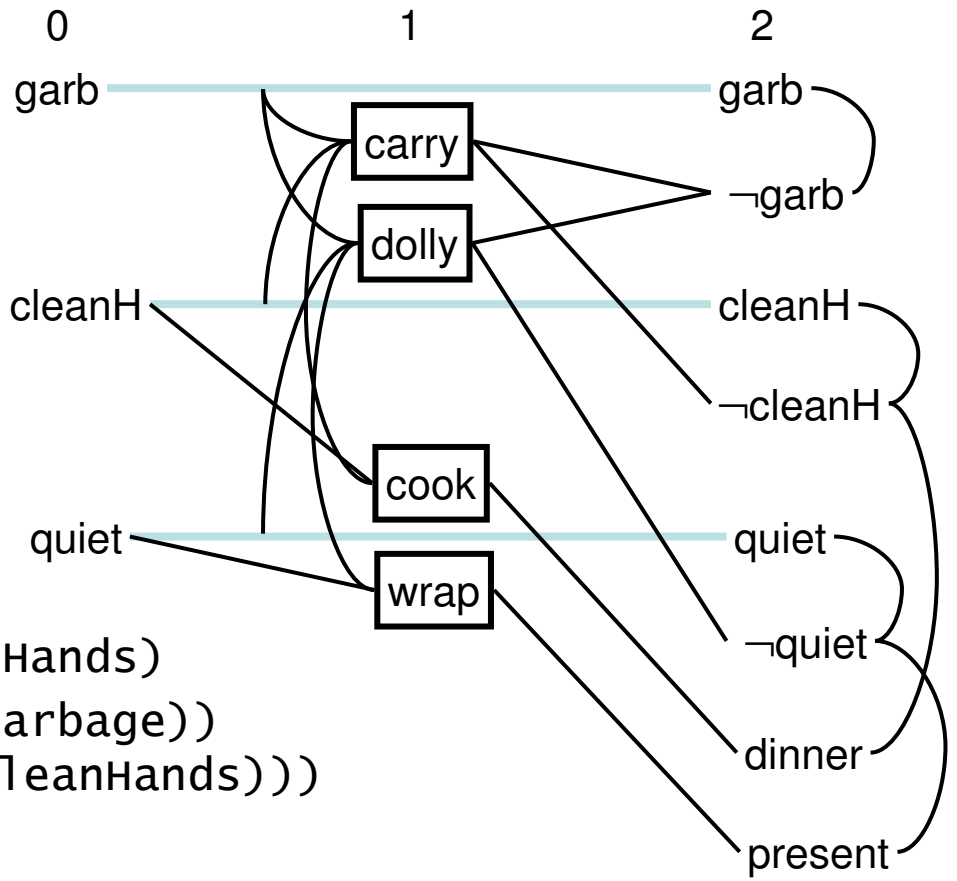
```
Initial Conditions:  (and (garbage) (cleanHands) (quiet))
Goal:                (and (dinner) (present) (not (garbage))

Actions:

  cook     :precondition (cleanHands)
           :effect (dinner)
  wrap     :precondition (quiet)
           :effect (present)
  carry    :precondition
           :effect (and (not (garbage)) (not (cleanHands)))
  dolly    :precondition
           :effect (and (not (garbage)) (not (quiet)))
```

Also there are NoOps = “maintenance actions”

UNIVERSITÄT DES SAARLANDES

The Graph for this Example (1)

- Generate the first two levels of the planning graph
- carry is mutex with
  NoOp (garbage)
  (inconsistent effects)
- dolly is mutex with wrap
  (interference)
- ¬quiet is mutex with present
  (inconsistent support)




```
cook    :precondition (cleanHands)
        :effect (and (dinner)
                     (not (garbage))
                     (not (cleanHands)))
```

Extraction of a Solution for the Example (1):

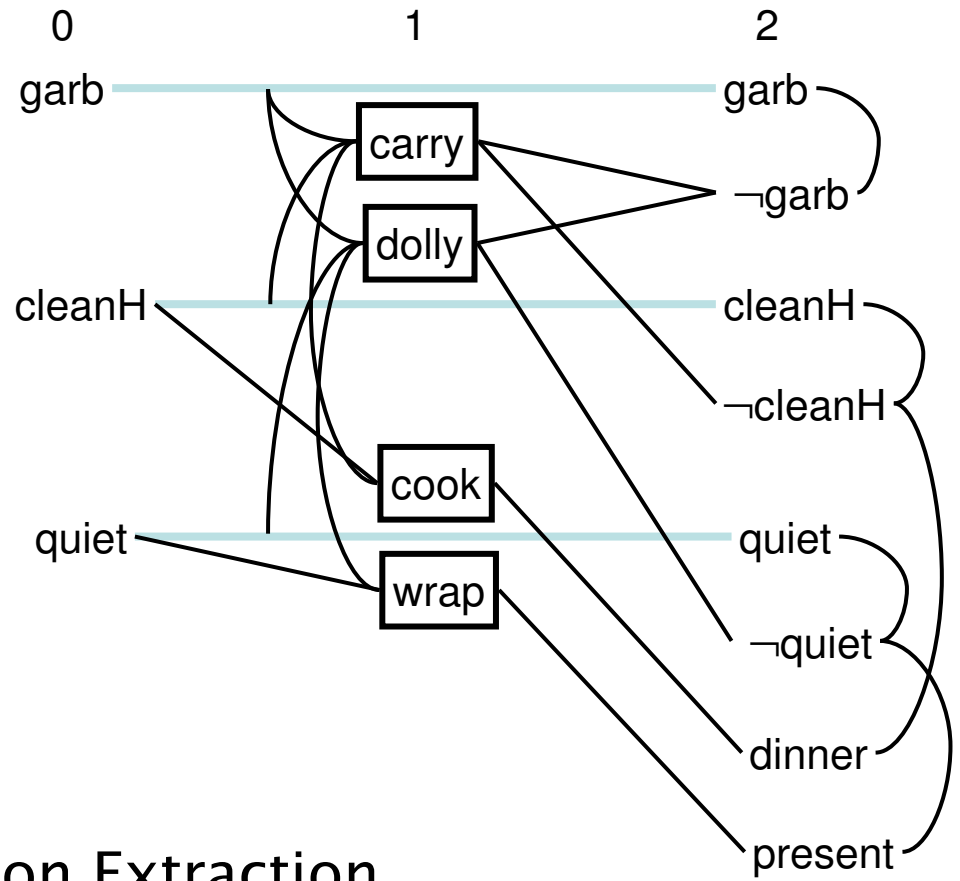
Check to see whether there's a possible plan:

