Plan Generation
Classical Planning

Manuela Veloso

Carnegie Mellon University
School of Computer Science

15-887 Planning, Execution, and Learning – Fall 2016

Outline

- What is a ***State and Goal***
- What is an ***Action***
- **What is a *Plan***
- ***Finding*** a Plan

What is a Plan?

- ***Sequence*** of Instantiated Actions
- ***Partial Order*** of Instantiated Actions
- ***Set*** of Instantiated Actions
- ***Policy***
  - Mapping from states to actions

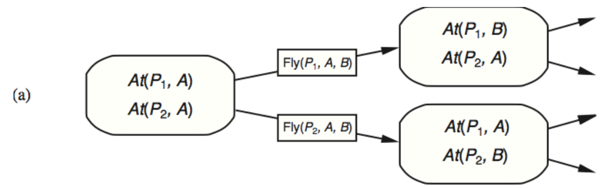
Increasing Generality ↓

Outline

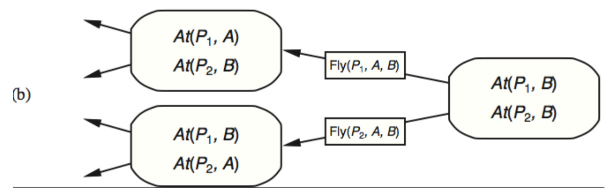
- What is a ***State and Goal***
- What is an ***Action***
- What is a ***Plan***
- ***Finding*** **a Plan**

Planning Algorithms

- Progression: Forward state-space search



- Regression: Backward state-space search



Finding a Plan – Plan Generation

- ***Backtracking Search*** Through a ***Search Space***
  - How to conduct the search
  - How to represent the search space
  - How to evaluate the solutions
- Non-Deterministic ***Choice Points*** Determine Backtracking
  - Choice of actions
  - Choice of variable bindings
  - Choice of temporal orderings
  - Choice of subgoals to work on

Properties of Planning Algorithms

- **Soundness**
  - A planning algorithm is ***sound*** if all solutions are *legal* plans
    - All preconditions, goals, and any additional constraints are satisfied
- **Completeness**
  - A planning algorithm is ***complete*** if a solution can be found whenever one actually exists
  - A planning algorithm is ***strictly complete*** if all solutions are included in the search space
- **Optimality**
  - A planning algorithm is ***optimal*** if it maximizes a predefined measure of plan quality

Linear Planning

- Basic Idea – Goal **stack**
  - *Work **on one goal until completely solved** before moving on to the next goal*

Means-Ends Analysis

- Basic Idea
  - ***Search by reducing the difference between the state and the goals***
  - What ***means*** (operators) are available to achieve the desired ***ends*** (goal)

Means-ends Analysis in Linear Planning

(*Newell and Simon 60s*)

GPS Algorithm (*state*, *goals, plan*)

- If *goals* $\subseteq$ *state*, then return (*state,plan)*
- **Choose** a difference $d \in$ *goals* between *state* and *goals*
- **Choose** an operator o to reduce the difference d
- If no applicable operators, then return *False*
- *(state,plan)* = **GPS** (*state*, preconditions(o), *plan*)
- If *state*, then return **GPS** (apply (o, *state*), *goals, [plan,o]*)

Initial call: GPS (initial-state, initial-goals, [])

GPS Blocks-World Example

1. Search Stack

- On(A, C) On(C, B)

State

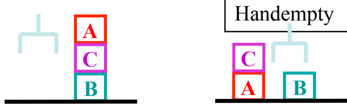
Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

2. Search Stack

- On(A, C) On(C, B)
- On(A, C)
- On(C, B)

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty



Goal **Initial State**

3. Search Stack

- On(A, C) On(C, B)
- On(A, C)
- Put_Block(C, B)
- Holding(C) Clear(B)

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

4. Search Stack

- On(A, C) On(C, B)
- On(A, C)
- Put_Block(C, B)
- Holding(C) Clear(B)
- Holding(C)
- Clear(B)

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

GPS Blocks-World Example

5. Search Stack

- On(A, C) On(C, B)
- On(A, C)
- Put_Block(C, B)
- Holding(C) Clear(B)
- Holding(C)

