CSC2542
Planning-Graph Techniques

Sheila McIlraith
Department of Computer Science
University of Toronto
Fall 2010

Administrative Announcements

- **Tutorial Time:** If you're taking the course for credit, please (re)vist the doodle poll and see whether you can work towards finding a time when we can all meet. We're at an impass!
- I will be posting a schedule with project milestone dates and the due date for the assignment.
- The lecture in 2 weeks will be given by our TA, Christian Muise.
- Suggested readings for next week:
  - Part III introduction of GNT
  - Chapter 9 of GNT
  - A review paper that I will post on our web page.
- Other Issues?

END of
Administrative Announcements

Acknowledgements

A number of the slides used in this course are modifications of 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/

Other slides are modifications of slides developed by Malte Helmert, Bernhard Nebel, and Jussi Rintanen.

I have also used some material prepared by Dan Weld, P@trick Haslum and Rao Kambhampati.

I would like to gratefully acknowledge the contributions of these researchers, and thank them for generously permitting me to use aspects of their presentation material.

History

- **GraphPlan** (Blum & Furst, 1995) was the first planner to use **planning-graph techniques**
- Before GraphPlan came out, most planning researchers were working on partial order planners (plan space planners *)
  - POP, SNLP, UCPOP, etc.
- GraphPlan caused a sensation because it was so much faster
- Many subsequent planning systems have used ideas from it either directly as close decendants of GraphPlan or by using the Planning Graph representation in some guise to improve the encoding of the planning problem most notably for SAT-based planning.

** Sometimes referred to as "PSP" planners, but "PSP" used for "partial satisfaction planners", nowadays*

History

But most importantly…

- GraphPlan's place in history is secured by its critical role in the development of *reachability heuristics** for heuristic search planners by approximating the search tree rooted at a given state
- Heuristic search planners have dominated the "satisficing planner" track of IPC planning competitions for the last 8 years.

* *Reachability heuristics aim to estimate the cost of a plan between the current search state and a goal state. We will talk about these more in the weeks to come.*

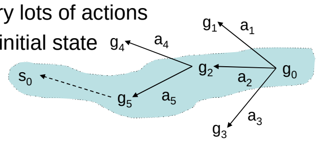
Outline

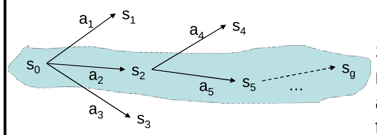
- Motivation
- The Graphplan algorithm
- Planning graphs
  - example
- Mutual exclusion
  - example (continued)
- Solution extraction
  - example (continued)
- Discussion

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





Similarly, a forward search may generate lots of actions that do not reach to the goal

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) **🡸 IMPORTANT**
  - The solutions to the relaxed problem will include 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

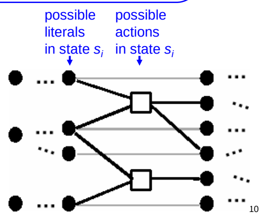
Graphplan

procedure Graphplan:

- for *k* = 0, 1, 2, …
  - *Graph Expansion:*
    - create a “planning graph” that contains *k* “levels”
  - Check whether the planning graph satisfies a necessary (but insufficient) condition for plan existence

relaxed problem

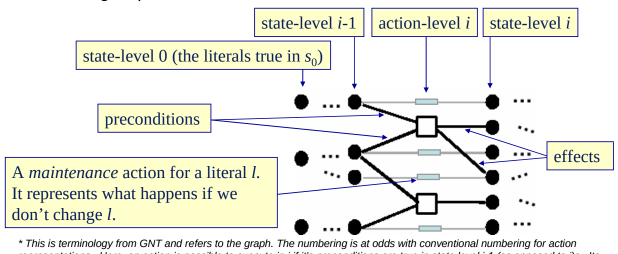
  - If it does, then
    - do *Solution Extraction:*
      - backward search, modified to consider only the actions in the planning graph
      - if we find a solution, then return it


possible literals in state s_i
possible actions in state s_i

The Planning Graph

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


state-level *i*-1
action-level *i*
state-level *i*
state-level 0 (the literals true in s_0)
preconditions
effects
A *maintenance* action for a literal *l*. It represents what happens if we don’t change *l*.