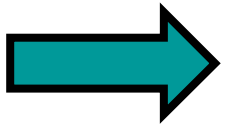
- Recall that the goal is:

```
(and (dinner)
     (present)
     (not (garbage)))
```

- Note that
  - All literals are present at level 2
  - None are mutex with each other

- Thus there is a chance that a plan exists



 Solution Extraction

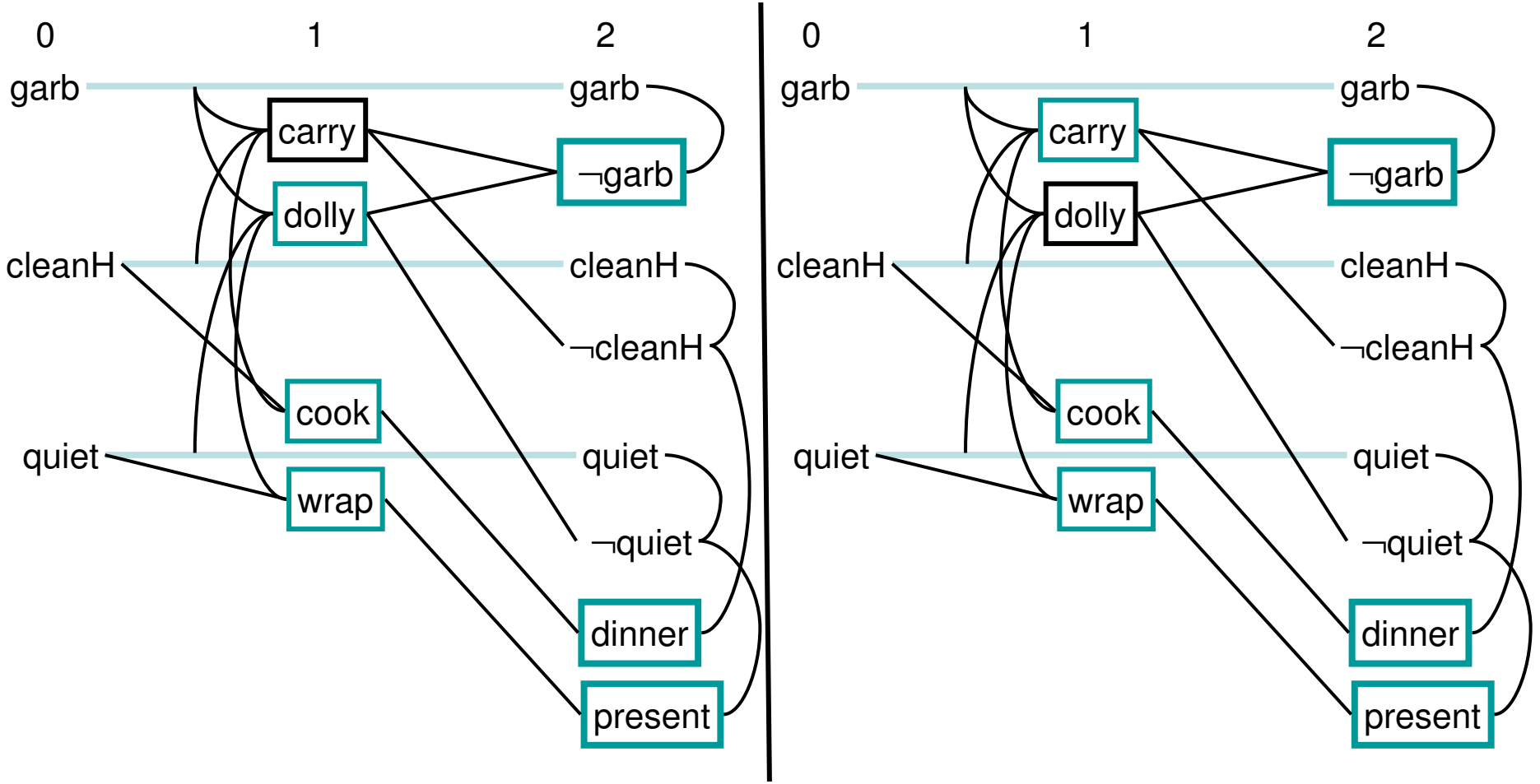
Solution Extraction: The Algorithm

```
procedure SOLUTION-EXTRACTION(goal_set, graphlevel)

    if graphlevel = 0 we have found a solution
    for each goal in goal_set
        choose (nondeterministically) an action
            at level graphlevel-1 that achieves it
    if any pair of chosen actions are mutex, then backtrack
    precondition_set:= {the precondition of the chosen actions}

    SOLUTION-EXTRACTION(precondition_set, graphlevel-2)
```

Level of the graph we are at

Solution Extraction for the Example (2):

