CMSC 671
Fall 2010

Tue 10/14/10

Classical Planning

Chapter 10

Prof. Laura Zavala, laura.zavala@umbc.edu, ITE 373, 410-455-8775

Today's class

- What is planning?
- Representation
  - PDDL
- Approaches to planning
  - GPS /STRIPS
  - Situation calculus formalism
  - State space planning
  - Graph-based planning

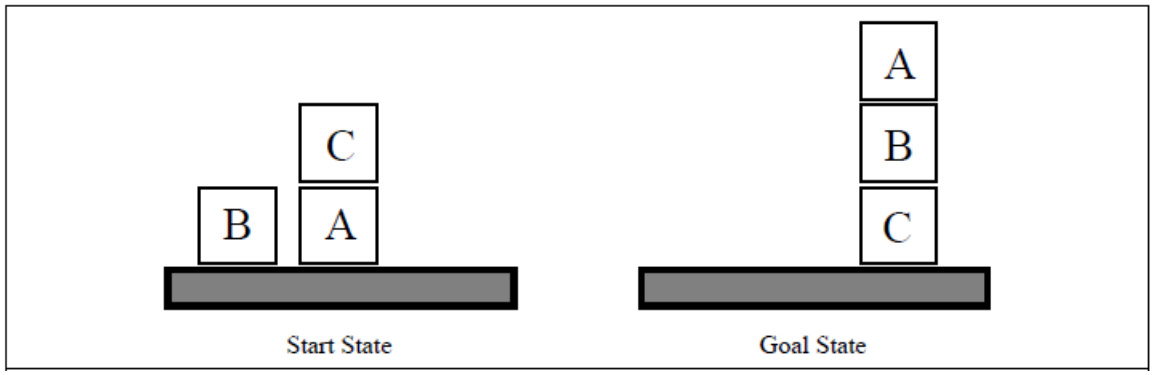
Planning problem

- Find a **sequence of actions** that achieves a given **goal** when executed from a given **initial world state**. That is, given
  - a set of operator descriptions (defining the possible primitive actions by the agent),
  - an initial state description, and
  - a goal state description or predicate,

  compute a plan, which is
  - a sequence of operator instances, such that executing them in the initial state will change the world to a state satisfying the goal-state description.
- Goals are usually specified as a conjunction of goals to be achieved

Planning

- Planning is finding and choosing a sequence (or a "**program**") of actions to achieve goals.
- Unlike theorem prover, not seeking whether the goal is true, but is there a sequence of actions to attain it



- Move C to the table
- Put B on top of C
- Put A on top of B

Planning vs. problem solving

- Planning and problem solving methods can often solve the same sorts of problems

Planning vs. problem solving

- Problem solving deals with atomic representations of states

- Planning is more powerful because of the representations and methods used

Planning vs. problem solving

- States, goals, and actions are decomposed into sets of sentences (usually in first-order logic or a subset of it)
- Search often proceeds through ***plan space*** rather than ***state space*** (though there are also state-space planners)
- Subgoals can be planned independently, reducing the complexity of the planning problem

Typical assumptions

- **Atomic time**: Each action is indivisible
- **No concurrent actions** are allowed (though actions do not need to be ordered with respect to each other in the plan)
- **Deterministic actions**: The result of actions are completely determined—there is no uncertainty in their effects
- Agent is the **sole cause of change** in the world
- Agent is **omniscient**: Has complete knowledge of the state of the world
- **Database semantics**: close world assumption and unique names assumption

Representation

- General planning algorithms require a way to represent states, actions and goals
- STRIPS, PDDL are languages based on propositional or first-order logic

PDDL: Planning Domain Definition Language

- States represented as a conjunction of **ground** literals
  - at(Home) ∧ have(Milk) ∧ have(bananas) ...
- Actions have
  - **Action Name and Parameter List**
  - **Precondition** - conjunction of positive literals
  - **Effect** - conjunction of positive or negative literals which describe how situation changes when operator is applied

Action(Fly(p,from, to),
PRECOND: At(p,from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
EFFECT: ¬AT(p,from) ∧ At(p,to))

Actions are specified in terms of what changes; everything that stays the same is left unmentioned

Applicability of Actions

- An action is *applicable* in any state that satisfies the precondition.
- For FO action schema applicability involves a substitution θ for the variables in the PRECOND.