* This is terminology from GNT and refers to the graph. The numbering is at odds with conventional numbering for action representations. Here, an action is possible to execute in i if it’s preconditions are true in state level i-1 (as opposed to i)s. Its effects are reflected in the propositions of state level i (as opposed to i+1)

Example

- Due to Dan Weld, Univ. Washington [Weld, AIM-09]
- Suppose you want to prepare dinner as a surprise for your sweetheart (who is asleep)

  s_0 = {garbage, cleanHands, quiet}

  g = {dinner, present, ¬garbage}

Action	Preconditions	Effects
cook()	cleanHands	dinner
wrap()	quiet	present
carry()	*none*	¬garbage, ¬cleanHands
dolly()	*none*	¬garbage, ¬quiet

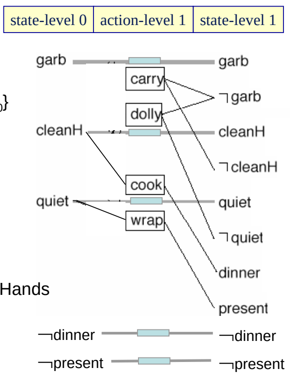
Also have the maintenance actions: one for each literal

Example (continued)

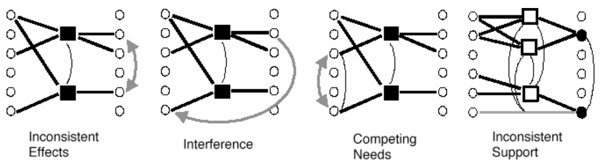
- state-level 0:
  {all atoms in s_0} U
  {negations of all atoms not in s_0}
- action-level 1:
  {all actions whose preconditions are satisfied and non-mutex in s_0}
- state-level 1:
  {all effects of all of the actions in action-level 1 (including maintenance actions)}

Action	Preconditions	Effects
cook()	cleanHands	dinner
wrap()	quiet	present
carry()	*none*	¬garbage, ¬cleanHands
dolly()	*none*	¬garbage, ¬quiet

Also have the maintenance actions



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**

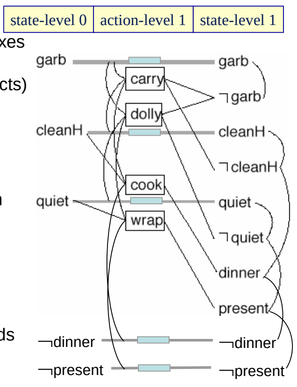
Recursive propagation of mutexes

Example (continued)

- Augment the graph to indicate mutexes
- *carry* is mutex with the maintenance action for *garbage* (inconsistent effects)
- *dolly* is mutex with *wrap*
  - interference
- ¬*quiet* is mutex with *present*
  - inconsistent support
- each of *cook* and *wrap* is mutex with a maintenance operation

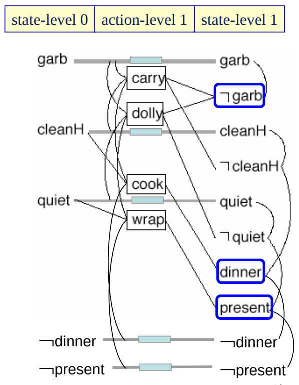
Action	Preconditions	Effects
cook()	cleanHands	dinner
wrap()	quiet	present
carry()	*none*	¬garbage, ¬cleanHands
dolly()	*none*	¬garbage, ¬quiet

Also have the maintenance actions



Example (continued)

- Check to see whether there's a possible solution
- Recall that the goal is
  {¬*garbage, dinner, present*}
- Note that in state-level 1,
  - All of them are there
  - None are mutex with each other
- Thus, there's a chance that a plan exists
- Try to find it
  - Solution extraction