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

6. Search Stack

- On(A, C) On(C, B)
- On(A, C)
- Put_Block(C, B)
- Holding(C) Clear(B)
- Pick_Block(C)
- Handempty Clear(C) On(C, ?b)

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

7. Search Stack

- On(A, C) On(C, B)
- On(A, C)
- Put_Block(C, B)
- Holding(C) Clear(B)
- Pick_Block(C)

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

8. Search Stack

- On(A, C) On(C, B)
- On(A, C)
- Put_Block(C, B)
- Holding(C) Clear(B)

[Pick_Block(C)]

State

Clear(B)
Clear(C)
On(A, Table)
On(B, Table)
Holding(C)
Clear(A)

GPS Blocks-World Example

9. Search Stack

- On(A, C) On(C, B)
- On(A, C)
- Put_Block(C, B)

State

Clear(B)
Clear(C)
On(A, Table)
On(B, Table)
Holding(C)
Clear(A)

[Pick_Block(C)]

10. Search Stack

- On(A, C) On(C, B)
- On(A, C)

State

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

[Pick_Block(C); Put_Block(C, B)]

11. Search Stack

- On(A, C) On(C, B)
- Put_Block(A, C)
- Holding(A) Clear(C)

State

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

[Pick_Block(C)
Put_Block(C, B)]

12. Search Stack

- On(A, C) On(C, B)
- Put_Block(A, C)
- Holding(A) Clear(C)
- Holding(A)
- Clear(C)

State

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

[Pick_Block(C)
Put_Block(C, B)]

GPS Blocks-World Example

13. Search Stack

- On(A, C) On(C, B)
- Put_Block(A, C)
- Holding(A) Clear(C)
- Holding(A)

State

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

[Pick_Block(C);
Put_Block(C, B)]

14. Search Stack

- On(A, C) On(C, B)
- Put_Block(A, C)
- Holding(A) Clear(C)
- Pick_Table(A)
- Handempty Clear(A) On(A, Table)

State

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

[Pick_Block(C); Put_Block(C, B)]

15. Search Stack

- On(A, C) On(C, B)
- Put_Block(A, C)
- Holding(A) Clear(C)
- Pick_Table(A)

State

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

[Pick_Block(C);
Put_Block(C, B)]

16. Search Stack

- On(A, C) On(C, B)
- Put_Block(A, C)
- Holding(A) Clear(C)

State

Clear(C)
On(B, Table)
Clear(A)
On(C, B)
Holding(A)

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A)]

GPS Blocks-World Example

17. Search Stack

On(A, C) On(C, B)
Put_Block(A, C)

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A)]

State

Clear(C)
On(B, Table)
Clear(A)
On(C, B)
Holding(A)

18. Search Stack

On(A, C) On(C, B)

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A);
Put_Block(A, C)]

State

On(B, Table)
Clear(A)
On(C, B)
Handempty
On(A, C)

19. Search Stack

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A);
Put_Block(A, C)]

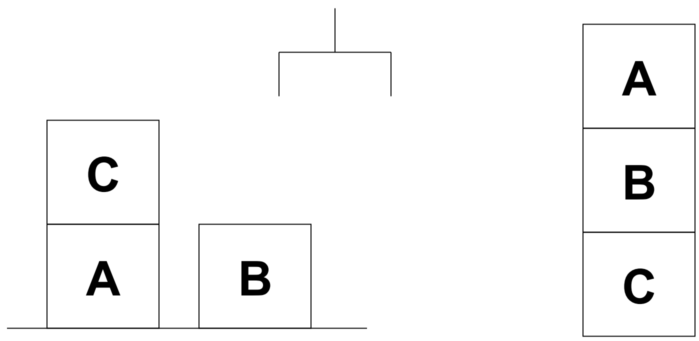
State

On(B, Table)
Clear(A)
On(C, B)
Handempty
On(A, C)

Linear Planning with MEA

- Sound?
- Optimal?
- Complete?