  $At(P1,JFK) \wedge At(P2,SFO) \wedge Plane(P1) \wedge Plane(P2) \wedge Airport(JFK) \wedge Airport(SFO)$

  Satisfies : $At(p,from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$

  With $\theta = \{p/P1, from/JFK, to/SFO\}$

  Thus the action is applicable.

Effect of Actions

- Executing action a in state s results in state s', where s' is same as s except
  - Any positive literal *P* in the effect of *a* is added to *s'*
  - Any negative literal ¬*P* is removed from *s'*

    s: At(P1,JFK) ∧ At(P2,SFO)
    ∧ Plane(P1) ∧ Plane(P2) ∧ Airport(JFK) ∧ Airport(SFO)

    EFFECT: ¬AT(p,from) ∧ At(p,to):

    s': At(P1,SFO) ∧ At(P2,SFO)
    ∧ Plane(P1) ∧ Plane(P2) ∧ Airport(JFK) ∧ Airport(SFO)

- assumption: *every literal NOT in the effect remains unchanged*

Blocks world operators

- Here are the classic basic operations for the blocks world:
  - stack(X,Y): put block X on block Y
  - unstack(X,Y): remove block X from block Y
  - pickup(X): pickup block X
  - putdown(X): put block X on the table
- Each action will be represented by:
  - a list of preconditions
  - a list of new facts to be added (add-effects)
  - a list of facts to be removed (delete-effects)
  - optionally, a set of (simple) variable constraints
- For example:

  preconditions(stack(X,Y), [holding(X), clear(Y)])
  deletes(stack(X,Y), [holding(X), clear(Y)]).
  adds(stack(X,Y), [handempty, on(X,Y), clear(X)])
  constraints(stack(X,Y), [X≠Y, Y≠table, X≠table])

Blocks world

The **blocks world** is a micro-world that consists of a table, a set of blocks and a robot hand.

Some domain constraints:

- Only one block can be on another block
- Any number of blocks can be on the table
- The hand can only hold one block

Typical representation:

ontable(a)

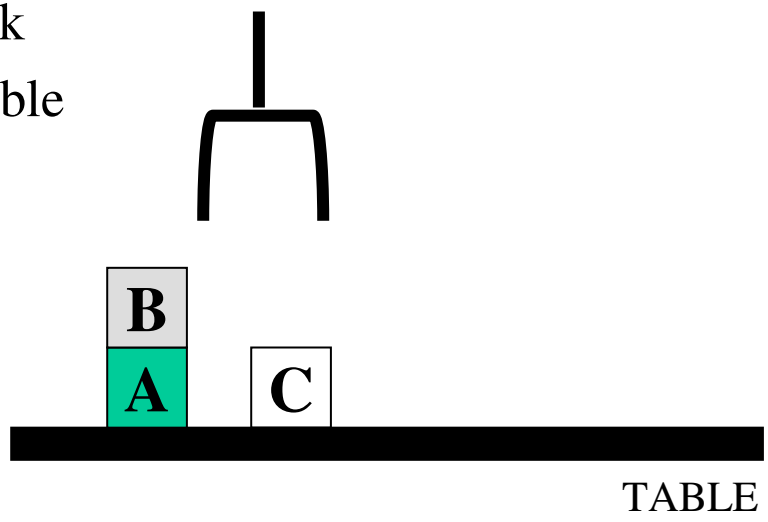
ontable(c)

on(b,a)

handempty

clear(b)

clear(c)



Blocks world operators II

operator(stack(X,Y),
Precond [holding(X), clear(Y)],
Add [handempty, on(X,Y), clear(X)],
Delete [holding(X), clear(Y)],
Constr [X≠Y, Y≠table, X≠table]).

operator(pickup(X),
[ontable(X), clear(X), handempty],
[holding(X)],
[ontable(X), clear(X), handempty],
[X≠table]).

operator(unstack(X,Y),
[on(X,Y), clear(X), handempty],
[holding(X), clear(Y)],
[handempty, clear(X), on(X,Y)],
[X≠Y, Y≠table, X≠table]).

operator(putdown(X),
[holding(X)],
[ontable(X), handempty, clear(X)],
[holding(X)],
[X≠table]).