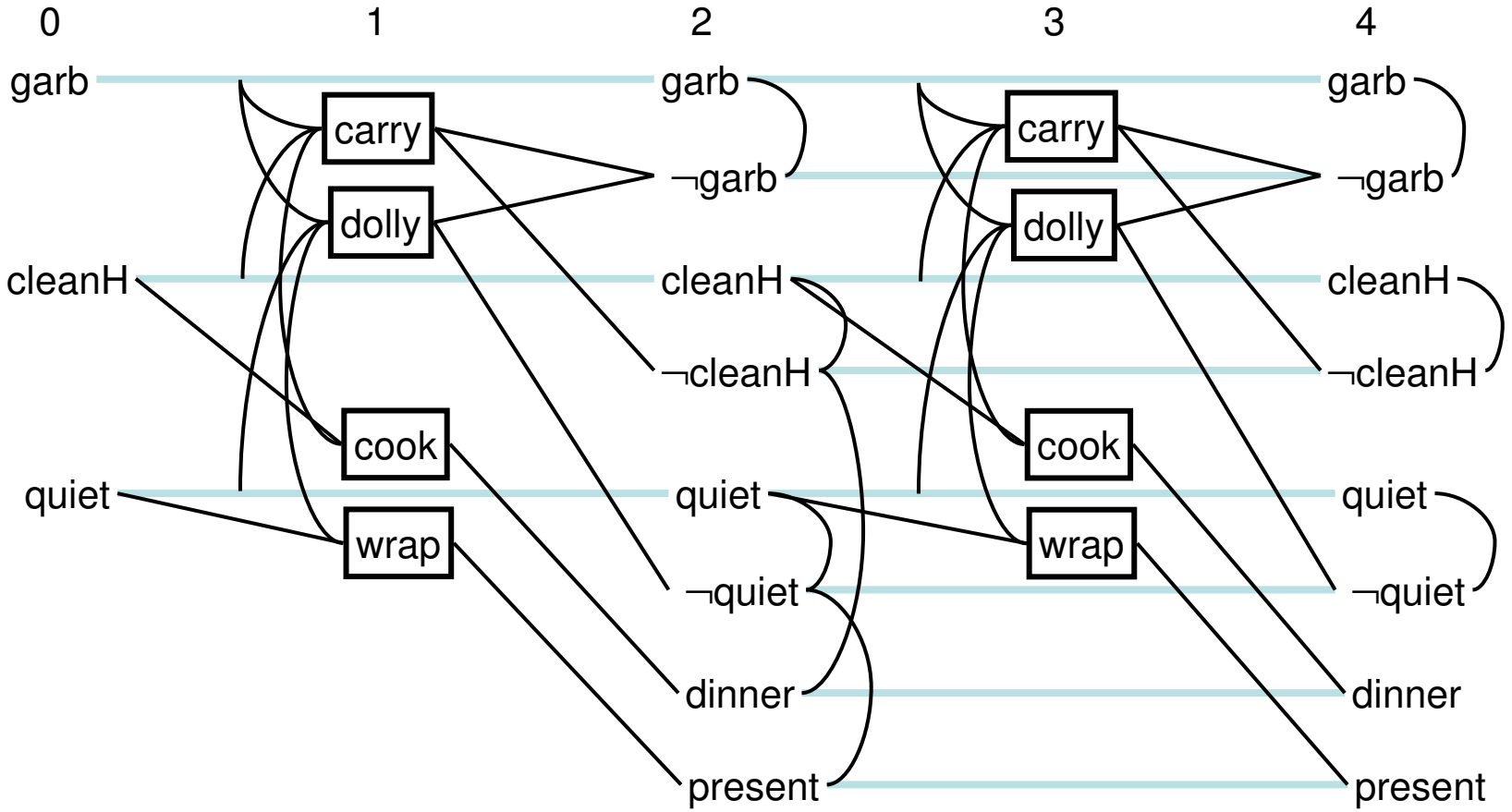
- Two sets of actions for the goals at level 2
- Neither works: both sets contain actions that are mutex



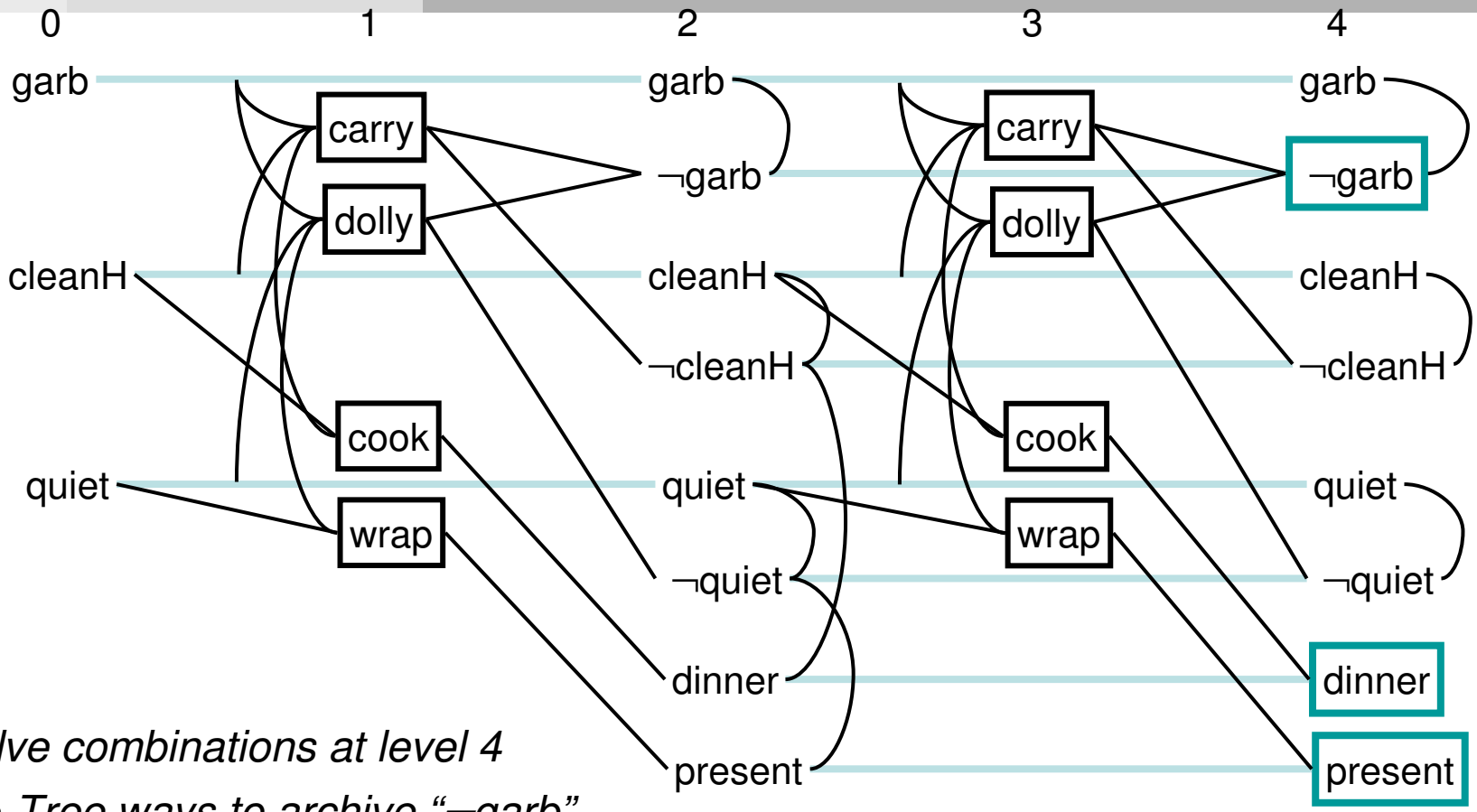
UNIVERSITÄT DES SAARLANDES

Solution Extraction Example (3)

