Action Planning

(Where logic-based representation of knowledge makes search problems more interesting)

R&N: Chap. 10.3, Chap. 11, Sect. 11.1–4
(2nd edition of the book – a pdf of chapter 11 can be found on http://aima.cs.berkeley.edu/2nd-ed/
Situation Calculus is 10.4.2 in 3rd edition)

Portions borrowed from Jean-Claude Latombe, Stanford University; Tom Lenaerts, IRIDIA, An example borrowed from Ruti Glick,Bar-Ilan University

- The goal of action planning is to choose actions and ordering relations among these actions to achieve specified goals
- Search-based problem solving applied to 8-puzzle was one example of planning, but our description of this problem used specific data structures and functions
- Here, we will develop a non-specific, logic-based language to represent knowledge about actions, states, and goals, and we will study how search algorithms can exploit this representation

Planning with situation calculus

Logic and Planning

- In Chapters 7 and 8 we learned how to represent the wumpus world in propositional and first-order logic.
- We avoided the problem of representing the actions of the agent – this caused problems because the agent’s position changed over time (and the logical representations were essentially capturing a ‘snapshot’ of the world).

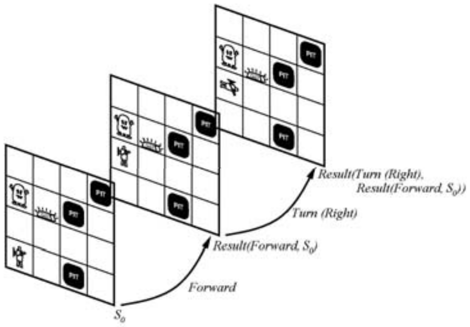
Representing Actions in Logic

(1) using temporal indices for items that might change (such as the location and orientation of the agent in the wumpus world).

(2) using situational calculus which allows us to capture how certain elements in a representation might change as a result of doing an action. These elements are indexed by the situation in which they occur.

The Ontology of Situation Calculus

- Need to be able to represent the current situation and what happens when actions are applied
- **Actions** – represented as logical terms E.g., Forward, Turn(right)
- **Situations** – logical terms consisting of the initial situation and all situations generated by applying an action to a situation. Function Result(a, s) names the situation that results when action a is done in situation s.



The Ontology of Situation Calculus

- **Fluents** – functions and predicates that vary from one situation to the next. By convention, the situation is always the last argument. E.g., ¬Holding(G1, S0); Age(Wumpus, S0)
- Atemporal or eternal predicates and functions are also allowed – they don't have a situation as an argument. E.g., Gold(g1); LeftLegOf(Wumpus)

(Sequences of) Actions in Situation Calculus

- Result([], S) = S
- Result([a|seq],S=Result(seq,Result(a,S))
- We can then describe a world as it stands, define a number of actions, and then attempt to prove there is a sequence of actions that results in some goal being achieved.
- An example using the Wumpus World…

Wumpus World

Let's look at a simplified version of the Wumpus world where we do not worry about orientation and the agent can Go to another location as long as it is adjacent to its current location.

- Suppose agent is at [1,1] and gold is at [1,2]
- Aim: have gold at [1,1]

Wumpus World with 2 Fluents: At(o,x,s) and Holding(o,s)

- Initial Knowledge: At(Agent, [1,1], S0) ^ At(G1,[1,2],S0)

Must also say that is all we know and what is not true:

- At(o,x,S0) ⟷ [(o=Agent^x=[1,1]) V
  (o=G1^x=[1,2])]
- ¬Holding(o,S0)

Need the gold and what things are Adjacent:

- Gold(G1) ^ Adjacent([1,1],[1,2]) ^
  Adjacent([1,2],[1,1])

Goal

Want to be able to prove something like:

- At(G1,[1,1],Result([Go([1,1],[1,2]),
  Grab(G1),
  Go([1,2],[1,1])],S0)

Or – more interesting - construct a plan to get the gold:

$$\exists seq(At(G1,[1,1],Result(seq,S0)))$$

- What has to go in our knowledge base to prove these things?
- Need to have a description of actions

Describing Actions

- Need 2 axioms for each action: A possibility Axiom that says when it is possible to execute, and an effect axiom that says what happens when the action is executed.

Possibility Axiom:

- Preconditions → Poss(a,s)

Effect Axiom:

- Poss(a,s) → Changes that result from taking an action

Possibility Axioms

- At(Agent,x,s) ^ Adjacent(x,y)
  → Poss(Go(x,y), s)
  (an agent can go between adjacent locations)
- Gold(g) ^ At(Agent,x,s) ^ At(g,x,s)
  → Poss(Grab(g), s)
  (an agent can grab a piece of gold in its location)
- Holding(g,s) → Poss(Release(g), s)
  (an agent can release something it is holding)

Effect Axioms

- Poss(Go(x,y), s) →
  At(Agent,y, Result(Go(x,y),s))
  (going from x to y results in being in y in the new situation)
- Poss(Grab(g), s) →
  Holding(g, Result(Grab(g),s))
  (grabbing g results in holding g in the new situation)
- Poss(Release(g), s) →
  ¬Holding(g, Result(Release(g), s))
  (releasing g results in not holding g in the new situation)

Putting the Actions Together…

- At(Agent,x,s) ^ Adjacent(x,y) →
  At(Agent,y, Result(Go(x,y),s))
- Gold(g) ^ At(Agent,x,s) ^ At(g,x,s) →
  Holding(g, Result(Grab(g),s))
- Holding(g,s) →
  ¬Holding(g, Result(Release(g), s))

Not enough to plan because we don't know what stays the same in the result situations (we have only specified what changes).

So, after Go([1,1], [1,2]) in S0 we know

- At(Agent,[1,2],Result(Go([1,1],[1,2]),S0))
- But, we don't know where the gold is in that new situation.
- This is called the frame problem…

Frame Problem

- Problem is that the effect axioms say what changes, but don't say what stays the same. Need Frame axioms that do say that (for every fluent that doesn't change).

Frame Problem

- One solution: write explicit frame axioms that say what stays the same.
- If (At(o,x,s) and o is not the agent and the agent isn't holding o), then At(o,x, Result(Go(y,z),s))

Need such an axiom for each fluent for each action (where the fluent doesn't change)

Part of a Prelims Question

- **Planning** Your ceiling light is controlled by two switches. As usual, changing either switch changes the state of the light. Assume all bulbs work. The light only works if there is a bulb in the socket, but you have no way to add a bulb. Initially the light is off and there is a bulb in the socket.
- (5 points) Formalize this situation in situational calculus. (Looks like FOPC; don't plan, just formalize.)

Have unary predicates Switch(x) and On(s) and Bulbin(s), initial state S0, switches x and situations s. Reified action predicate MoveSwitch and new-situation function Do (NOTE: the book uses Result instead of Do).

Initial State is S0 and we have: Bulbin(S0), ~On(S0)

Rules:

- (On(s) ^ Bulbin(s) ^ Switch(x)) -> ~On(Do(MoveSwitch(x,s)))
- (~On(s) ^ Bulbin(s) ^ Switch(x)) -> On(Do(MoveSwitch(x,s))) ;; two action rules
- (Bulbin(s)) -> (Bulbin(Do(Moveswitch(x,s)))) ;; frame axiom

Planning – Does it Scale?

2 types of planning so far

- Regular state space search
- Logic-based situational calculus

These suffer from being overwhelmed by irrelevant actions

Reasoning backwards (goal directed), problem decomposition (nearly decomposable), heuristic functions.

Knowledge Representation Tradeoff

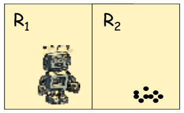
- Expressiveness vs. computational efficiency
- STRIPS: a simple, still reasonably expressive planning language based on propositional logic
  1) Examples of planning problems in STRIPS
  2) Extensions of STRIPS
  3) Planning methods

SHAKEY the robot

- Like programming, knowledge representation is still an art

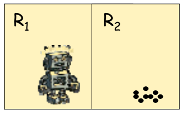
STRIPS Language through Examples

Vacuum-Robot Example



- Two rooms: R_1 and R_2
- A vacuum robot
- Dust

State Representation

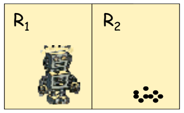


$\underbrace{\text{In(Robot, } R_1)}\ \underbrace{\wedge}\ \underbrace{\text{Clean}(R_1)}$

Propositions that "hold" (i.e. are true) in the state

Logical "and" connective

State Representation



In(Robot, R_1) ∧ Clean(R_1)

- Conjunction of propositions
- No negated proposition, such as ¬Clean(R_2)
- Closed-world assumption: Every proposition that is not listed in a state is false in that state
- No "or" connective, such as In(Robot,R_1)∨In(Robot,R_2)
- No variable, e.g., ∃x Clean(x) [literals ground and function free]

Goal Representation

Example: Clean(R_1) ∧ Clean(R_2)

- Conjunction of propositions
- No negated proposition
- No "or" connective
- No variable

A goal G is achieved in a state S if all the propositions in G (called sub-goals) are also in S

A goal is a partial representation of a state

Action Representation

Right

- Precondition = In(Robot, R_1)
- Delete-list = In(Robot, R_1)
- Add-list = In(Robot, R_2)

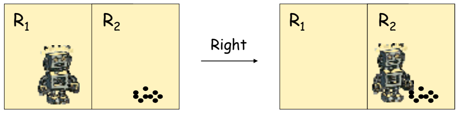
Sets of propositions

Same form as a goal: conjunction of propositions

Action Representation

Right

- Precondition = In(Robot, R_1)
- Delete-list = In(Robot, R_1)
- Add-list = In(Robot, R_2)



In(Robot, R_1) ∧ Clean(R_1)

In(Robot, R_2) ∧ Clean(R_1)

Action Representation

Right

- Precondition = In(Robot, R_1)
- Delete-list = In(Robot, R_1)
- Add-list = In(Robot, R_2)

- An action A is applicable to a state S if the propositions in its precondition are all in S (this may involve unifying variables)
- The application of A to S is a new state obtained by (1) applying the variable substitutions required to make the preconditions true, (2) deleting the propositions in the delete list from S, and (3) adding those in the add list

Other Actions

Left

- P = In(Robot, R_2)
- D = In(Robot, R_2)
- A = In(Robot, R_1)

Suck(r)

- P = In(Robot, r)
- D = ∅ [empty list]
- A = Clean(r)

Action Schema

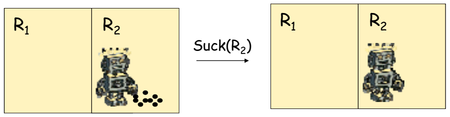
It describes several actions, here: Suck(R_1) and Suck(R_2)