Major approaches

- GPS / STRIPS /PDDL
- Progression and Regression
- Situation calculus
- Partial-order planning
- Planning with constraints (SATplan, Graphplan)

- Hierarchical decomposition (HTN planning)
- Reactive planning

General Problem Solver

- The General Problem Solver (GPS) system was an early planner (Newell, Shaw, and Simon)
- GPS generated actions that reduced the difference between some state and a goal state
- GPS used Means-Ends Analysis
  - Compare what is given or known with what is desired and select a reasonable thing to do next
  - Use a table of differences to identify procedures to reduce types of differences
- GPS was a state space planner: it operated in the domain of state space problems specified by an initial state, some goal states, and a set of operations

Situation calculus planning

- Intuition: Represent the planning problem using first-order logic
  - Situation calculus lets us reason about changes in the world
  - Use theorem proving to "prove" that a particular sequence of actions, when applied to the situation characterizing the world state, will lead to a desired result

Situation calculus

- **Initial state**: a logical sentence about (situation) S_0

  $At(Home, S_0) \wedge \neg Have(Milk, S_0) \wedge \neg Have(Bananas, S_0) \wedge \neg Have(Drill, S_0)$

- **Goal state**:

  $(\exists s)\ At(Home,s) \wedge Have(Milk,s) \wedge Have(Bananas,s) \wedge Have(Drill,s)$

- **Operators** are descriptions of how the world changes as a result of the agent's actions:

  $\forall(a,s)\ Have(Milk,Result(a,s)) \Leftrightarrow$
  $((a=Buy(Milk) \wedge At(Grocery,s)) \vee (Have(Milk, s) \wedge a \neq Drop(Milk)))$

- Result(a,s) names the situation resulting from executing action a in situation s.

- Action sequences are also useful: Result'(l,s) is the result of executing the list of actions (l) starting in s:

  $(\forall s)\ Result'([],s) = s$

  $(\forall a,p,s)\ Result'([a|p]s) = Result'(p,Result(a,s))$

Situation calculus II

- A solution is a plan that when applied to the initial state yields a situation satisfying the goal query:

  At(Home, Result'(p,S_0))

  $\wedge$ Have(Milk, Result'(p,S_0))

  $\wedge$ Have(Bananas, Result'(p,S_0))

  $\wedge$ Have(Drill, Result'(p,S_0))

- Thus we would expect a plan (i.e., variable assignment through unification) such as:

  p = [Go(Grocery), Buy(Milk), Buy(Bananas), Go(HardwareStore), Buy(Drill), Go(Home)]

Situation calculus: Blocks world

- A situation calculus rule for the blocks world:
  - Clear (X, Result(A,S)) ↔
    [Clear (X, S) ∧
    (¬(A=Stack(Y,X) ∨ A=Pickup(X))
    ∨ (A=Stack(Y,X) ∧ ¬(holding(Y,S))
    ∨ (A=Pickup(X) ∧ ¬(handempty(S) ∧ ontable(X,S) ∧ clear(X,S))))]
    ∨ [A=Stack(X,Y) ∧ holding(X,S) ∧ clear(Y,S)]
    ∨ [A=Unstack(Y,X) ∧ on(Y,X,S) ∧ clear(Y,S) ∧ handempty(S)]
    ∨ [A=Putdown(X) ∧ holding(X,S)]

- English translation: A block is clear if (a) in the previous state it was clear and we didn't pick it up or stack something on it successfully, or (b) we stacked it on something else successfully, or (c) something was on it that we unstacked successfully, or (d) we were holding it and we put it down.

Situation calculus planning: Analysis

- This is fine in theory, but remember that problem solving (search) is exponential in the worst case
- Also, resolution theorem proving only finds *a* proof (plan), not necessarily a good plan
- So we restrict the language and use a special-purpose algorithm (a planner) rather than general theorem prover

STRIPS planning

- STRIPS maintains two additional data structures:
  - **State List** - all currently true predicates.
  - **Goal Stack** - a push-down stack of goals to be solved, with current goal on top of stack.
- If current goal is not satisfied by present state, examine add lists of operators, and push operator and preconditions list on stack. (Subgoals)
- When a current goal is satisfied, POP it from stack.
- When an operator is on top of the stack, record the application of that operator in the plan sequence and use the operator's add and delete lists to update the current state.