- Go back and do more graph extension: generate two more levels



Example: Solution extraction (4)



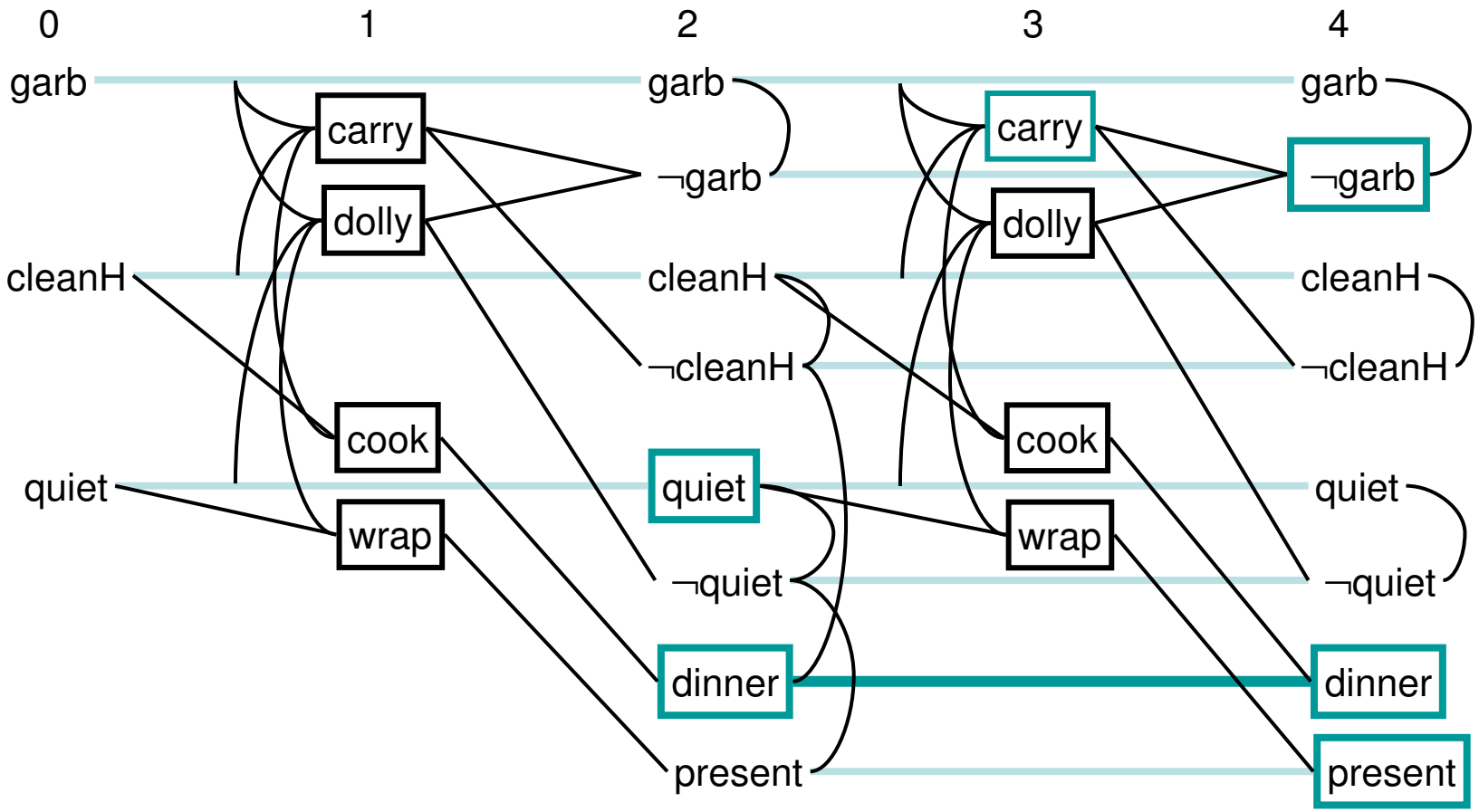
Twelve combinations at level 4

- *Tree ways to archive "¬garb"*
- *Two ways to archive "dinner"*
- *Two ways to archive "present"*