Parameter that will get "instantiated" by matching the precondition against a state

Suck(r)

- P = In(Robot, r)
- D = ∅
- A = Clean(r)

Action Schema



In(Robot, R_2) ∧ Clean(R_1)

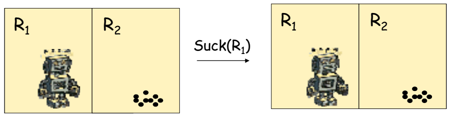
In(Robot, R_2) ∧ Clean(R_1) ∧ Clean(R_2)

r ← R_2

Suck(r)

- P = In(Robot, r)
- D = ∅
- A = Clean(r)

Action Schema



In(Robot, R_1) ∧ Clean(R_1)

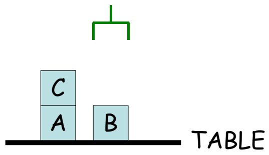
In(Robot, R_1) ∧ Clean(R_1)

r ← R_1

Suck(r)

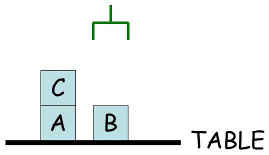
- P = In(Robot, r)
- D = ∅
- A = Clean(r)

Blocks-World Example



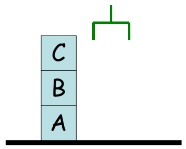
- A robot hand can move blocks on a table
- The hand cannot hold more than one block at a time
- No two blocks can fit directly on the same block
- The table is arbitrarily large

State



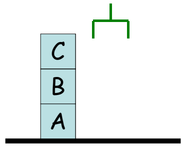
Block(A) ∧ Block(B) ∧ Block(C) ∧
On(A,Table) ∧ On(B,Table) ∧ On(C,A) ∧
Clear(B) ∧ Clear(C) ∧ Handempty

Goal



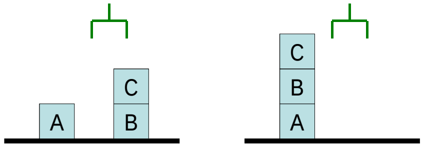
On(A,TABLE) ∧ On(B,A) ∧ On(C,B) ∧ Clear(C)

Goal



On(A,TABLE) ∧ On(B,A) ∧ On(C,B) ∧ Clear(C)

Goal



On(A,Table) ∧ On(C,B)

Action

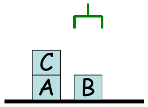
Unstack(x,y)

P = Handempty∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)
D = Handempty, Clear(x), On(x,y)
A = Holding(x), Clear(y)

Action

Unstack(x,y)

P = Handempty∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)
D = Handempty, Clear(x), On(x,y)
A = Holding(x), Clear(y)



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

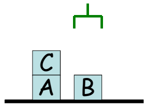
Unstack(C,A)

P = Handempty∧ Block(C) ∧ Block(A) ∧ Clear(C) ∧ On(C,A)
D = Handempty, Clear(C), On(C,A)
A = Holding(C), Clear(A)

Action

Unstack(x,y)

P = Handempty∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)
D = Handempty, Clear(x), On(x,y)
A = Holding(x), Clear(y)



Block(A) ∧ Block(B) ∧ Block(C) ∧
On(A,Table) ∧ On(B,Table) ∧ ~~On(C,A)~~
∧ Clear(B) ∧ ~~Clear(C)~~ ∧ ~~Handempty~~
∧ Holding(A) ∧ Clear(A)

Unstack(C,A)

P = Handempty∧ Block(C) ∧ Block(A) ∧ Clear(C) ∧ On(C,A)
D = Handempty, Clear(C), On(C,A)
A = Holding(C), Clear(A)

All Actions

Unstack(x,y)
P = Handempty ∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)
D = Handempty, Clear(x), On(x,y)
A = Holding(x), Clear(y)

Stack(x,y)
P = Holding(x) ∧ Block(x) ∧ Block(y) ∧ Clear(y)
D = Clear(y), Holding(x)
A = On(x,y), Clear(x), Handempty

Pickup(x)
P = Handempty ∧ Block(x) ∧ Clear(x) ∧ On(x,Table)
D = Handempty, Clear(x), ¬On(x,Table)
A = Holding(x)

Putdown(x)
P = Holding(x), ∧ Block(x)
D = Holding(x)
A = On(x,Table), Clear(x), Handempty

All Actions

Unstack(x,y)
P = Handempty ∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)
D = Handempty, Clear(x), On(x,y)
A = Holding(x), Clear(y)

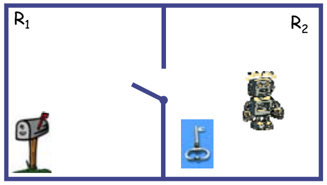
Stack(x,y)
P = Holding(x) ∧ Block(x) ∧ Block(y) ∧ Clear(y)
D = Clear(y), Holding(x),
A = On(x,y), Clear(x), Handempty

Pickup(x)
P = Handempty ∧ Block(x) ∧ Clear(x) ∧ On(x,Table)
D = Handempty, Clear(x), On(x,Table)
A = Holding(x) ⇦

Putdown(x)
P = Holding(x), ∧ Block(x)
D = Holding(x) ⇦
A = On(x,Table), Clear(x), Handempty

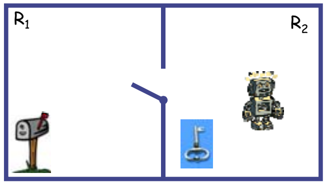
A block can always fit on the table

Key-in-Box Example



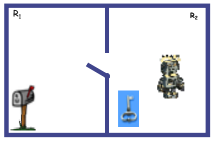
- The robot must lock the door and put the key in the box
- But, once the door is locked, the robot can't unlock it
- Once the key is in the box, the robot can't get it back

Initial State



In(Robot,R_2) ∧ In(Key,R_2) ∧ Unlocked(Door)

Actions



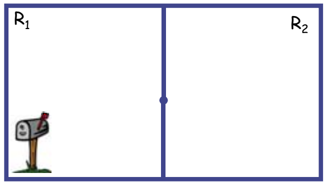
Grasp-Key-in-R_2
P = In(Robot,R_2) ∧ In(Key,R_2)
D = ∅
A = Holding(Key)

Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

Move-Key-from-R_2-into-R_1
P = In(Robot,R_2) ∧ Holding(Key) ∧ Unlocked(Door)
D = In(Robot,R_2), In(Key,R_2)
A = In(Robot,R_1), In(Key,R_1)

Put-Key-Into-Box
P = In(Robot,R_1) ∧ Holding(Key)
D = Holding(Key), In(Key,R_1)
A = In(Key,Box)

Goal

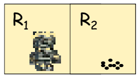


Locked(Door) ∧ In(Key,Box)

[The robot's location isn't specified in the goal]

Some Extensions of STRIPS Language

Extensions of STRIPS
1. Negated propositions in a state



In(Robot, R_1) ∧ ¬In(Robot, R_2) ∧ Clean(R_1) ∧ ¬Clean(R_2)

Dump-Dirt(r)
P = In(Robot, r) ∧ Clean(r)
E = ¬Clean(r)

Suck(r)
P = In(Robot, r) ∧ ¬Clean(r)
E = Clean(r)

- Q in E means delete ¬Q and add Q to the state
- ¬Q in E means delete Q and add ¬Q

Open world assumption: A proposition in a state is true if it appears positively and false otherwise. A non-present proposition is unknown

Planning methods can be extended rather easily to handle negated proposition (see R&N), but state descriptions are often much longer (e.g., imagine if there were 10 rooms in the above example)

Extensions of STRIPS
2. Equality/Inequality Predicates

Blocks world:

Move(x,y,z)
P = Block(x) ∧ Block(y) ∧ Block(z) ∧ On(x,y) ∧ Clear(x)
∧ Clear(z) ∧ (x≠z)
D = On(x,y), Clear(z)
A = On(x,z), Clear(y)

Move(x,Table,z)
P = Block(x) ∧ Block(z) ∧ On(x,Table) ∧ Clear(x)
∧ Clear(z) ∧ (x≠z)
D = On(x,y), Clear(z)
A = On(x,z)

Move(x, y, Table)
P = Block(x) ∧ Block(y) ∧ On(x,y) ∧ Clear(x)
D = On(x,y)
A = On(x,Table), Clear(y)

Extensions of STRIPS
2. Equality/Inequality Predicates

Blocks world:

Move(x,y,z)
P = Block(x) ∧ Block(y) ∧ Block(z) ∧ On(x,y) ∧ Clear(x)
∧ Clear(z) ∧ (x≠z)
D = On(x,y), Clear(z)
A = On(x,z), Clear(y)

Move(x,Table,z)
P = Block(x) ∧ Block(z) ∧
∧ Clear(z) ∧ (x≠z)
D = On(x,y), Clear(z)
A = On(x,z)

Move(x, y, Table)
P = Block(x) ∧ Block(y) ∧
D = On(x,y)
A = On(x,Table), Clear(y)

Planning methods simply evaluate (x≠z) when the two variables are instantiated

This is equivalent to considering that propositions (A ≠ B) , (A ≠ C) , ... are implicitly in every state

Extensions of STRIPS (not covered)
3. Algebraic expressions

Two flasks F_1 and F_2 have volume capacities of 30 and 50, respectively

F_1 contains volume 20 of some liquid

F_2 contains volume 15 of this liquid

State:

Cap(F_1,30) ∧ Cont (F_1,20) ∧ Cap(F_2, 50) ∧ Cont (F_2,15)

Action of pouring a flask into the other:

Pour(f,f')

P = Cont(f,x) ∧ Cap(f,'c') ∧ Cont(f',y)
D = Cont(f,x), Cont(f',y),
A = Cont(f,max{x+y-c',0}), Cont(f',min{x+y,c'})

Extensions of STRIPS (not covered)
3. Algebraic expressions

Two flasks F_1 and F_2 have volume capacities of 30 and 50, respectively

F_1 contains volume 20 of some liquid

F_2 co

Stat

C ,15)

This extension requires some planning methods to be equipped with algebraic manipulation capabilities

Action of pouring a flask into the other:

Pour(f,f')

P = Cont(f,x) ∧ Cap(f,'c') ∧ Cont(f',y)
D = Cont(f,x), Cont(f',y),
A = Cont(f,max{x+y-c',0}), Cont(f',min{x+y,c'})

Extensions of STRIPS (not covered)
4. State Constraints

h	b	
c	d	g
e	a	f

State:

Adj(1,2) ∧ Adj(2,1) ∧ ... ∧ Adj(8,9) ∧ Adj(9,8) ∧
At(h,1) ∧ At(b,2) ∧ At(c,4) ∧ ... ∧ At(f,9) ∧ Empty(3)

Move(x,y)

P = At(x,y) ∧ Empty(z) ∧ Adj(y,z)
D = At(x,y), Empty(z)
A = At(x,z), Empty(y)

Extensions of STRIPS (not covered)
4. State Constraints

h	b	
c	d	g
e	a	f

State:

Adj(1,2) ~~∧ Adj(2,1)~~ ∧ ... ∧ Adj(8,9) ~~∧ Adj(9,8)~~ ∧
At(h,1) ∧ At(b,2) ∧ At(c,4) ∧ ... ∧ At(f,9) ∧ Empty(3)

State constraint:

Adj(x,y) → Adj(y,x)

Move(x,y)

P = At(x,y) ∧ Empty(z) ∧ Adj(y,z)
D = At(x,y), Empty(z)
A = At(x,z), Empty(y)

More Complex State Constraints (not covered) in 1st-Order Predicate Logic

Blocks world:

(∀x)[Block(x) ∧ ¬(∃y)On(y,x) ∧ ¬Holding(x)] → Clear(x)

(∀x)[Block(x) ∧ Clear(x)] → ¬(∃y)On(y,x) ∧ ¬Holding(x)

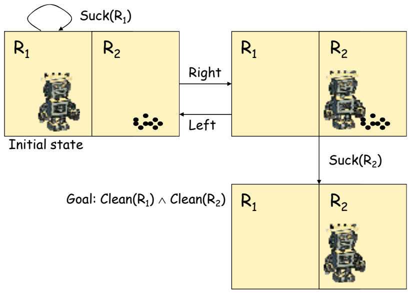
Handempty ↔ ¬(∃x)Holding(x)

would simplify greatly the description of the actions

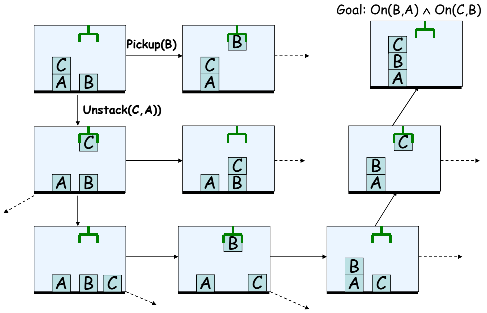
State constraints require equipping planning methods with logical deduction capabilities to determine whether goals are achieved or preconditions are satisfied

Planning Methods

Forward Planning



Forward Planning



Need for an Accurate Heuristic

- Forward planning simply searches the space of world states from the initial to the goal state
- Imagine an agent with a large library of actions, whose goal is G, e.g., G = Have(Milk)
- In general, many actions are applicable to any given state, so the branching factor is huge
- In any given state, most applicable actions are irrelevant to reaching the goal Have(Milk)
- Fortunately, an accurate consistent heuristic can be computed using planning graphs (we'll come back to that!)

- Forward planning still suffers from an excessive branching factor
- In general, there are many fewer actions that are relevant to achieving a goal than actions that are applicable to a state
- How to determine which actions are relevant? How to use them?
- → Backward planning

Goal-Relevant Action

- An action is relevant to achieving a goal if a proposition in its add list matches a sub-goal proposition
- For example:

Stack(B,A)
P = Holding(B) ∧ Block(B) ∧ Block(A) ∧ Clear(A)
D = Clear(A), Holding(B),
A = On(B,A), Clear(B), Handempty

is relevant to achieving On(B,A)∧On(C,B)

Regression of a Goal

The regression of a goal G through an action A is the least constraining precondition R[G,A] such that:

If a state S achieves R[G,A] then:
1. The precondition of A is achieved in S
2. Applying A to S yields a state that achieves G

Example

- G = On(B,A) ∧ On(C,B)
- **Stack(C,B)**

  P = Holding(C) ∧ Block(C) ∧ Block(B) ∧ Clear(B)

  D = Clear(B), Holding(C)

  A = On(C,B), Clear(C), Handempty
- R[G,Stack(C,B)] =

  On(B,A) ∧

  Holding(C) ∧ Block(C) ∧ Block(B) ∧ Clear(B)

Example

- G = On(B,A) ∧ On(C,B)
- **Stack(C,B)**

  P = Holding(C) ∧ Block(C) ∧ Block(B) ∧ Clear(B)

  D = Clear(B), Holding(C)

  A = On(C,B), Clear(C), Handempty
- R[G,Stack(C,B)] =

  On(B,A) ∧

  Holding(C) ∧ Block(C) ∧ Block(B) ∧ Clear(B)

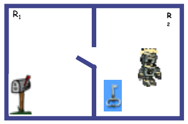
Another Example

- G = In(key,Box) ∧ Holding(Key)
- **Put-Key-Into-Box**

  P = In(Robot,R_1) ∧ Holding(Key)

  D = Holding(Key), In(Key,R_1)

  A = In(Key,Box)



- R[G,Put-Key-Into-Box] = False

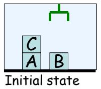
  where False is the un-achievable goal
- This means that In(key,Box) ∧ Holding(Key) can't be achieved by executing **Put-Key-Into-Box**

Computation of R[G,A]

1. If any sub-goal of G is in A's delete list then return False
2. Else
   a. G' ← Precondition of A
   b. For every sub-goal SG of G do
   c. If SG is not in A's add list then add SG to G'
3. Return G'

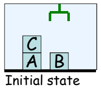
Backward Planning

On(B,A) ∧ On(C,B)



Initial state

Backward Planning



Initial state

On(B,A) ∧ On(C,B)

Stack(C,B)

On(B,A) ∧ Holding(C) ∧ Clear(B)

Pickup(C)

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

Stack(B,A)

Clear(C) ∧ On(C,TABLE) ∧ Holding(B) ∧ Clear(A)

Pickup(B)

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

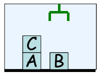
Putdown(C)

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

Unstack(C,A)

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

Backward Planning



Initial state

On(B,A) ∧ On(C,B)

↓ Stack(C,B)

On(B,A) ∧ Holding(C) ∧ Clear(B)

↓ Pickup(C)

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

↓ Stack(B,A)

Clear(C) ∧ On(C,TABLE) ∧ Holding(B) ∧ Clear(A)

↓ Pickup(B)

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

↓ Putdown(C)

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

↓ Unstack(C,A)

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

Search Tree

- Backward planning searches a space of goals from the original goal of the problem to a goal that is satisfied in the initial state
- There are often many fewer actions relevant to a goal than there are actions applicable to a state → smaller branching factor than in forward planning
- The lengths of the solution paths are the same

How Does Backward Planning Detect Dead-Ends? (not covered)

On(B,A) ∧ On(C,B)

↓ Stack(B,A)

Holding(B) ∧ Clear(A) ∧ On(C,B)

↓ Stack(C,B)

Holding(B) ∧ Clear(A) ∧ Holding(C) ∧ Clear (B)

↓ Pick(B) [delete list contains Clear(B)]

False

How Does Backward Planning Detect Dead-Ends? (not covered)

On(B,A) ∧ On(C,B)

↓ Stack(B,A)

Holding(B) ∧ Clear(A) ∧ On(C,B)

A state constraint such as Holding(x) → ¬(∃y)On(y,x) would have made it possible to prune the path earlier

Drawbacks of Forward and Backward Planning

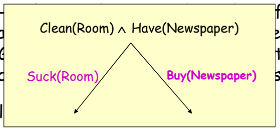
- Along any path of the search tree, they commit to a total ordering on selected actions (linear planning)
- They do not take advantage of possible (almost) independence among sub-goals, nor do they deal well with interferences among sub-goals

Independent Sub-Goals

- Example:
  Clean(Room) ∧ Have(Newspaper)
- Two sub-goals G_1 and G_2 are independent if two plans P_1 and P_2 can be computed independently of each other to achieve G_1 and G_2, respectively, and executing the two plans in any order, e.g., P_1 then P_2, achieves $G_1 \wedge G_2$
- Sub-goals are often (almost) independent

Independent Sub-Goals

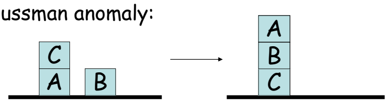
- Example:
  Clean(Room) ∧ Have(Newspaper)
- Two sub- f two plans P_1 and P_2 ca each other to achieve G ting the two plans in s $G_1 \wedge G_2$



- Sub-goal
- By not breaking a goal into sub-goals, forward and backward planning methods may increase the size of the search tree. They may also produce plans that oddly oscillate between goals

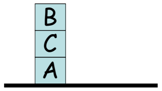
Interference Among Sub-Goals

Sussman anomaly:



On(B,C) ∧ On(A,B)

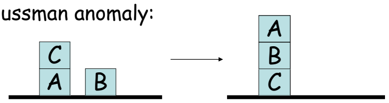
If we achieve On(B,C) first, we reach:



Then, to achieve On(A,B) we need to undo On(B,C)

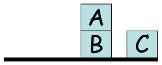
Interference Among Sub-Goals

Sussman anomaly:



On(B,C) ∧ On(A,B)

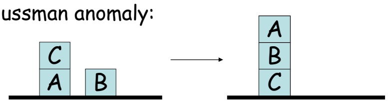
Instead, if we achieve On(A,B) first, we reach:



Then, to achieve On(B,C) we new to undo On(A,B)

Interference Among Sub-Goals

Sussman anomaly:

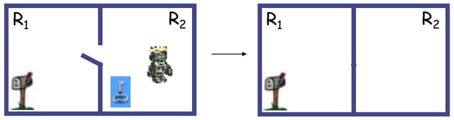


On(B,C) ∧ On(A,B)

To solve this problem, one must interweave actions aimed at one sub-goal and actions aimed at the other sub-goal

Interference Among Sub-Goals

Key-in-box example:



Locked(Door) ∧ In(Key,Box)

Here, achieving a sub-goal before the other leads to the loss of a "resource" – the key or the door – that prevents the robot from achieving the other sub-goal

Nonlinear (Partial-Order) Planning

- Idea: Avoid any ordering on actions until interferences have been detected
- Form of "least" commitment reasoning

Search Tree

- Nonlinear planning searches a space of plans

Search method	Search space
Forward planning	States
Backward planning	Goals
Nonlinear planning	Plans

Partial-order planning

- Progression and regression planning are *totally ordered plan search* forms.
  - They cannot take advantage of problem decomposition.
    - Decisions must be made on how to sequence actions on all the subproblems
- Least commitment strategy:
  - Delay choice during search

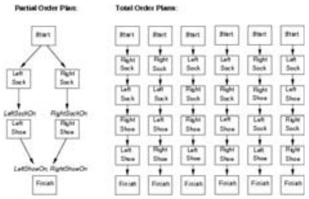
Shoe example

Goal(RightShoeOn ∧ LeftShoeOn)
Init()
Action(RightShoe, PRECOND: RightSockOn
EFFECT: RightShoeOn)
Action(RightSock, PRECOND:
EFFECT: RightSockOn)
Action(LeftShoe, PRECOND: LeftSockOn
EFFECT: LeftShoeOn)
Action(LeftSock, PRECOND:
EFFECT: LeftSockOn)

Planner: combine two action sequences (1)leftsock, leftshoe (2)rightsock, rightshoe that can be independently derived.