The Sussman Anomaly



4-Action Blocks World Domain

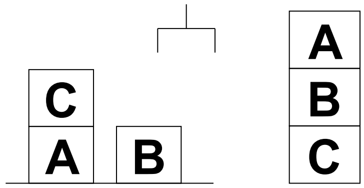
Pickup (?b)
Pre: (handempty)
(clear ?b)
(on-table ?b)
Add: (holding ?b)
Delete: (handempty)
(on-table ?b)
(clear ?b)

Putdown (?b)
Pre: (holding ?b)
Add: (handempty)
(on-table ?b)
Delete: (holding ?b)

Unstack (?a, ?b)
Pre: (handempty)
(clear ?a) (on ?a ?b)
Add: (holding ?a) (clear ?b)
Delete: (handempty)
(on ?a ?b) (clear ?a)

Stack (?a, ?b)
Pre: (holding ?a) (clear ?b)
Add: (handempty)
(on ?a ?b)
Delete: (holding ?a)
(clear ?b)

The Sussman Anomaly



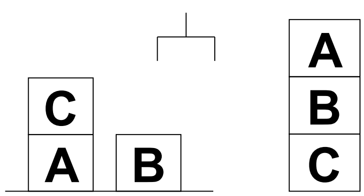
Linear Solution:

- *(on B C)*
  - ***Pickup (B)***
  - ***Stack (B, C)***
- *(on A B)*
  - ***Unstack (B, C)***
  - ***Putdown (B)***
  - ***Unstack (C, A)***
  - ***Putdown (C)***
  - ***Stack (A, B)***
- *(on B C)*
  - ***Unstack (A, B)***
  - ***Putdown (A)***
  - ***Pickup (B)***
  - ***Stack (B, C)***
- *(on A B)*
  - ***Pickup (A)***
  - ***Stack (A,B)***

Linear Solution:

- *(on A B)*
  - ***Unstack (C, A)***
  - ***Putdown (C)***
  - ***Stack (A, B)***
- *(on B C)*
  - ***Unstack (A, B)***
  - ***Putdown (A)***
  - ***Pickup (B)***
  - ***Stack (B, C)***
- *(on A B)*
  - ***Pickup (A)***
  - ***Stack (A,B)***

NonLinear Solution – Optimal



NonLinear Solution:

- *(on A B)*
  - ***Unstack (C, A)***
  - ***Putdown (C)***
- *(on B C)*
  - ***Pickup (B)***
  - ***Stack (B, C)***
- *(on A B)*
  - ***Pickup (A)***
  - ***Stack (A, B)***

Linear Planning – Goal Stack

- **Advantages**
  – Reduced search space, since goals are solved one at a time, and not all possible goal orderings are considered
  – Advantageous if goals are (mainly) independent
  – Linear planning is ***sound***

- **Disadvantages**
  – Linear planning may produce ***suboptimal*** solutions (based on the number of operators in the plan)
  – Planner's efficiency is sensitive to goal orderings
    - Control knowledge for the “right” ordering
    - Random restarts
    - Iterative deepening

- Completeness?

Example: One-Way Rocket (Veloso 89)

```
(OPERATOR LOAD-ROCKET
 :preconds
  ?roc ROCKET
  ?obj OBJECT
  ?loc LOCATION
 (and (at ?obj ?loc)
      (at ?roc ?loc))
 :effects
  add (inside ?obj ?roc)
  del (at ?obj ?loc))
```

```
(OPERATOR UNLOAD-ROCKET
 :preconds
  ?roc ROCKET
  ?obj OBJECT
  ?loc LOCATION
 (and (inside ?obj ?roc)
      (at ?roc ?loc))
 :effects
  add (at ?obj ?loc)
  del (inside ?obj ?roc))
```

```
(OPERATOR MOVE-ROCKET
 :preconds
  ?roc ROCKET
  ?from-l LOCATION
  ?to-l LOCATION
 (and (at ?roc ?from-l)
      (has-fuel ?roc))
 :effects
  add (at ?roc ?to-l)
  del (at ?roc ?from-l)
  del (has-fuel ?roc))
```

Incompleteness of Linear Planning

```
Initial state:                   Goal statement:
          (at obj1 locA)                (and
          (at obj2 locA)                  (at obj1 locB)
          (at ROCKET locA)                (at obj2 locB))
          (has-fuel ROCKET)
```

Goal	*Plan*
(at obj1 locB)	(LOAD-ROCKET obj1 locA) (MOVE-ROCKET) (UNLOAD-ROCKET obj1 locB)
(at obj2 locB)	*failure*

State-Space Nonlinear Planning

Extend linear planning:

- From **stack** to **set** of goals.
- Include in the search space all possible interleaving of goals.

State-space nonlinear planning is **complete**.