UNIVERSITÄT DES SAARLANDES

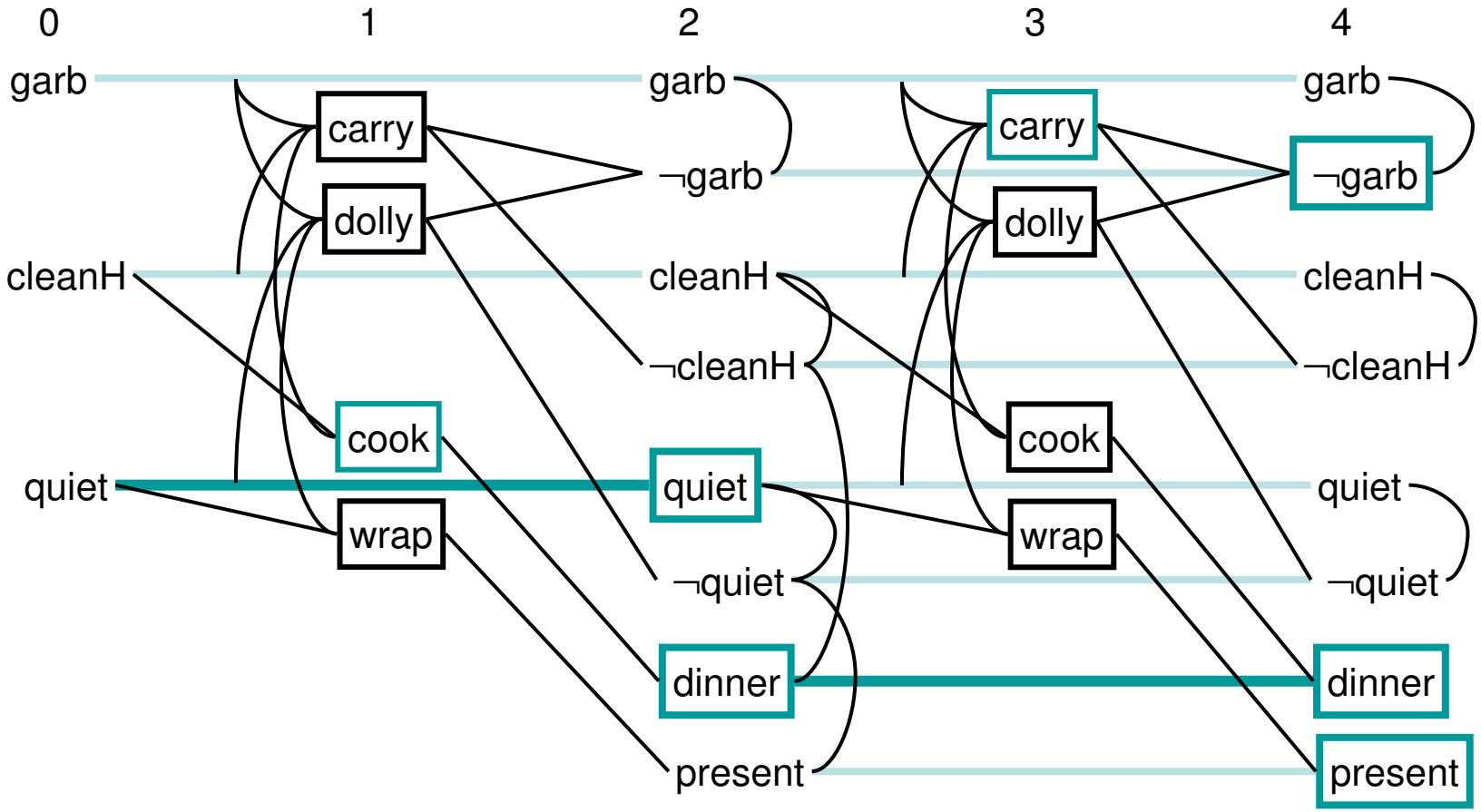
Example: Solution extraction (5)

Several of the combinations look OK at level 2. Here is one of them:



Example: Solution extraction (6)

Call Solution-Extraction recursively at level 2; one combination works, so we have got a plan



Compare with POP

- Graphplan will examine fewer actions than POP, because it will only look at actions in the planning graph
- However (like POP), Graphplan may examine the same actions more than once if it finds multiple plans to achieve them
- Look at ways to make Graphplan more efficient

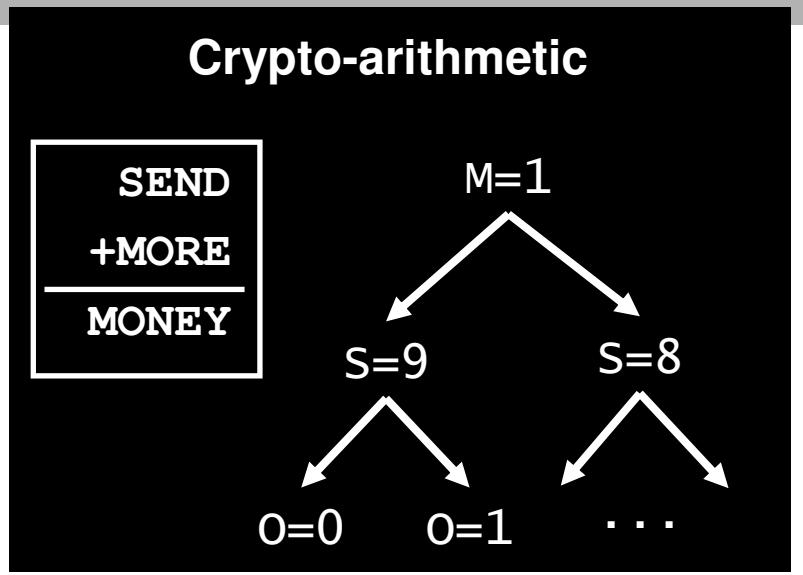
UNIVERSITÄT DES SAARLANDES

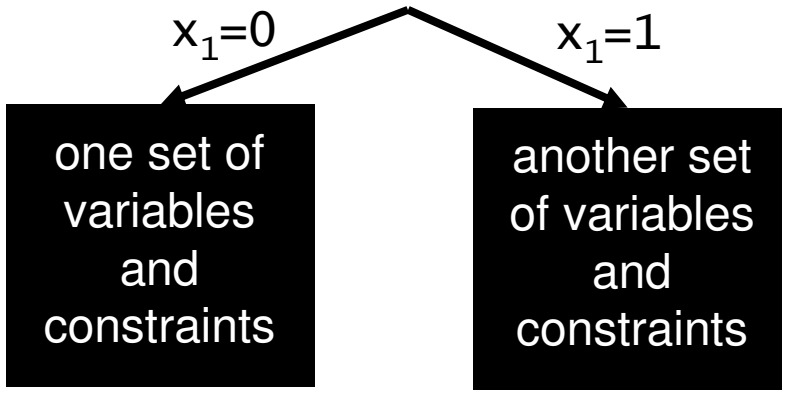
Ways to improve GraphPlan

- Techniques from Constraint satisfaction Problems (CSP)
  - Forward checking
  - Memorization
  - Variable ordering
  - Handling the closed-world assumption
- Action schemata, type analysis, and simplification
- Not reachable states (because of domain theory)
  - e.g. impossible: at(car1 Washington, t) $\wedge$ at(car1 Philadelphia, t)

Constraint Satisfaction Problems

- Set of variables:
  Each variable has a domain of possible values
- Set of constraints
  Find values for the variables that satisfy all the constraints
- Dynamic constraint satifaction problems
  When we select values for some variables, this changes what the remaining variables and constraints are