Partial-order planning

- Any planning algorithm that can place two actions into a plan without which comes first is a POL.



POL as a search problem

- States (or our search) are (mostly unfinished) plans.
  - Initial state: the empty plan contains only start and finish actions.
  - Actions refine the plan (adding to it) until we come up with a complete plan that solves the problem.
  - Actions on plans: add a step, impose an ordering, instantiate a variable, etc…

POL as a search problem through plans

- Each plan has 4 components:
  - A set of actions (steps of the plan)
  - A set of ordering constraints: A < B
    - Cycles represent contradictions.
  - A set of causal links $A \xrightarrow{p} B$
    - Read: A achieves p for B
    - The plan may not be extended by adding a new action C that **conflicts** with the causal link. (if the effect of C is ¬p and if C could come after A and before B)
  - A set of open preconditions.
    - If precondition is not achieved by action in the plan.
    - Planners will work to reduce the set of open precondtions to the empty set, without introducing a contradition

POL as a search problem

- A plan is *consistent* iff there are no cycles in the ordering constraints and no conflicts with the causal links.
- A consistent plan with no open preconditions is a *solution*.
- A partial order plan is executed by repeatedly choosing *any* of the possible next actions.
  - This flexibility is a benefit in non-cooperative environments.

Solving POL

- Assume propositional planning problems:
  - The initial plan contains *Start* and *Finish*, the ordering constraint *Start* < *Finish*, no causal links, all the preconditions in *Finish* are open.
  - Successor function :
    - picks one open precondition *p* on an action *B* and
    - generates a successor plan for every possible consistent way of choosing action *A* that achieves *p*.
  - Test goal

Enforcing consistency

- When generating successor plan:
  - The causal link A--p->B and the ordering constraint A < B are added to the plan.
    - If A is new also add start < A and A < finish to the plan
  - Resolve conflicts between new causal link and all existing actions (i.e., if C "undoes" p then order by adding either B<C or C<A)
  - Resolve conflicts between action A (if new) and all existing causal links.

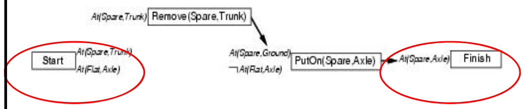
Process summary

- Operators on partial plans
  - Add link from existing plan to open precondition.
  - Add a step to fulfill an open condition.
  - Order one step w.r.t another to remove possible conflicts
- Gradually move from incomplete/vague plans to complete/correct plans
- Backtrack if an open condition is unachievable or if a conflict is unresolvable.

Example: Spare tire problem

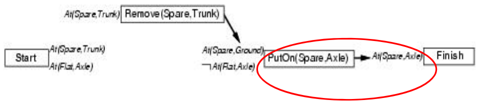
Init(At(Flat, Axle) ∧ At(Spare,trunk))
Goal(At(Spare,Axle))
Action(Remove(Spare,Trunk)
PRECOND: *At(Spare,Trunk)*
EFFECT: *¬At(Spare,Trunk) ∧ At(Spare,Ground))*
Action(Remove(Flat,Axle)
PRECOND: *At(Flat,Axle)*
EFFECT: *¬At(Flat,Axle) ∧ At(Flat,Ground))*
Action(PutOn(Spare,Axle)
PRECOND: *At(Spare,Groundp) ∧¬At(Flat,Axle)*
EFFECT: *At(Spare,Axle) ∧ ¬Ar(Spare,Ground))*
Action(LeaveOvernight
PRECOND:
EFFECT: *¬ At(Spare,Ground) ∧ ¬ At(Spare,Axle) ∧ ¬ At(Spare,trunk) ∧*
¬ At(Flat,Ground) ∧ ¬ At(Flat,Axle))

Solving the problem



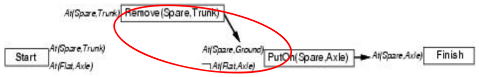
- Intial plan: Start with EFFECTS and Finish with PRECOND.

Solving the problem



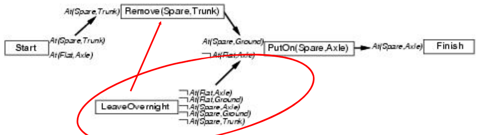
- Intial plan: Start with EFFECTS and Finish with PRECOND.
- Pick an open precondition: *At(Spare, Axle)*
- Only *PutOn(Spare, Axle)* is applicable
- Add causal link: $PutOn(Spare,Axle) \xrightarrow{At(Spare,Axle)} Finish$
- Add constraint : *PutOn(Spare, Axle) < Finish*

Solving the problem



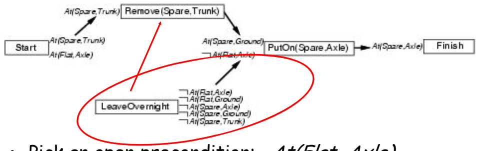
- Pick an open precondition: *At(Spare, Ground)*
- Only *Remove(Spare, Trunk)* is applicable
- Add causal link:
  $Remove(Spare,Trunk) \xrightarrow{At(Spare,Ground)} PutOn(Spare,Axle)$
- Add constraint : *Remove(Spare, Trunk) < PutOn(Spare,Axle)*

Solving the problem



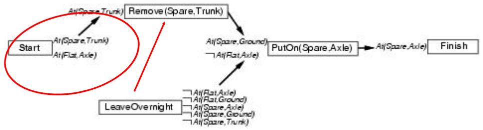
- Pick an open precondition: *¬At(Flat, Axle)*
- *LeaveOverNight* is applicable
- Conflict: $Remove(Spare,Trunk) \xrightarrow{At(Spare,Ground)} PutOn(Spare,Axle)$
- To resolve, add constraint : *LeaveOverNight < Remove(Spare, Trunk)*

Solving the problem



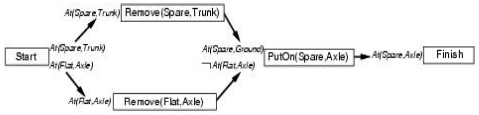
- Pick an open precondition: *¬At(Flat, Axle)*
- *LeaveOverNight* is applicable
- conflict: $Remove(Spare,Trunk) \xrightarrow{At(Spare,Ground)} PutOn(Spare,Axle)$
- To resolve, add constraint : *LeaveOverNight < Remove(Spare, Trunk)*
- Add causal link: $LeaveOverNight \xrightarrow{\neg At(Spare,Ground)} PutOn(Spare,Axle)$

Solving the problem

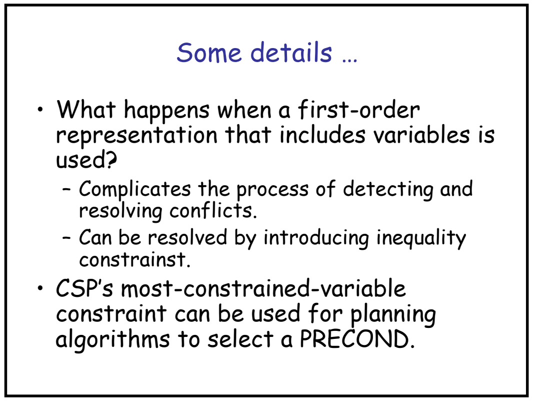


- Pick an open precondition: *At(Spare, Trunk)*
- Only *Start* is applicable
- Add causal link: $Start \xrightarrow{At(Spare,Trunk)} Remove(Spare,Trunk)$
- Conflict: of causal link with effect *At(Spare,Trunk)* in *LeaveOverNight*
  - *No re-ordering solution possible.*
- backtrack

Solving the problem

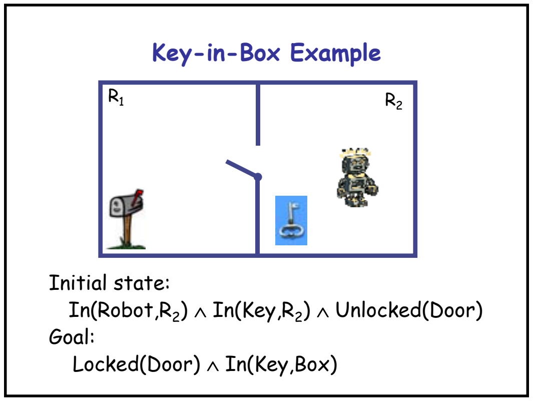


- Remove *LeaveOverNight*, *Remove(Spare, Trunk)* and causal links
- Repeat step with Remove(Spare,Trunk)
- Add also RemoveFlatAxle and finish



Some details …

- What happens when a first-order representation that includes variables is used?
  - Complicates the process of detecting and resolving conflicts.
  - Can be resolved by introducing inequality constrainst.
- CSP’s most-constrained-variable constraint can be used for planning algorithms to select a PRECOND.