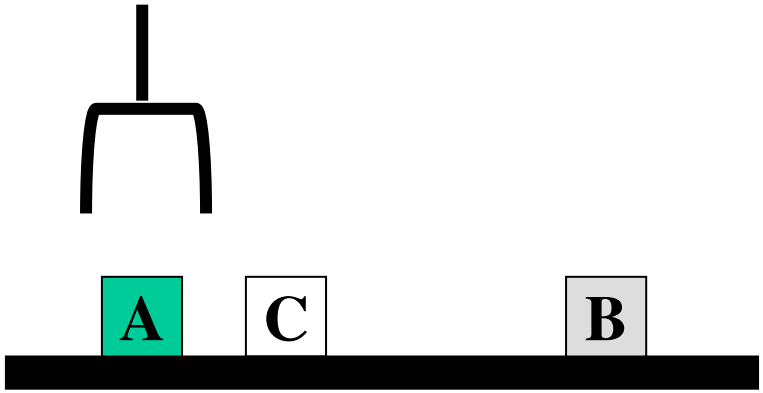
Typical BW planning problem

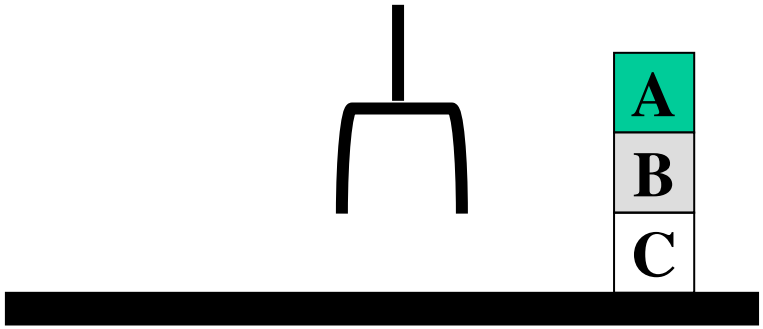
Initial state:

- clear(a)
- clear(b)
- clear(c)
- ontable(a)
- ontable(b)
- ontable(c)
- handempty

Goal:

- on(b,c)
- on(a,b)
- ontable(c)





A plan:

- pickup(b)
- stack(b,c)
- pickup(a)
- stack(a,b)

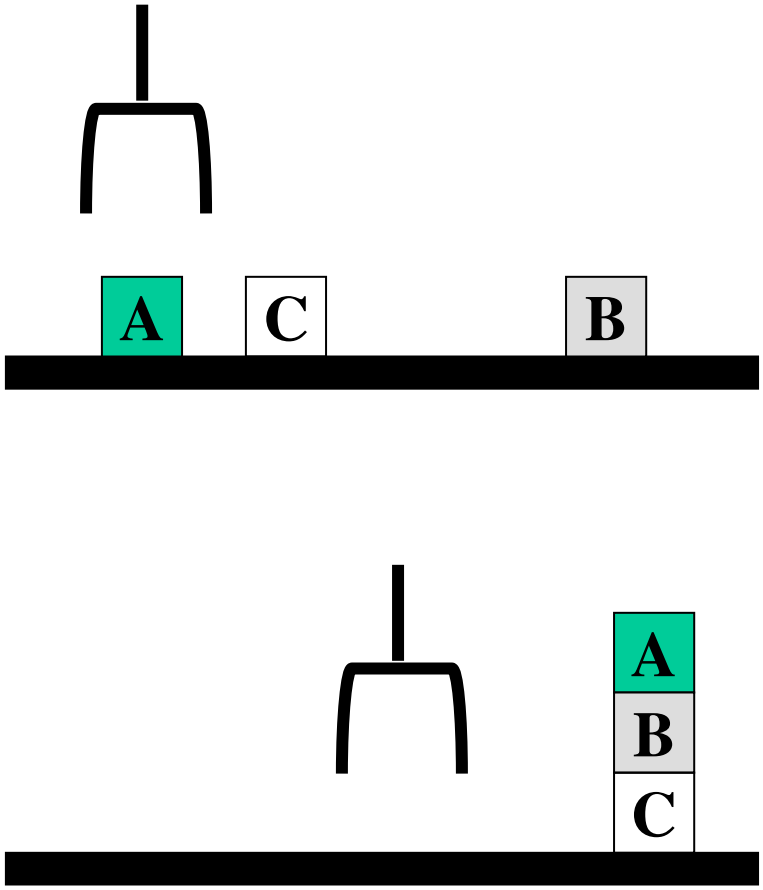
Another BW planning problem

Initial state:

- clear(a)
- clear(b)
- clear(c)
- ontable(a)
- ontable(b)
- ontable(c)
- handempty

Goal:

- on(a,b)
- on(b,c)
- ontable(c)