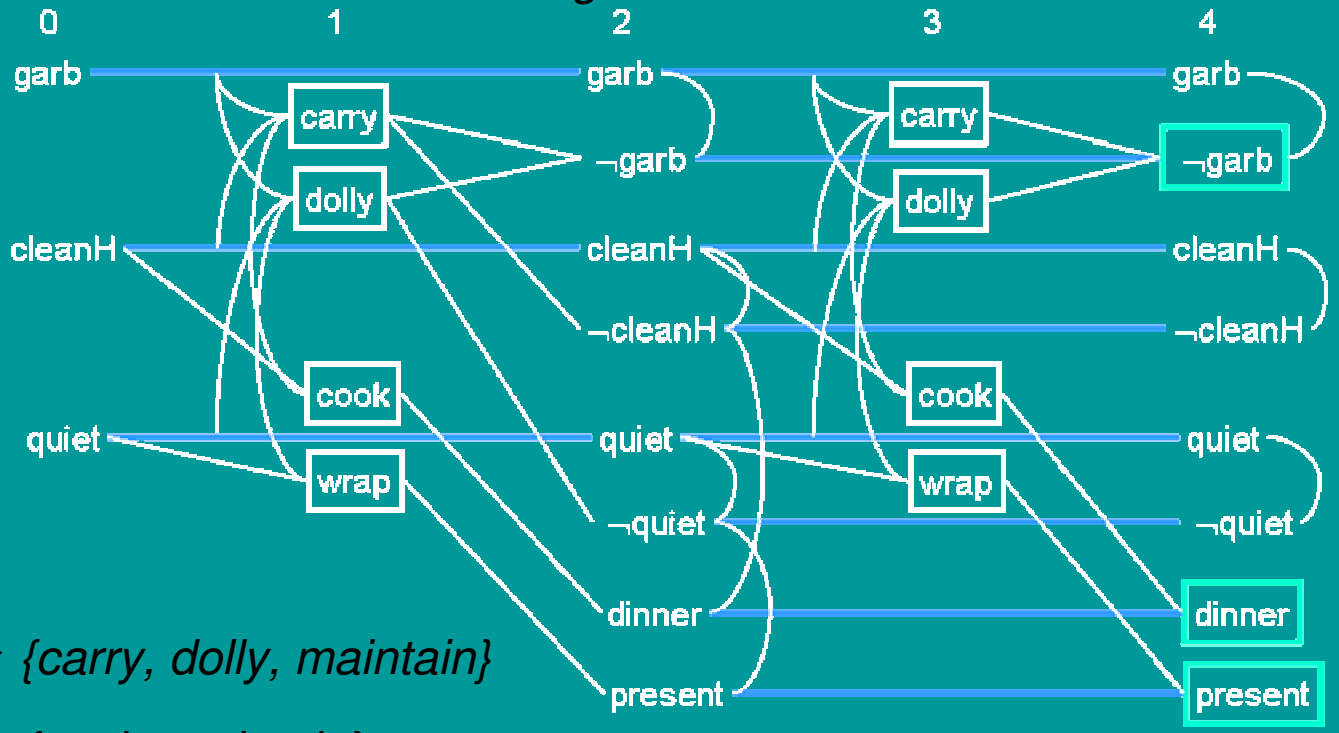




UNIVERSITÄT DES SAARLANDES

Solution extraction as Dynamic Constraint Satisfaction (1)

- *A different CSP variable for each subgoal at each level*



- $V_{4,\neg garb} \in$ {carry, dolly, maintain}
- $V_{4,dinner} \in$ {cook, maintain}
- $V_{4,present} \in$ {wrap, maintain}
- Constraints: the mutex constraints (just like before)

UNIVERSITÄT DES SAARLANDES

Solution extraction as Dynamic Constraint Satisfaction (2)

- Once a solution is found at level 4 this defines another CSP at level 2
- So far, no real difference to what we had before, but

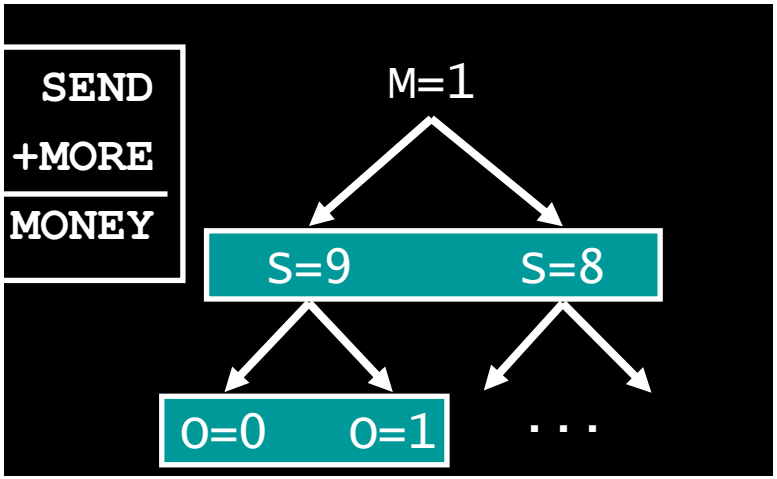
Use of some standard CSP techniques:

1. Forward checking
2. Dynamic Variable Ordering
3. Memorization

UNIVERSITÄT DES SAARLANDES

CSP-Techniques: Forward Checking

- Whenever we assign a value to a variable:
  - Also check unassigned variables to see if this shrinks their domains
  - If a domain shrinks to the empty set, then backtrack
- Example GraphPlan:
  - Suppose we assign $V_{4,\neg garb}$:= dolly
  - Then no longer feasible to have $V_{4,quiet}$:= *maintain*

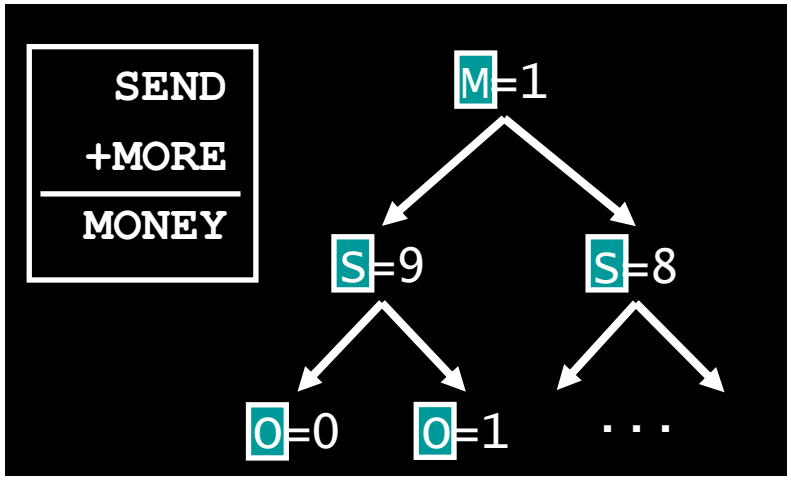


UNIVERSITÄT DES SAARLANDES

CSP-Techniques: Dynamic Variable Ordering

Heuristics for choosing the order in which to assign variable values

- Examples
  - If a variable has only one possible value, then assign it immediately
    - Otherwise, assign the variable with the fewest remaining possible values (use forward checking to figure out which variable)
    - Speeds up GraphPlan by about 50% [Kambhampati]