Goal	*Plan*
(at obj1 locB)	(LOAD-ROCKET obj1 locA)
(at obj2 locB)	(LOAD-ROCKET obj2 locA)
(at obj1 locB)	(MOVE-ROCKET) (UNLOAD-ROCKET obj1 locB)
(at obj2 locB)	(UNLOAD-ROCKET obj1 locB)

Prodigy Planner

- Extension to GPS
  - Set of goals, instead of stack of goals
  - Means-ends analysis for selection of “pending goals”
  - Choice point for applying an operator when applicable and continue backward-chaining (subgoaling)

Prodigy4.0 (Veloso et al. 90)

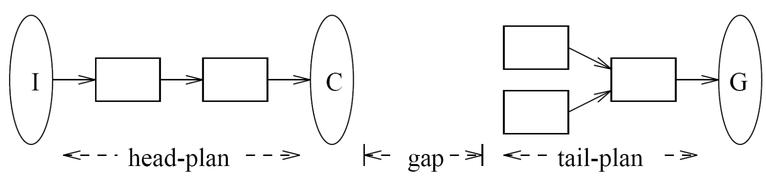
1. Terminate if the goal statement is satisfied in the current state. Initially the set of applicable *relevant* operators is empty.
2. Compute the **SET** of *pending goals G*, and the **SET** of *applicable relevant operators A*.
   - A goal is pending if it is a precondition, not satisfied in the current state, of a *relevant operator* already in the plan.
   - A relevant operator is *applicable* when all its preconditions are satisfied in the state.

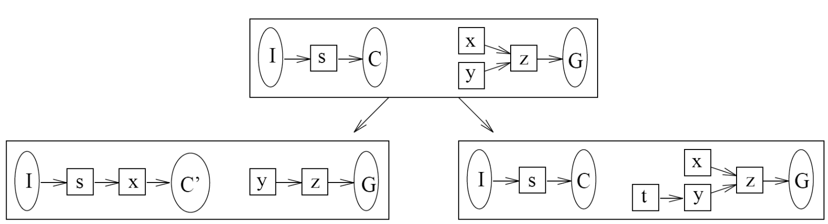
1. Choose a pending goal G in *G* or choose a relevant applicable operator A in *A*.

Prodigy4.0 Planning Algorithm

4. If the pending goal G has been chosen, then
   - *Expand goal* G,
     i.e., get the set O of *relevant instantiated operators* that could achieve G,
   - Choose an operator O from O, as a *relevant operator* for goal G.
   - Go to step 1.
5. If a relevant operator A has been selected as directly applicable, then
   - *Apply* A,
   - Go to step 1.

Prodigy4.0 – Search Representation





Applying an operator (moving it to the head) Adding an operator to the tail-plan

Why is Planning Hard?

Planning involves a complex search:

- Alternative operators to achieve a goal
- Multiple goals that interact
- Solution optimality, quality
- Planning efficiency, soundness, completeness

Many Issues in Planning

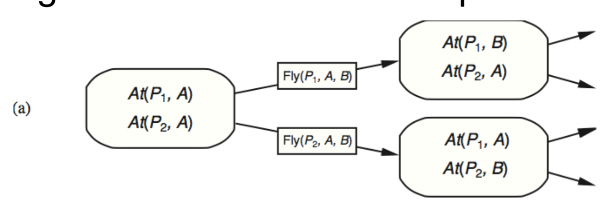
- State representation
  - The frame problem
  - The "choice" of predicates
    - On-table (x), On (x, table), On-table-A, On-table-B,…
- Action representation
  - Many alternative definitions
  - Reduce to "needed" definition
  - Conditional effects
  - Uncertainty
  - Quantification
  - Functions
- Generation – planning algorithm(S)

Summary

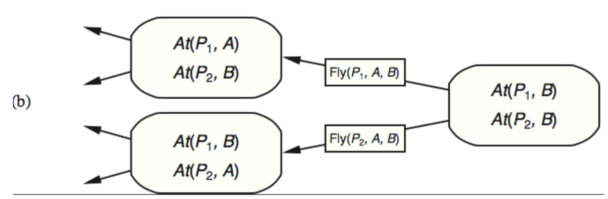
- **Planning:** selecting one sequence of actions (operators) that transform (apply to) an initial state to a final state where the goal statement is true.
- **Means-ends analysis:** identify and reduce, as soon as possible, *differences* between state and goals.
- **Linear planning:** backward chaining with means-ends analysis using a stack of goals - potentially efficient, possibly unoptimal, incomplete; GPS, STRIPS.
- **Nonlinear planning with means-ends analysis:** backward chaining using a set of goals; reason about *when* "to reduce the differences;" Prodigy4.0.