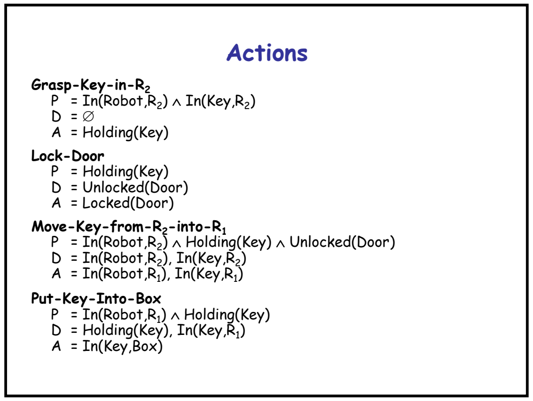
Key-in-Box Example

Initial state:

$In(Robot,R_2) \wedge In(Key,R_2) \wedge Unlocked(Door)$

Goal:

$Locked(Door) \wedge In(Key,Box)$



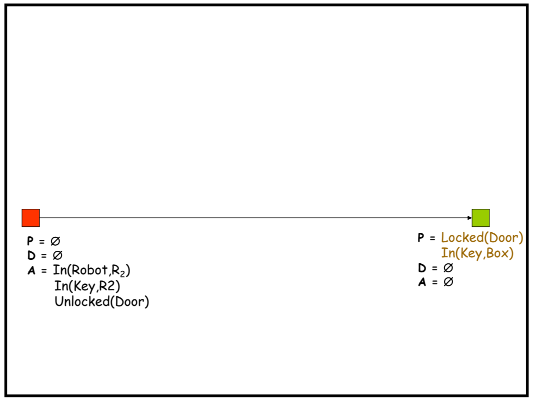
Actions

Grasp-Key-in-R_2
P = $In(Robot,R_2) \wedge In(Key,R_2)$
D = ∅
A = Holding(Key)

Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

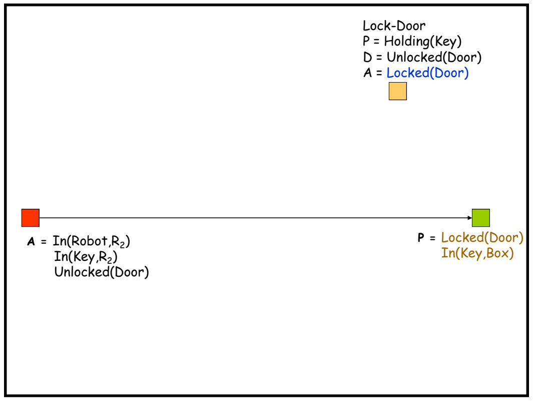
Move-Key-from-R_2-into-R_1
P = $In(Robot,R_2) \wedge Holding(Key) \wedge Unlocked(Door)$
D = $In(Robot,R_2), In(Key,R_2)$
A = $In(Robot,R_1), In(Key,R_1)$

Put-Key-Into-Box
P = $In(Robot,R_1) \wedge Holding(Key)$
D = $Holding(Key), In(Key,R_1)$
A = In(Key,Box)



P = ∅
D = ∅
A = $In(Robot,R_2)$
In(Key,R2)
Unlocked(Door)

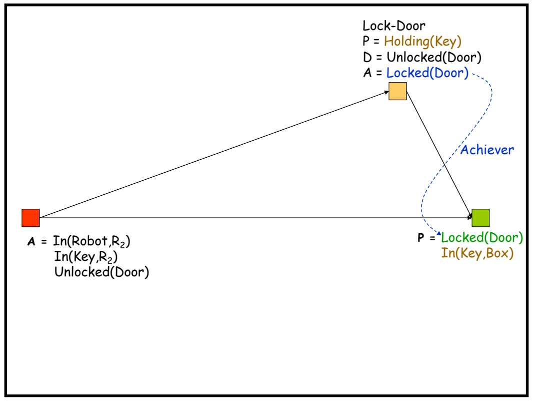
P = Locked(Door)
In(Key,Box)
D = ∅
A = ∅



Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

A = $In(Robot,R_2)$
$In(Key,R_2)$
Unlocked(Door)

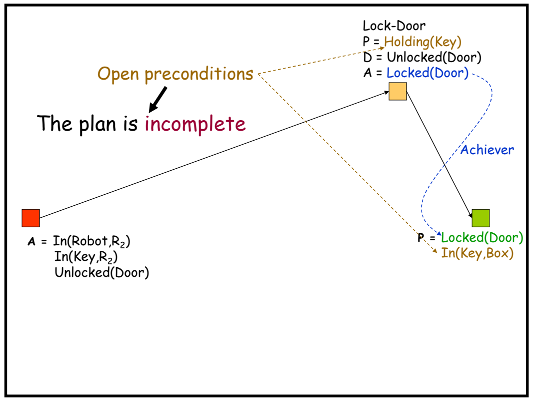
P = Locked(Door)
In(Key,Box)



Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

A = $In(Robot,R_2)$
$In(Key,R_2)$
Unlocked(Door)

P = Locked(Door)
In(Key,Box)



Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

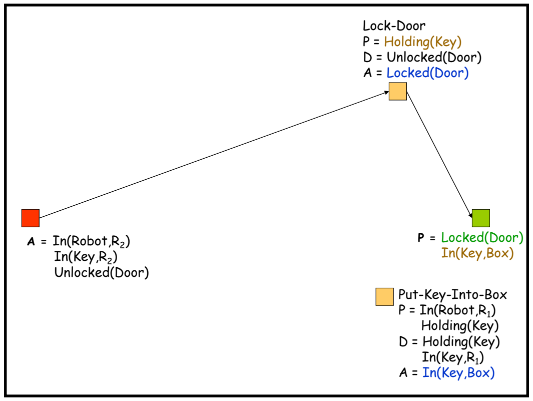
Open preconditions

The plan is incomplete

Achiever

A = In(Robot,R_2)
In(Key,R_2)
Unlocked(Door)

P = Locked(Door)
In(Key,Box)

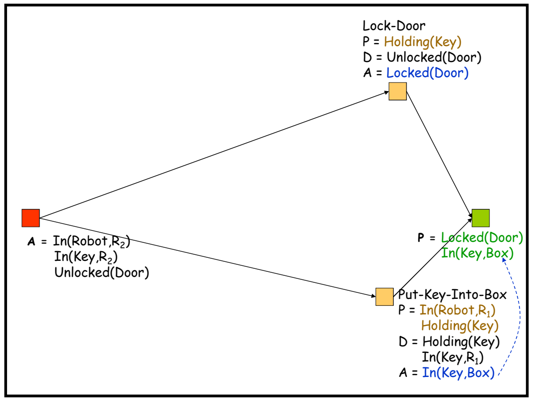


Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

A = In(Robot,R_2)
In(Key,R_2)
Unlocked(Door)

P = Locked(Door)
In(Key,Box)

Put-Key-Into-Box
P = In(Robot,R_1)
Holding(Key)
D = Holding(Key)
In(Key,R_1)
A = In(Key,Box)

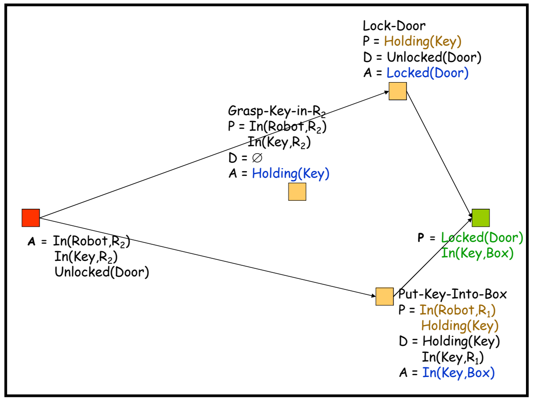


Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

A = In(Robot,R_2)
In(Key,R_2)
Unlocked(Door)

P = Locked(Door)
In(Key,Box)

Put-Key-Into-Box
P = In(Robot,R_1)
Holding(Key)
D = Holding(Key)
In(Key,R_1)
A = In(Key,Box)



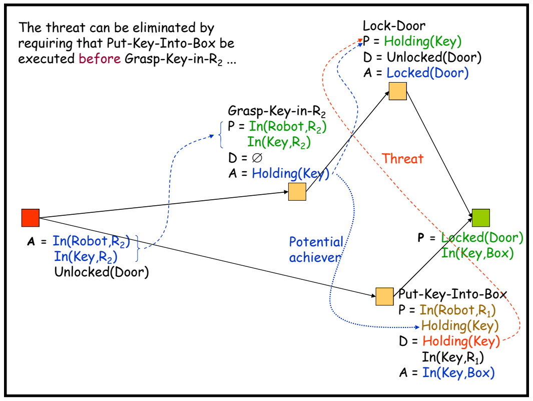
Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

Grasp-Key-in-R_2
P = In(Robot,R_2)
In(Key,R_2)
D = ∅
A = Holding(Key)

A = In(Robot,R_2)
In(Key,R_2)
Unlocked(Door)

P = Locked(Door)
In(Key,Box)

Put-Key-Into-Box
P = In(Robot,R_1)
Holding(Key)
D = Holding(Key)
In(Key,R_1)
A = In(Key,Box)



The threat can be eliminated by requiring that Put-Key-Into-Box be executed before Grasp-Key-in-R_2 ...

Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

Grasp-Key-in-R_2
P = In(Robot,R_2)
In(Key,R_2)
D = ∅
A = Holding(Key)

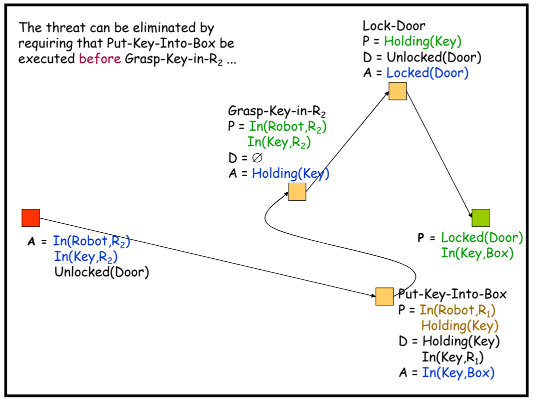
Threat

Potential achiever

A = In(Robot,R_2)
In(Key,R_2)
Unlocked(Door)

P = Locked(Door)
In(Key,Box)

Put-Key-Into-Box
P = In(Robot,R_1)
Holding(Key)
D = Holding(Key)
In(Key,R_1)
A = In(Key,Box)



The threat can be eliminated by requiring that Put-Key-Into-Box be executed before Grasp-Key-in-R_2 ...