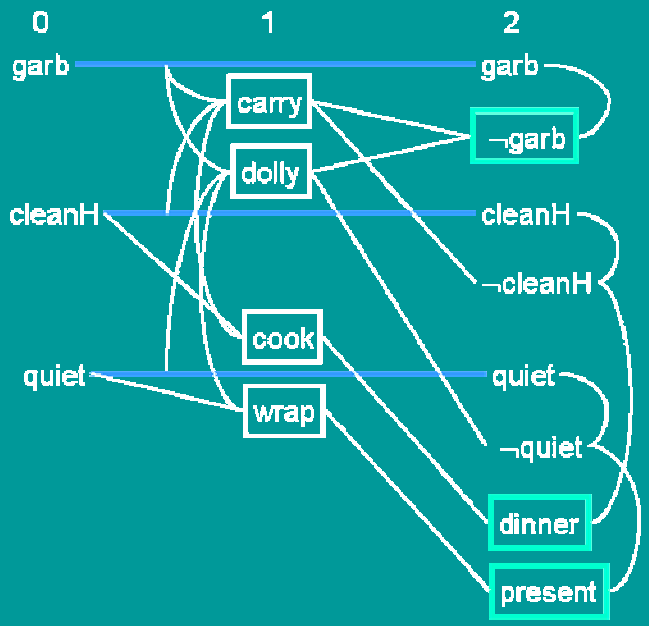


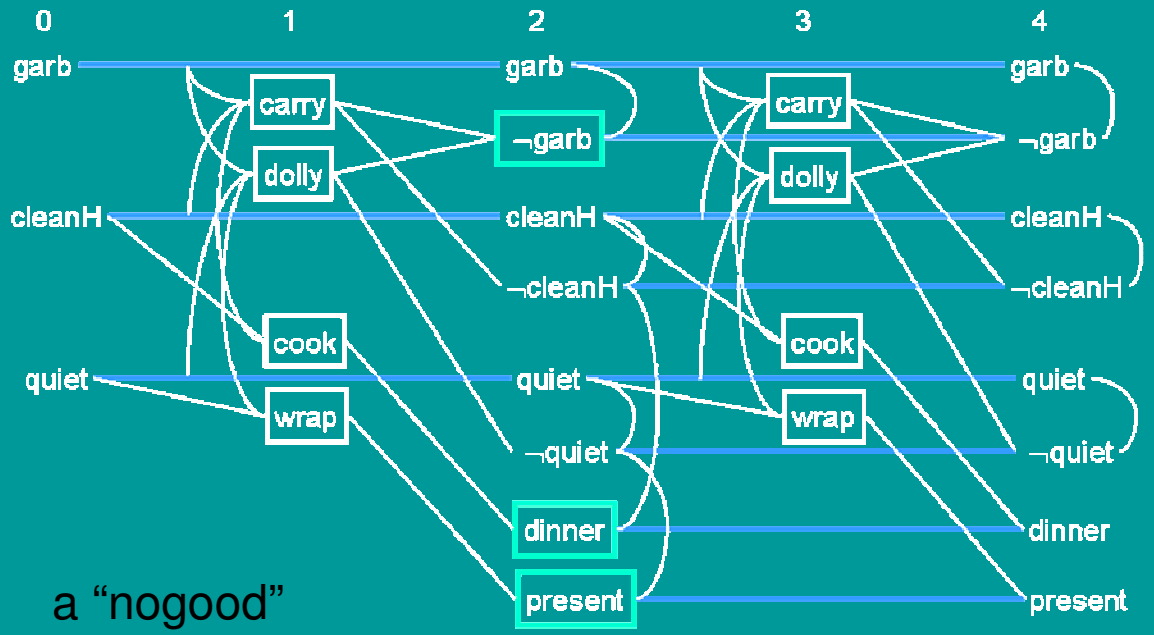
  - Choose Variabel ordering based on analysis of subgoal interactions
    - Speeds up GraphPlan by orders of magnitude on some problems [Koehler]

UNIVERSITÄT DES SAARLANDES

CSP-Techniques: Memorization

- If a set of subgoals is inconsistent at level *i*, record this info for future use
- For recursive calls to level *i* with the same subgoals backtrack immediately
- Can take lots of space, but can also speed things up significantly





a “nogood”
at level 2

UNIVERSITÄT DES SAARLANDES

Closed-World assumption

- Closed-world assumption:
  Any proposition not explicitly known to be true can be assumed to be false
- For GraphPlan:
  If something isn't explicitly mentioned in the initial state, then assume it's false