Planning Algorithms

- Progression: Forward state-space search

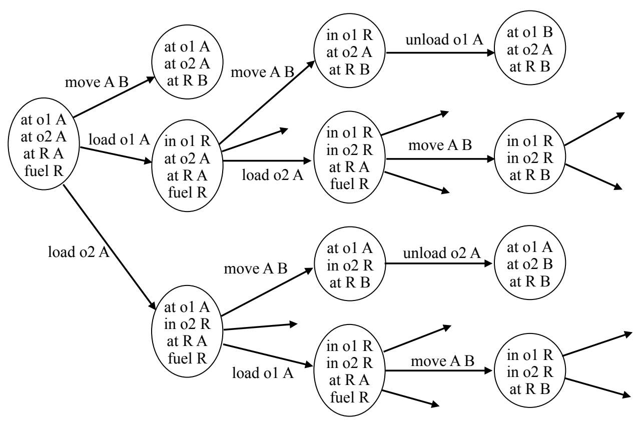


- Regression: Backward state-space search



Planning Graph – Forward Expansion

- State reachability – "until" goal
  - Can find ***all*** goals reachable from initial state
  - Exponential in time and memory

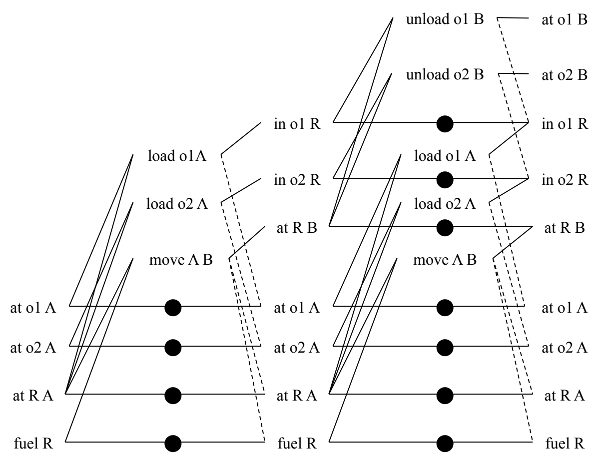


Graphplan

Blum & Furst 95

- Preprocessing before engaging in search.
- Forward search combined with backward search.
- Construct a *planning graph* to reveal constraints
- Two stages:
  - Extend: One time step in the planning graph.
  - Search: Find a valid plan in the planning graph.
- Graphplan finds a plan or proves that no plan has fewer "time steps."

Plan Graph
One-Way Rocket Example



Extending a Planning Graph - Actions

- To create an action-level *i*:
  - Add each instantiated operator, for which all of its preconditions are present at proposition-level *i* AND *no two of its preconditions are exclusive*.
  - Add all the no-op actions.
- Determine the exclusive actions.

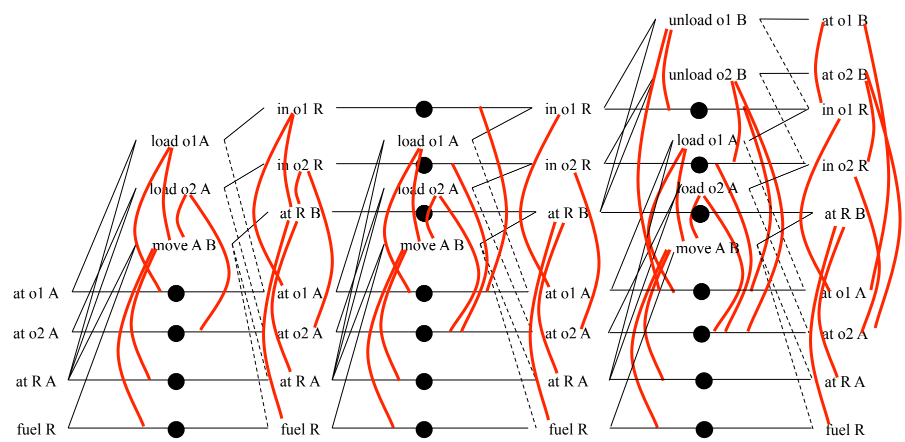
Extending a Planning Graph – Propositions

- To create a proposition-level $i + 1$:
  - Add all the effects of the inserted actions at action-level i - distinguishing add and delete effects.
- Determine the exclusive actions.