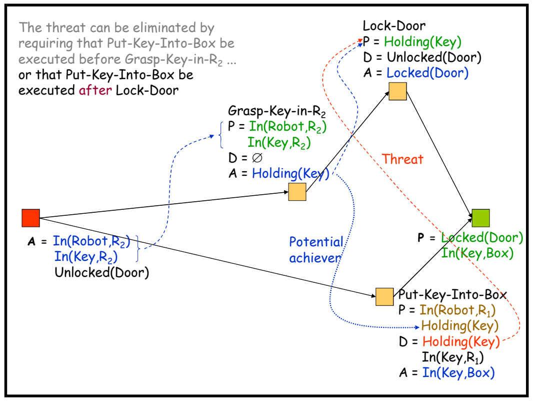
Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

Grasp-Key-in-R_2
P = In(Robot,R_2)
In(Key,R_2)
D = ∅
A = Holding(Key)

A = In(Robot,R_2)
In(Key,R_2)
Unlocked(Door)

P = Locked(Door)
In(Key,Box)

Put-Key-Into-Box
P = In(Robot,R_1)
Holding(Key)
D = Holding(Key)
In(Key,R_1)
A = In(Key,Box)



The threat can be eliminated by requiring that Put-Key-Into-Box be executed before Grasp-Key-in-R_2 ...
or that Put-Key-Into-Box be executed after Lock-Door

Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

Grasp-Key-in-R_2
P = In(Robot,R_2)
In(Key,R_2)
D = ∅
A = Holding(Key)

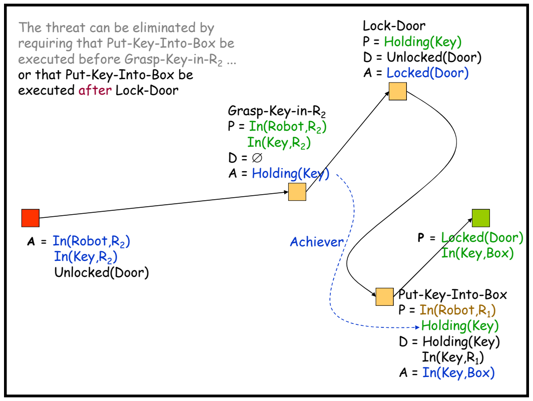
Threat

A = In(Robot,R_2)
In(Key,R_2)
Unlocked(Door)

Potential achiever

P = Locked(Door)
In(Key,Box)

Put-Key-Into-Box
P = In(Robot,R_1)
Holding(Key)
D = Holding(Key)
In(Key,R_1)
A = In(Key,Box)



The threat can be eliminated by requiring that Put-Key-Into-Box be executed before Grasp-Key-in-R_2 ...
or that Put-Key-Into-Box be executed after Lock-Door

Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

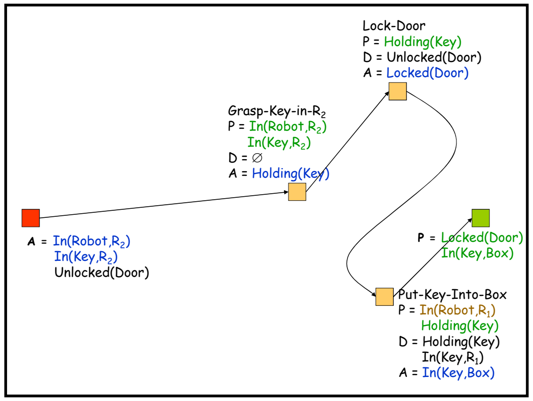
Grasp-Key-in-R_2
P = In(Robot,R_2)
In(Key,R_2)
D = ∅
A = Holding(Key)

A = In(Robot,R_2)
In(Key,R_2)
Unlocked(Door)

Achiever

P = Locked(Door)
In(Key,Box)

Put-Key-Into-Box
P = In(Robot,R_1)
Holding(Key)
D = Holding(Key)
In(Key,R_1)
A = In(Key,Box)



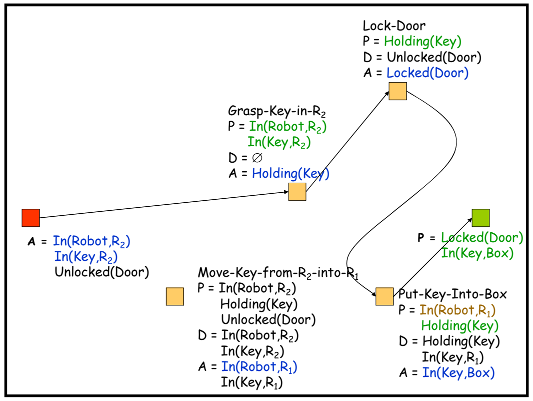
Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

Grasp-Key-in-R_2
P = In(Robot,R_2)
In(Key,R_2)
D = ∅
A = Holding(Key)

A = In(Robot,R_2)
In(Key,R_2)
Unlocked(Door)

P = Locked(Door)
In(Key,Box)

Put-Key-Into-Box
P = In(Robot,R_1)
Holding(Key)
D = Holding(Key)
In(Key,R_1)
A = In(Key,Box)



Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

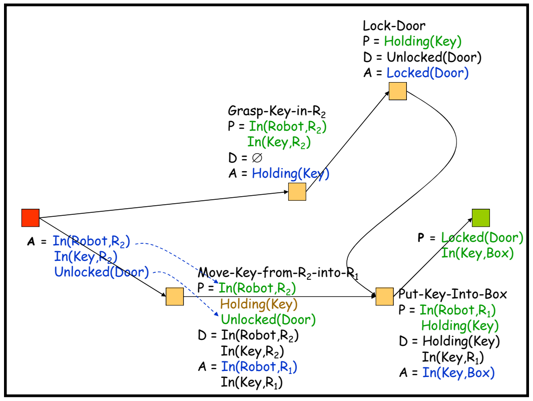
Grasp-Key-in-R_2
P = In(Robot,R_2)
In(Key,R_2)
D = ∅
A = Holding(Key)

A = In(Robot,R_2)
In(Key,R_2)
Unlocked(Door)

Move-Key-from-R_2-into-R_1
P = In(Robot,R_2)
Holding(Key)
Unlocked(Door)
D = In(Robot,R_2)
In(Key,R_2)
A = In(Robot,R_1)
In(Key,R_1)

P = Locked(Door)
In(Key,Box)

Put-Key-Into-Box
P = In(Robot,R_1)
Holding(Key)
D = Holding(Key)
In(Key,R_1)
A = In(Key,Box)



Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

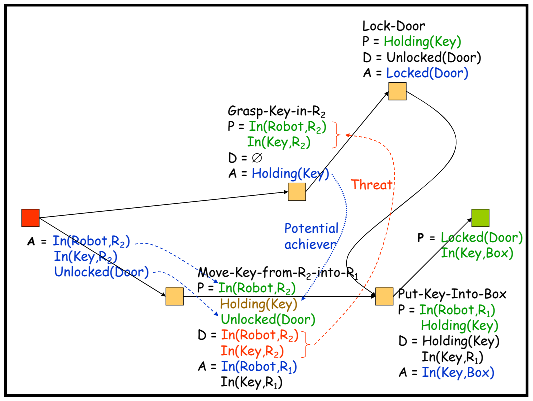
Grasp-Key-in-R_2
P = In(Robot,R_2)
In(Key,R_2)
D = ∅
A = Holding(Key)

A = In(Robot,R_2)
In(Key,R_2)
Unlocked(Door)

Move-Key-from-R_2-into-R_1
P = In(Robot,R_2)
Holding(Key)
Unlocked(Door)
D = In(Robot,R_2)
In(Key,R_2)
A = In(Robot,R_1)
In(Key,R_1)

P = Locked(Door)
In(Key,Box)

Put-Key-Into-Box
P = In(Robot,R_1)
Holding(Key)
D = Holding(Key)
In(Key,R_1)
A = In(Key,Box)



Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

Grasp-Key-in-R_2
P = In(Robot,R_2)
In(Key,R_2)
D = ∅
A = Holding(Key)

Threat

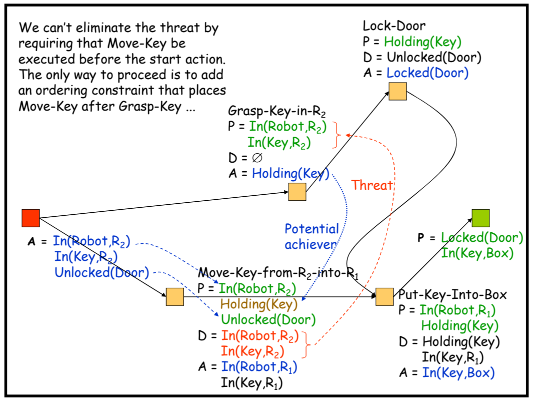
A = In(Robot,R_2)
In(Key,R_2)
Unlocked(Door)

Potential achiever

Move-Key-from-R_2-into-R_1
P = In(Robot,R_2)
Holding(Key)
Unlocked(Door)
D = In(Robot,R_2)
In(Key,R_2)
A = In(Robot,R_1)
In(Key,R_1)

P = Locked(Door)
In(Key,Box)

Put-Key-Into-Box
P = In(Robot,R_1)
Holding(Key)
D = Holding(Key)
In(Key,R_1)
A = In(Key,Box)



We can't eliminate the threat by requiring that Move-Key be executed before the start action. The only way to proceed is to add an ordering constraint that places Move-Key after Grasp-Key ...

Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

Grasp-Key-in-R_2
P = In(Robot,R_2)
In(Key,R_2)
D = ∅
A = Holding(Key)

Threat

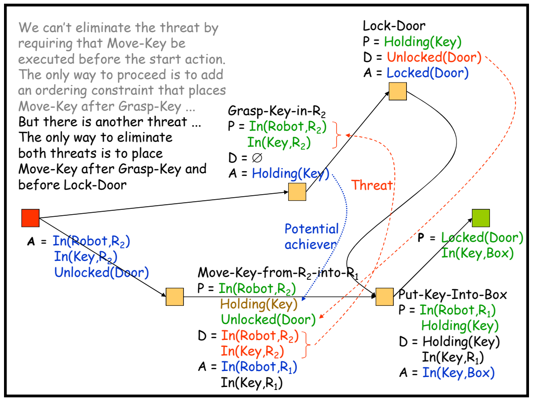
Potential achiever

A = In(Robot,R_2)
In(Key,R_2)
Unlocked(Door)

P = Locked(Door)
In(Key,Box)

Move-Key-from-R_2-into-R_1
P = In(Robot,R_2)
Holding(Key)
Unlocked(Door)
D = In(Robot,R_2)
In(Key,R_2)
A = In(Robot,R_1)
In(Key,R_1)

Put-Key-Into-Box
P = In(Robot,R_1)
Holding(Key)
D = Holding(Key)
In(Key,R_1)
A = In(Key,Box)



We can't eliminate the threat by requiring that Move-Key be executed before the start action. The only way to proceed is to add an ordering constraint that places Move-Key after Grasp-Key ...
But there is another threat ...

Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

Grasp-Key-in-R_2
P = In(Robot,R_2)
In(Key,R_2)
D = ∅
A = Holding(Key)

Threat

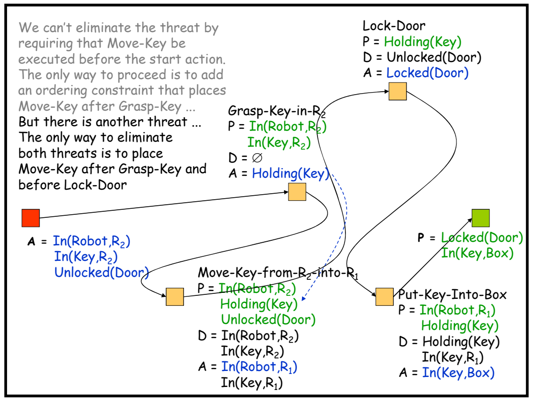
Potential achiever

A = In(Robot,R_2)
In(Key,R_2)
Unlocked(Door)

P = Locked(Door)
In(Key,Box)

Move-Key-from-R_2-into-R_1
P = In(Robot,R_2)
Holding(Key)
Unlocked(Door)
D = In(Robot,R_2)
In(Key,R_2)
A = In(Robot,R_1)
In(Key,R_1)

Put-Key-Into-Box
P = In(Robot,R_1)
Holding(Key)
D = Holding(Key)
In(Key,R_1)
A = In(Key,Box)



We can't eliminate the threat by requiring that Move-Key be executed before the start action. The only way to proceed is to add an ordering constraint that places Move-Key after Grasp-Key ...
But there is another threat ...
The only way to eliminate both threats is to place Move-Key after Grasp-Key and before Lock-Door

Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

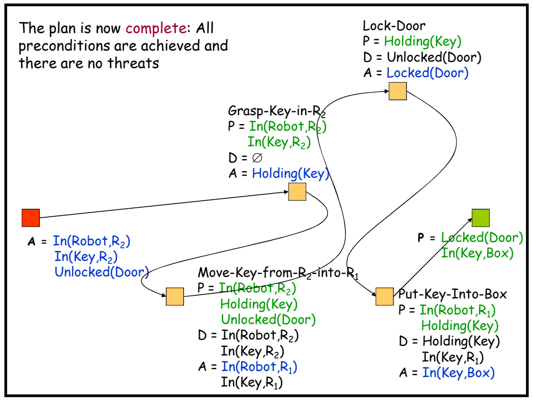
Grasp-Key-in-R_2
P = In(Robot,R_2)
In(Key,R_2)
D = ∅
A = Holding(Key)

A = In(Robot,R_2)
In(Key,R_2)
Unlocked(Door)

P = Locked(Door)
In(Key,Box)

Move-Key-from-R_2-into-R_1
P = In(Robot,R_2)
Holding(Key)
Unlocked(Door)
D = In(Robot,R_2)
In(Key,R_2)
A = In(Robot,R_1)
In(Key,R_1)

Put-Key-Into-Box
P = In(Robot,R_1)
Holding(Key)
D = Holding(Key)
In(Key,R_1)
A = In(Key,Box)



The plan is now complete: All preconditions are achieved and there are no threats

Lock-Door
P = Holding(Key)
D = Unlocked(Door)
A = Locked(Door)

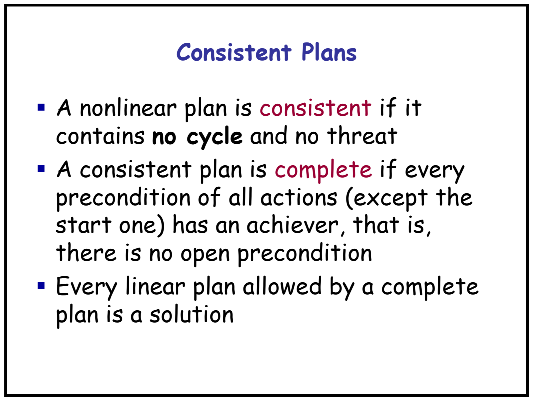
Grasp-Key-in-R_2
P = In(Robot,R_2)
In(Key,R_2)
D = ∅
A = Holding(Key)

A = In(Robot,R_2)
In(Key,R_2)
Unlocked(Door)

P = Locked(Door)
In(Key,Box)

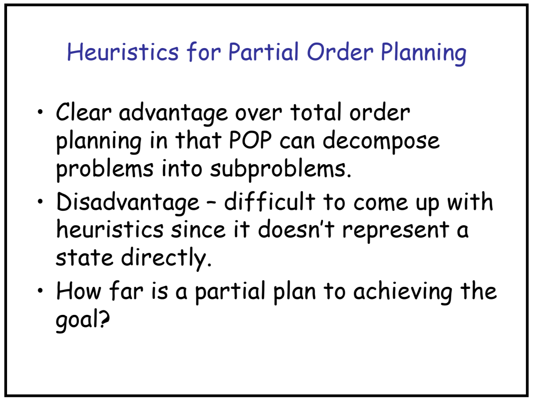
Move-Key-from-R_2-into-R_1
P = In(Robot,R_2)
Holding(Key)
Unlocked(Door)
D = In(Robot,R_2)
In(Key,R_2)
A = In(Robot,R_1)
In(Key,R_1)

Put-Key-Into-Box
P = In(Robot,R_1)
Holding(Key)
D = Holding(Key)
In(Key,R_1)
A = In(Key,Box)



Consistent Plans

- A nonlinear plan is consistent if it contains **no cycle** and no threat
- A consistent plan is complete if every precondition of all actions (except the start one) has an achiever, that is, there is no open precondition
- Every linear plan allowed by a complete plan is a solution



Heuristics for Partial Order Planning

- Clear advantage over total order planning in that POP can decompose problems into subproblems.
- Disadvantage – difficult to come up with heuristics since it doesn't represent a state directly.
- How far is a partial plan to achieving the goal?

Where can heuristics be used?

- Select a partial plan to refine – this is not really shown in our examples
  - Choose the partial plan with the fewest open preconditions
    - Overestimates cost when there are actions that achieve multiple preconditions
    - Underestimates cost when there are negative interactions between steps
      - Example: a set of predonditions P1, P2, P3 where P1 is satisfied in the initial state. But, action for achieving P2 has ¬P1 as one of its effect, so now must plan for an action for achieving P1.

Where (else) can heuristics be used?

- Selecting the open precondition to work on in a partial plan
  - Most constrained precondition heuristic: select the open precondition for which there are the fewest actions for achieving it.
    - Allows you to fail early (if no action can achieve it, need to find out fast)
    - Need to eventually achieve it, so might as well achieve it early because it might place further constraints on other actions.

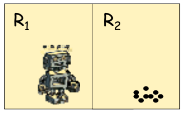
Planning Graph to Compute (Better) Heuristics

- Plan graph consists of levels corresponding to time steps in a plan.
- Level 0 = initial state.
- Each level consists of
  - Literals that could be true at that time step (depending on which actions were executed in prior state)
  - Actions that could have their preconditions satisfied at that time step
- Note: The GRAPHPLAN algorithm extracts a solution directly from a plan graph…

Planning Graph

- May be optimistic about the minimum number of time steps needed to achieve a literal (because doesn't record all negative interactions)
- Does provide a good estimate of how difficult it is to achieve a given literal from the initial state.
- NOTE: assume all actions cost 1 – so want to make a plan with fewest actions!
- Works for proposition planning problems only – NO VARIABLES!

Vacuum Cleaning Robot



Initial State:

In(Robot, R_1) $\wedge$ Clean(R_1)

GOAL:

Clean(R_1) ^ Clean(R_2)

Action Representation

Right

- Precondition = In(Robot, R_1)
- Delete-list = In(Robot, R_1)
- Add-list = In(Robot, R_2)

Other Actions

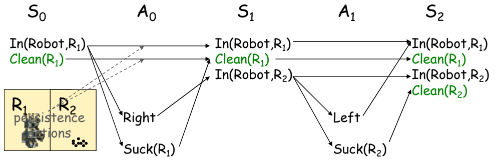
Left

- P = In(Robot, R_2)
- D = In(Robot, R_2)
- A = In(Robot, R_1)

Suck(r)

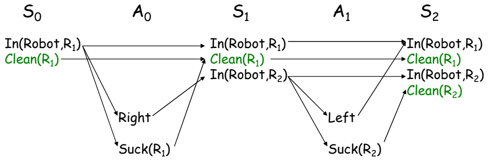
- P = In(Robot, r)
- D = ∅ [empty list]
- A = Clean(r)

Planning Graph for a State of the Vacuum Robot



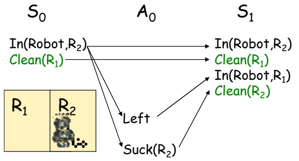
- S_0 contains the state's propositions (here, the initial state)
- A_0 contains all actions whose preconditions appear in S_0
- S_1 contains all propositions that were in S_0 or are contained in the add lists of the actions in A_0
- So, S_1 contains all propositions that may be true in the state reached after the first action
- A_1 contains all actions whose preconditions appear in S_1, hence that may be executed in the state reached after executing the first action. Etc...
- NOTE: Right, and Suck(R1) should be in A1!!!

Planning Graph for a State of the Vacuum Robot



- The value of i such that S_i contains all the goal propositions is called the level cost of the goal (here i=2)
- By construction of the planning graph, it is a lower bound on the number of actions needed to reach the goal
- In this case, 2 is the actual length of the shortest path to the goal

Planning Graph for Another State

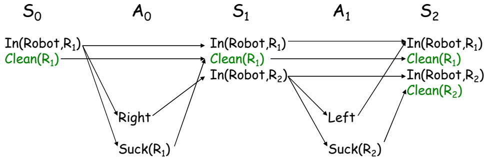


- The level cost of the goal is 1, which again is the actual length of the shortest path to the goal

Application of Planning Graphs to Forward Planning

- Compute the planning graph of each generated state [simply update the graph plan at parent node]
- Stop computing the planning graph when:
  - Either the goal propositions are in a set S_i [then i is the level cost of the goal]
  - Or when $S_{i+1} = S_i$ [then the current state is not on a solution path]
- Set the heuristic h(N) of a node N to the level cost of the goal
- h is a consistent heuristic for unit-cost actions
- Hence, A* using h yields a solution with minimum number of actions

Size of Planning Graph

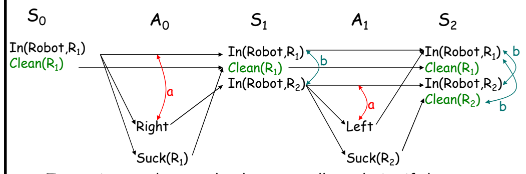


- An action appears at most once (delete)
- A proposition is added at most once and each S_k ($k \neq i$) is a strict superset of S_{k-1}
- So, the number of levels is bounded by Min{number of actions, number of propositions}
- In contrast, the state space can be exponential in the number of propositions
- The computation of the planning graph may save a lot of unnecessary search work

Improvement of Planning Graph: Mutual Exclusions (mutex links)

- Goal: Refine the level cost of the goal to be a more accurate estimate of the number of actions needed to reach it
- Method: Detect obvious exclusions among actions at the same level and among propositions at the same level

Improvement of Planning Graph: Mutual Exclusions



a. Two actions at the same level are mutually exclusive if the same proposition appears in the add list of one and the delete list of the other

b. Two propositions in S_k are mutually exclusive if no single action in A_{k-1} contains both of them in its add list and every pair of actions in A_{k-1} that could achieve them are mutually exclusive

Mutex Relations Between Actions

- *Inconsistent effects*: one action negates an effect of the other. E.g., Eat(Cake) and Have(Cake)
- *Inteference*: one of the effects of one action is the negation of a preconditon of the other. E.g., Eat(Cake) interferes with the persistence of Have(Cake)
- *Competing Needs*: one of the preconditions of one action is mutually exclusive with a precondition of another. E.g., Bake(Cake) and Eat(Cake) are mutex because they compete for the Have(Cake) precondition.