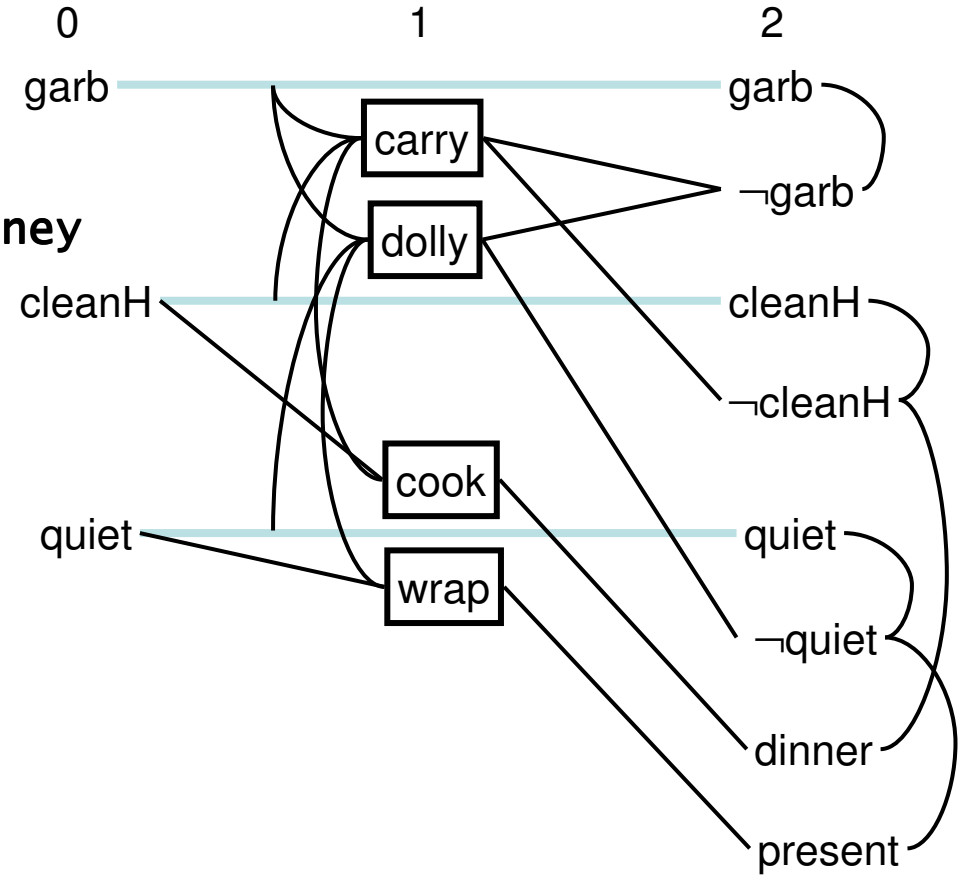
Example Closed-world Assumption

- Suppose we had another operator:

```
buy-dinner
    precondtion: have-money
    effects: dinner, ¬have-money
```

- Since "have-money" isn't mentioned in the initial state, assume it is false

- Thus, can't apply this operator



Action Schemata (i.e., Planning Operators) (1)

- So far, all our GraphPlan operators have been ground
  - What about operators that contain variables?

```
(defschema (drive)
   :parameters     (?v ?s ?d)
   :precondition   (and    (vehicle ?v)
                   (location ?s)
                   (location ?d)
                   (road-connect ?s ?d)
                   (at ?v ?s))
   :effect     (and  (¬(at ?v ?s))(at ?v ?d)))


drive(?v ?s ?d)
   precondition:   vehicle (?v)
                location (?s)
                location (?d)
                road-connect (?s ?d)
                at (?v ?s)
   effect:      ¬at (?v ?s), at (?v ?d)
```

UNIVERSITÄT DES SAARLANDES

Action Schemata (i.e., Planning Operators) (2)

- Put all applicable ground instances into the planning graph
  - Suppose the initial state contains *n* constants
  - Suppose an operator contains *k* variables
  - $O(n^k)$ instances of the operator at each level of the planning graph:

```
drive (?v, ?s, ?d)
        n * n * n
```

UNIVERSITÄT DES SAARLANDES

Type Analysis: Sorted logics

- Many operator instances will be irrelevant because they use variable values that can never be satisfied
- Type analysis
  - Determine which predicates represent types
  - Calculate the set of possible constants for each type
  - Instantiate ground actions only for plausibly typed combinations of constants

```
location(Philadelphia),
location(Washington),
location(Richmond),
at(Philadelphia),
vehicle(car1), vehicle(car2),
road-connect(Philadelphia,
Washington)
road-connect(Washington, Richmond)
```

```
drive(Washington, Philadelphia,
Richmond)
```

Not a vehicle

- Simplest form:
  If an initial condition is absent from the effects of any operators, then it will never change, so conclude that it is a type.
  E.g., “location” and “vehicle”

UNIVERSITÄT DES SAARLANDES

Simplification

- Don't need to include static terms in the planning graph
  - Remove them from the preconditions of actions
  - Only create instantiations that satisfy the static preconditions
- Can do even more complicated things [Fox & Long]

```
location(Philadelphia),
location(Washington), location(Richmond),
at(Philadelphia),
vehicle(car1), vehicle(car2),
road-connect(Philadelphia, Washington)
road-connect(Washington, Richmond)
at(Phiadelphia)
```

```
drive(car1, Philadelphia, Washington)
precondition:   at(car1, Philadelphia)
effect:         ¬at(car1, Philadelphia),
                at(car1, Washington)
```

```
drive(car2, Philadelphia, Washington)
precondition:   at(car2, Philadelphia)
effect:         ¬at(car2, Philadelphia),
                at(car2, Washington)
```

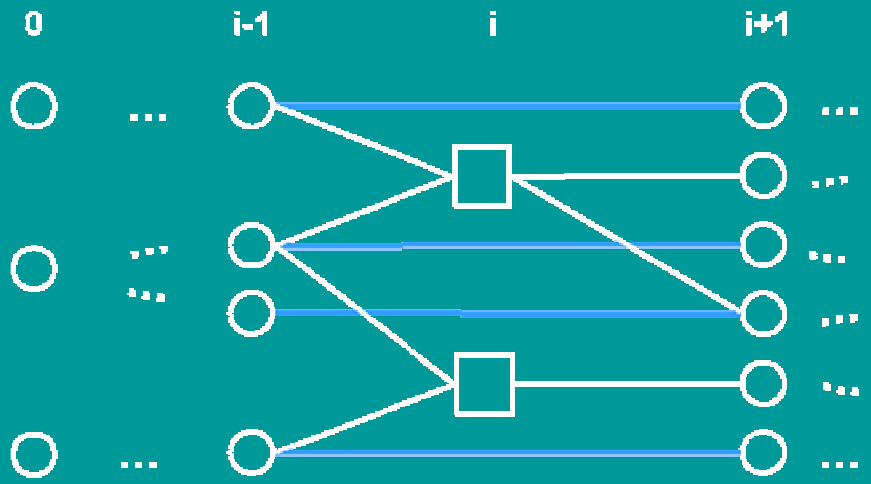
```
drive(car1, Philadelphia, Richmond)
precondition:   at(car1, Philadelphia)
effect:         ¬at(car1, Philadelphia),
                at(car1, Richmond)
```

UNIVERSITÄT DES SAARLANDES

Backward Planning:Regression Focussing

- GraphPlan-Algorithm: (short form)

```
loop
        -grow the planning graph
        -forward two levels
        -do solution extraction
repeat
```



- Instead, backward

```
loop
      -grow the planning graph              two levels
      -augment it by going forward
      -from the intersection of its leftmost fringe with
        the initial state
         do solution extraction in the normal way
repeat
```