Planning Graphs

- A literal may exist at level $i + 1$ if it is an Add-Effect of some action in level i.
- Two propositions p and q are *exclusive* in a proposition-level if ALL actions that add p are exclusive of ALL actions that add q.
- Actions A and B are *exclusive* at action-level i, if:
  - Interference: A (or B) deletes a precondition or an Add-Effect of B (or A).
  - Competing Needs: p is a precondition of A and q is a precondition of B, and p and q are exclusive in proposition-level i - 1.

Mutex Exclusivity Relations
One-Way Rocket Example



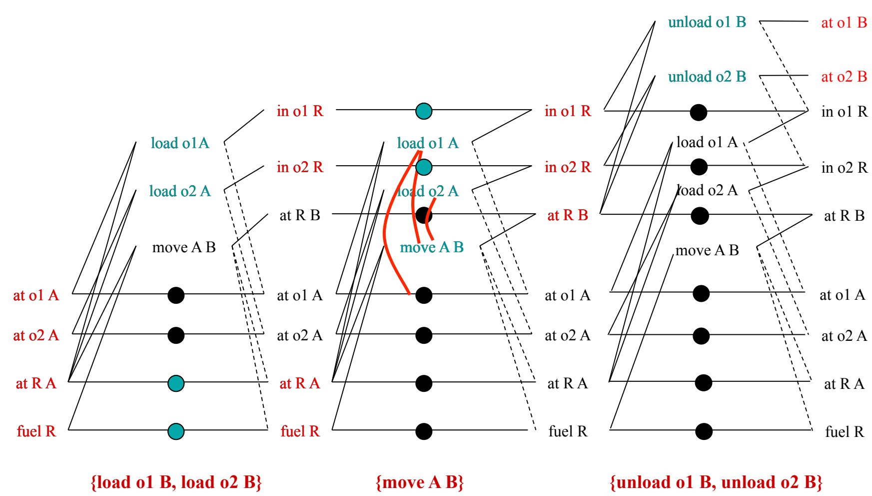
Exclusivity Examples

- Exclusive Actions: (Move A B) deletes a precondition of (Load o1 A). Therefore exclusive (existence of threats).
- Exclusive Propositions: (at R A) and (at R B) at time 2 are exclusive. (at R A) is added by a no-op and (at R B) is added by (Move A B) and no-op and (Move A B) are exclusive actions.
- Exclusive Actions: Then (Load o1 A) and (Load o2 B) are exclusive because (at R A) and (at R B) are exclusive.
- Propositions can be exclusive in some time step and not in others: If (at o1 A) and (at R A) at time 1, then (in o1 A) and (at R B) are exclusive at time 2, but not at time 3.

Searching a Planning Graph

- Level-by-level backward-chaining approach to use the exclusivity constraints.
- Given a set of goals at time *t*, identify all the sets of actions (including no-ops) at time *t* - 1 who add those goals and are not exclusive. The preconditions of these actions are new goals for *t* - 1.

Searching a Planning Graph



Recursive Search

- For each goal at time *t* in some arbitrary order:
  - Select some action at time *t* - 1 that achieves that goal and it is not exclusive with any other action already selected.
  - Do this recursively for all the goals at time *t* - do not add new action, but use the ones already selected if they add another goal.
  - If recursion returns failure, then select a different action.
- The new goal set is the set of all the preconditions of the selected actions.

Enhancements

- Forward-checking - for the goals ahead, check if all the actions that add it are exclusive with the selected action.
- Memoization - when a set of goals is not solvable at some time *t*, then this is recorded and hashed. If back at time *t*, the hash table is checked and search proceeds backing up right away.

Planning as Satisfiability

- One interpretation: ``first-order deductive theorem-proving does not scale well.' ‘
- One solution: ``propositional satisfiability' ‘
- Uniform clausal representation for goals and operators.
- Stochastic local search is a powerful technique for planning.