Recall what the algorithm does

procedure Graphplan:
- for *k* = 0, 1, 2, …
  - *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

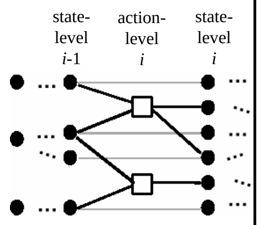
Solution Extraction

The set of goals we are trying to achieve

The level of the state s_i

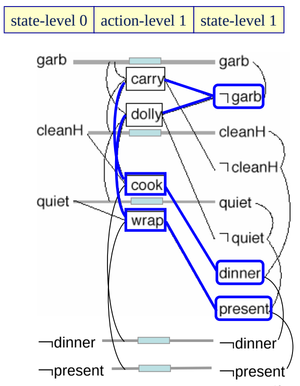
A real action or a maintenance action

```
procedure Solution-extraction($g$,$i$)
    if $i$=0 then return the solution
    for each literal $l$ in $g$
        nondeterministically choose an action
          to use in state $s_{i-1}$ to achieve $l$
    if any pair of chosen actions are mutex
        then backtrack
    $g'$ := {the preconditions of
             the chosen actions}
    Solution-extraction($g'$, $i$–1)
end Solution-extraction
```



Example (continued)

- Recall that the goal is
  {¬*garbage, dinner, present*}
- Two sets of actions for the goals at state-level 1
- Neither of them works
  - Both sets contain actions that are mutex

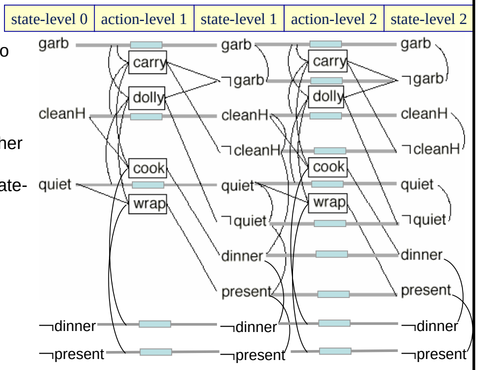


Recall what the algorithm does

procedure Graphplan:
- for *k* = 0, 1, 2, …
  - *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

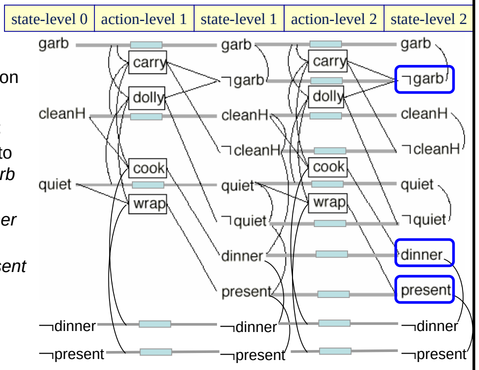
Example (continued)

- Go back and do more graph expansion
- Generate another action-level and another state-level



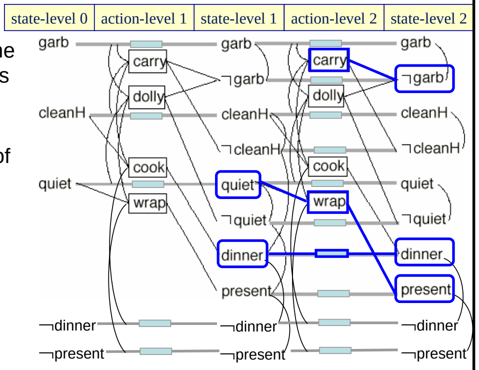
Example (continued)

- Solution extraction
- Twelve combos:
  - Three ways to achieve ¬*garb*
  - Two ways to achieve *dinner*
  - Two ways to achieve *present*