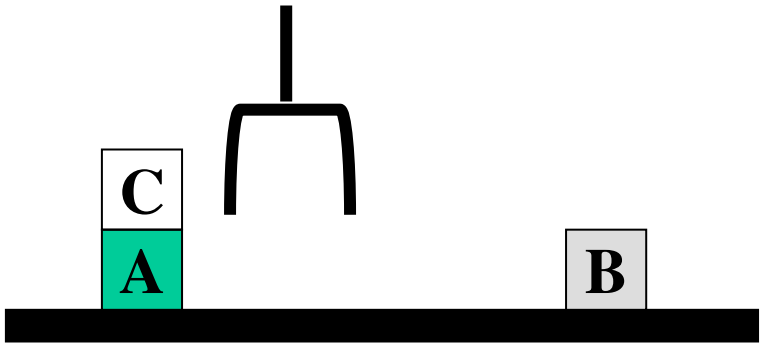


A plan:

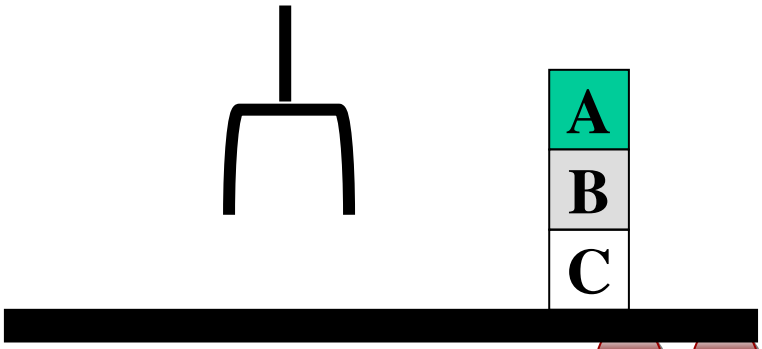
- pickup(a)
- stack(a,b)
- unstack(a,b)
- putdown(a)
- pickup(b)
- stack(b,c)
- pickup(a)
- stack(a,b)

Goal interaction

- Simple planning algorithms assume that the goals to be achieved are independent
  - Each can be solved separately and then the solutions concatenated
- This planning problem, called the "Sussman Anomaly," is the classic example of the goal interaction problem:
  - Solving on(A,B) first (by doing unstack(C,A), stack(A,B) will be undone when solving the second goal on(B,C) (by doing unstack(A,B), stack(B,C)).
  - Solving on(B,C) first will be undone when solving on(A,B)
- Classic STRIPS could not handle this, although minor modifications can get it to do simple cases



Initial state



Goal state

Sussman Anomaly

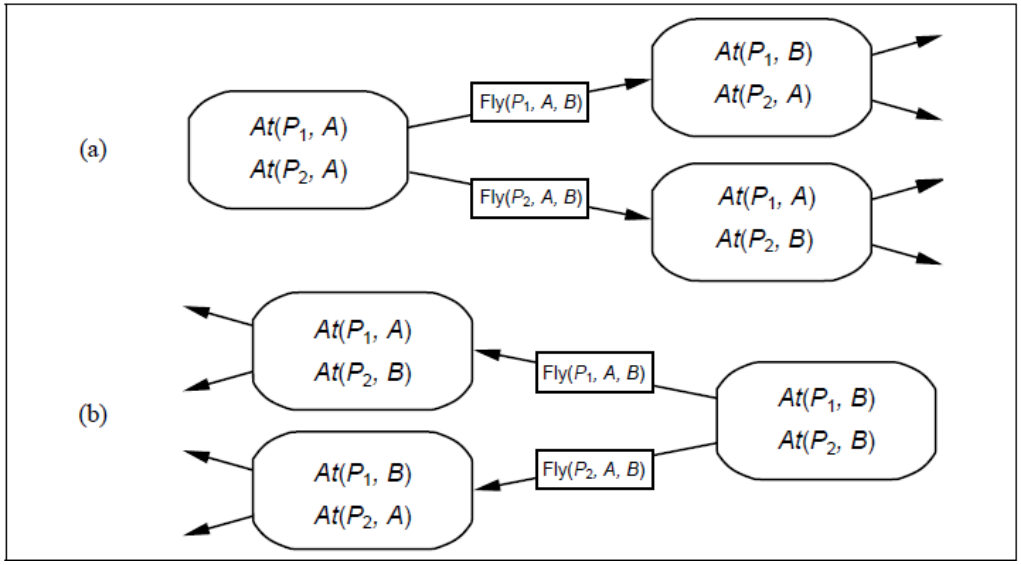
Achieve on(a,b) via stack(a,b) with preconds: [holding(a),clear(b)]
|Achieve holding(a) via pickup(a) with preconds: [ontable(a),clear(a),handempty]
||Achieve clear(a) via unstack(_1584,a) with preconds:
[on(_1584,a),clear(_1584),handempty]
||Applying unstack(c,a)
||Achieve handempty via putdown(_2691) with preconds: [holding(_2691)]
||Applying putdown(c)
|Applying pickup(a)
Applying stack(a,b)
Achieve on(b,c) via stack(b,c) with preconds: [holding(b),clear(c)]
|Achieve holding(b) via pickup(b) with preconds: [ontable(b),clear(b),handempty]
||Achieve clear(b) via unstack(_5625,b) with preconds:
[on(_5625,b),clear(_5625),handempty]
||Applying unstack(a,b)
||Achieve handempty via putdown(_6648) with preconds: [holding(_6648)]
||Applying putdown(a)
|Applying pickup(b)
Applying stack(b,c)
Achieve on(a,b) via stack(a,b) with preconds: [holding(a),clear(b)]
|Achieve holding(a) via pickup(a) with preconds: [ontable(a),clear(a),handempty]
|Applying pickup(a)
Applying stack(a,b)

From
[clear(b),clear(c),ontable(a),ontable(b),on
(c,a),handempty]
To [on(a,b),on(b,c),ontable(c)]
Do:
- unstack(c,a)
- putdown(c)
- pickup(a)
- stack(a,b)
- unstack(a,b)
- putdown(a)
- pickup(b)
- stack(b,c)
- pickup(a)
- stack(a,b)

State-space planning

- We initially have a space of situations (where you are, what you have, etc.)
- The plan is a solution found by "searching" through the situations to get to the goal
- A **progression planner** searches forward from initial state to goal state
- A **regression planner** searches backward from the goal
  - This works if operators have enough information to go both ways
  - Ideally this leads to reduced branching: the planner is only considering things that are relevant to the goal

State-space planning



- Progression planners
  - forward state-space search
  - consider the effect of all possible actions in a given state
- Regression planners
  - backward state-space search
  - Determine what must have been true in the previous state in order to achieve the current state

Progression algorithm

- Formulation as state-space search problem:
  - Initial state and goal test: obvious
  - Successor function: generate from applicable actions
  - Step cost = each action costs 1
- Any complete graph search algorithm is a complete planning algorithm.
  - E.g. A*
- Inherently inefficient:
  - (1) irrelevant actions lead to very broad search tree
  - (2) good heuristic required for efficient search

Regression algorithm

- How to determine predecessors?
  - What are the states from which applying a given action leads to the goal?

    Goal state = $At(C1, B) \wedge At(C2, B) \wedge \ldots \wedge At(C20, B)$

    Relevant action for first conjunct: $Unload(C1,p,B)$

    Works only if pre-conditions are satisfied.

    Previous state= $In(C1, p) \wedge At(p, B) \wedge At(C2, B) \wedge \ldots \wedge At(C20, B)$

    Subgoal At(C1,B) should not be present in this state.
- Actions must not undo desired literals (consistent)
- Main advantage: only relevant actions are considered.
  - Often much lower branching factor than forward search.

Regression algorithm

- General process for predecessor construction
  - Give a goal description G
  - Let A be an action that is relevant and consistent
  - The predecessors are as follows:
    - Any positive effects of A that appear in G are deleted.
    - Each precondition literal of A is added , unless it already appears.
- Any standard search algorithm can be added to perform the search.
- Termination when predecessor satisfied by initial state.
  - In FO case, satisfaction might require a substitution.

Heuristics for state-space search

- Neither progression or regression are very efficient without a good heuristic.
  - How many actions are needed to achieve the goal?
  - Exact solution is NP hard, find a good estimate
- Two approaches to find admissible heuristic:
  - The optimal solution to the relaxed problem.
    - Remove all preconditions from actions
  - The subgoal independence assumption:

    The cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving the subproblems independently.

Planning heuristics

- Just as with search, we need an **admissible** heuristic that we can apply to planning states
  - Estimate of the distance (number of actions) to the goal
- Planning typically uses **relaxation** to create heuristics
  - Ignore all or selected preconditions
  - Ignore delete lists (movement towards goal is never undone)
  - Use state abstraction (group together "similar" states and treat them as though they are identical) – e.g., ignore fluents
  - Assume subgoal independence (use max cost; or if subgoals actually are independent, can sum the costs)
  - Use pattern databases to store exact solution costs of recurring subproblems

Forward search explores irrelevant actions