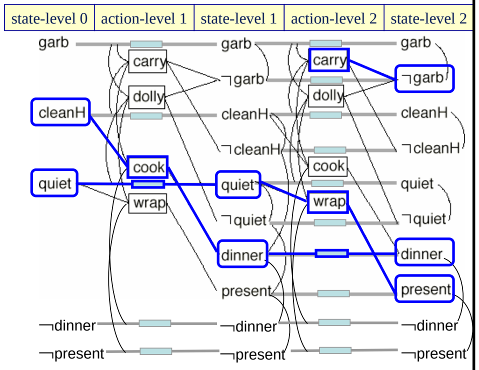
Example (continued)

- Several of the combinations look OK at level 2
- Here's one of them



Example (continued)

- Call Solution-Extraction recursively at level 2
- **It succeeds**
- Solution whose *parallel length* is 2



Observation

- The solution is a sequence of *sets of actions* (as opposed to simply a sequence of actions)
- To generate a sequential plan, the solution can be serialized (in a variety of ways)

Properties of GraphPlan

- Graphplan is sound and complete
  - If Graphplan returns a plan, then it is a solution to the planning pblm.
  - If solutions exist, then Graphplan will return one of them.
- The size of the planning graph Graphplan generates is polynomial in the size of the planning problems.
- Solutions extraction is still exponential in the worst case.

For many problems, Graph Expansion dominates problem solving time.

- The planning algorithm always terminates
  - There is a fixpoint on the number of levels of the planning graphs such that the algorithm either generates a solution or returns failure

Further Analysis

- (+) The backward-search part of Graphplan—which is the hard part—will only look at the actions in the planning graph – a smaller search space than the original one.
- (-) 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

Optimizations and Extensions

Optimizations to the original Graphplan model:

- Improvements to solution extraction (e.g., forward checking, memoization, EBL)
- Improvements to graph expansion (e.g., closed-world assumption, compilation of action schemata w/ type analysis, in-place graph expansion)

Most of the **extensions** relate to language extensions:

- E.g., Universal quantification, conditional effects, negated preconditions and goals, …
- Addressed by compilation ➔large increases in problem size, or
- Addressed by changing (complicating) the expansion, mutex determination and extraction

See [Weld, AIM-99] posted on our web page.

Optimizations and Extensions

Optimizations to the original Graphplan model:

- Improvements to solution extraction (e.g., forward checking, memoization, EBL)
- Improvements to graph expansion (e.g., closed-world assumption, compilation of action schemata w/ type analysis, in-place graph expansion)

Most of the **extensions** relate to language extensions:

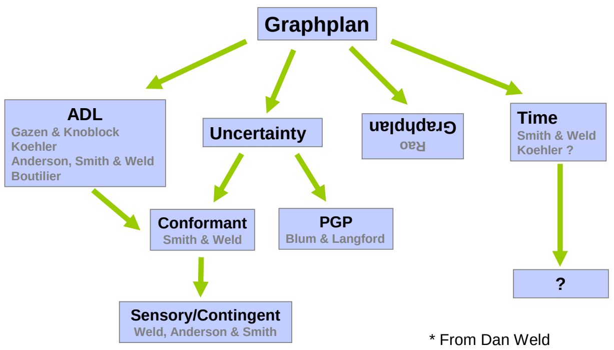
- E.g., Universal quantification, conditional effects, negated preconditions and goals, …
- Addressed by compilation ➔large increases in problem size, or
- Addressed by changing (complicating) the expansion, mutex determination and extraction

See [Weld, AIM-99] posted on our web page.

Graphplan and Its Descendants

- **Graphplan** (Blum & Furst,1995) – C implementation
- **IPP** (Koehler et al. 1997) – highly optimized C implementation, extended to handle universal quantification and conditional effects
- **STAN** (Long & Fox, 1998)- highly optimized C, implementation – uses in-place graph rep'n and performs sophisticated type analysis
- **SGP** (Weld et al., 1998) – Lisp implementation – extended to handl universal quantification, conditional effects, and uncertainty

Perverting Graphplan*



* From Dan Weld