2 literals are mutex if…

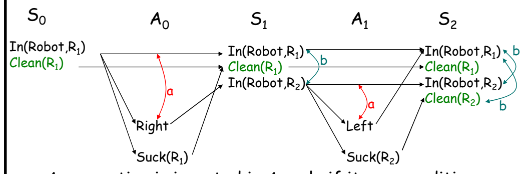
A mutex relation holds between two literals at the same level if:

- One is the negation of the other

or

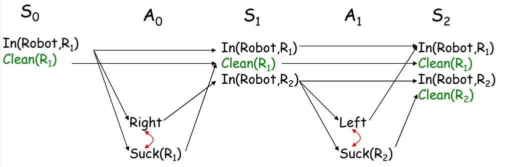
- If each possible pair of actions that could achieve the two literals is mutually exclusive

Improvement of Planning Graph: Mutual Exclusions



- A new action is inserted in A_k only if its preconditions are in S_k and no two of them are mutually exclusive
- The computation of the planning graph ends when:
  - Either the goal propositions are in a set S_i and no two of them are mutually exclusive
  - Or when two successive sets S_{i+1} and S_i contain the same propositions with the same mutual exclusions

Another Possible Mutual Exclusion (NOT COVERED)



- Any two non-persistence actions at the same level are mutually exclusive (→ serial planning graph)
- Then an action may re-appear at a new level if it leads to removing mutual exclusions among propositions
- In general, the more mutual exclusions, the longer and the bigger the planning graph

Heuristics

- Pre-compute the planning graph of the initial state until it levels off
- For each node N added to the search tree, set h(N) to the maximum level cost of any open precondition in the plan associated with N or to the sum of these level costs

Consistent Heuristic for Backward Planning

A consistent heuristic can be computed as follows :

1. Pre-compute the planning graph of the initial state until it levels off
2. For each node N added to the search tree, set h(N) to the level cost of the goal associated with N

If the goal associated with N can't be satisfied in any set S_k of the planning graph, it can't be achieved (prune it!)

Only one planning graph is pre-computed

- Mutual exclusions in planning graphs only deal with very simple interferences
- State constraints may help detect early some interferences in backward planning
- In general, however, interferences lead linear planning to explore un-fructuous paths

Extracting a Plan – Search Problem

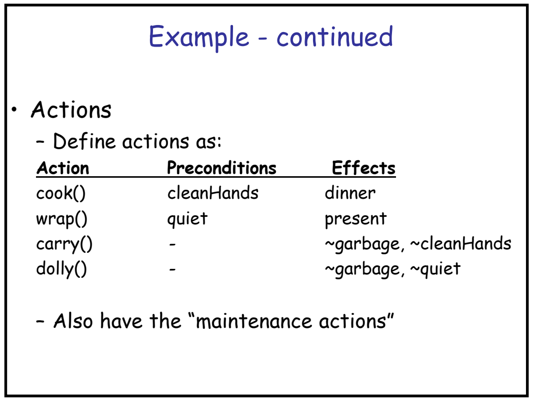
- Try to do if all goal literals true and not mutex at ending level Si.
- Initial State: level Si along with goals
- Actions: select any conflict-free subset of the action in Ai-1 whose effects cover the goals in the state. (New State is Si-1 with preconditions of selected actions.)
- Goal: reach state at level S0 such that goals satisfied.

Another example…

By
Ruti Glick
Bar-Ilan University

Example - Dinner

- World predicates
  - garbage
  - cleanhands
  - quiet
  - present
  - Dinner
- initial state:
  - s0: {garbage, cleanHands, quiet}
- Goal
  - *g*: {dinner, present, ~garbage}

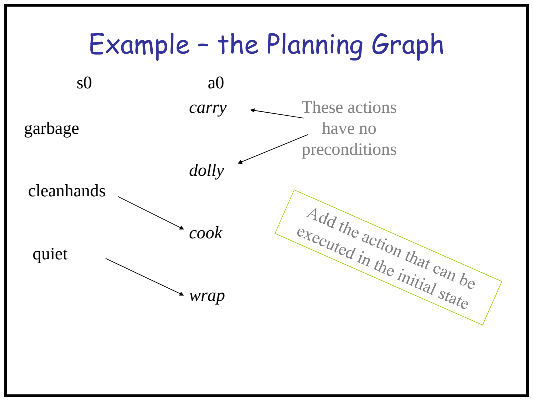


Example - continued

- Actions
  - Define actions as:

Action	Preconditions	Effects
cook()	cleanHands	dinner
wrap()	quiet	present
carry()	-	~garbage, ~cleanHands
dolly()	-	~garbage, ~quiet

  - Also have the "maintenance actions"



Example – the Planning Graph

s0

a0

garbage

cleanhands

quiet

carry

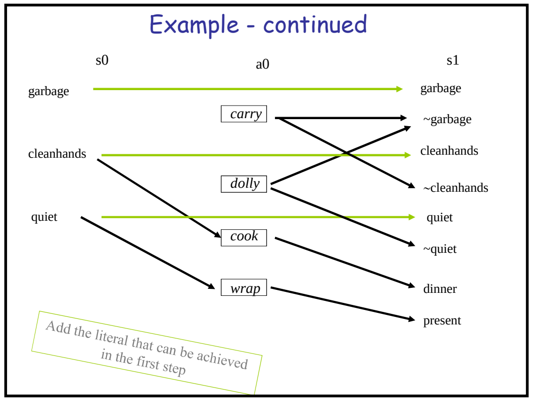
dolly

cook

wrap

These actions have no preconditions

Add the action that can be executed in the initial state



Example - continued

s0

a0

s1

garbage

cleanhands

quiet

carry

dolly

cook

wrap

garbage

~garbage

cleanhands

~cleanhands

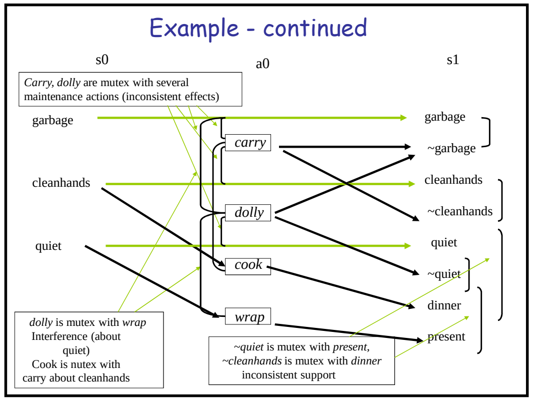
quiet

~quiet

dinner

present

Add the literal that can be achieved in the first step



Example - continued

s0

a0

s1

Carry, *dolly* are mutex with several maintenance actions (inconsistent effects)

garbage

cleanhands

quiet

carry

dolly

cook

wrap

garbage

~garbage

cleanhands

~cleanhands

quiet

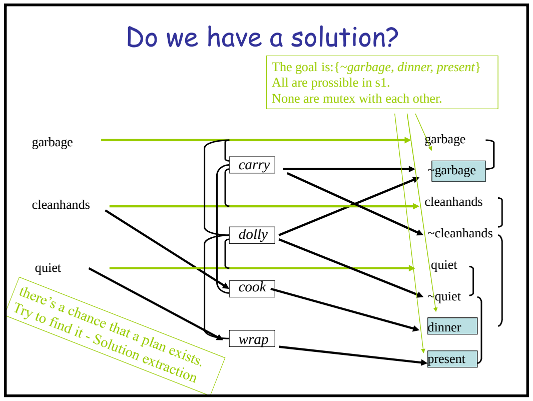
~quiet

dinner

present

dolly is mutex with *wrap*
Interference (about quiet)
Cook is nutex with carry about cleanhands

~quiet is mutex with *present*,
~cleanhands is mutex with *dinner*
inconsistent support



Do we have a solution?

The goal is:{*~garbage, dinner, present*}
All are prossible in s1.
None are mutex with each other.

garbage

cleanhands

quiet

carry

dolly

cook

wrap

garbage

~garbage

cleanhands

~cleanhands

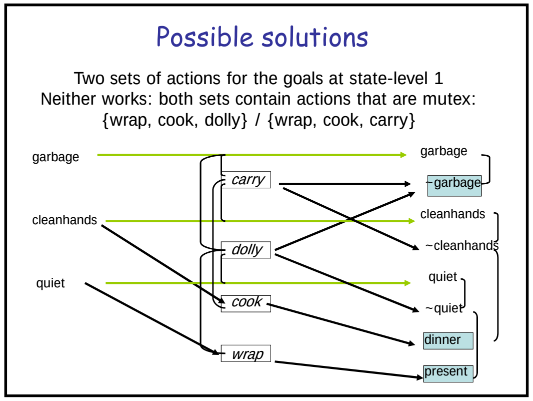
quiet

~quiet

dinner

present

there's a chance that a plan exists.
Try to find it - Solution extraction



Possible solutions

Two sets of actions for the goals at state-level 1
Neither works: both sets contain actions that are mutex:
{wrap, cook, dolly} / {wrap, cook, carry}

garbage

cleanhands

quiet

carry

dolly

cook

wrap

garbage

~garbage

cleanhands

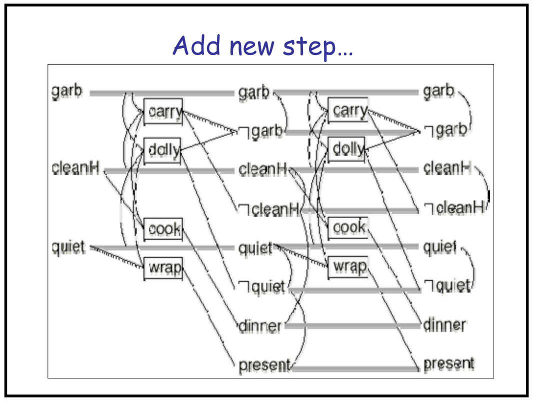
~cleanhands

quiet

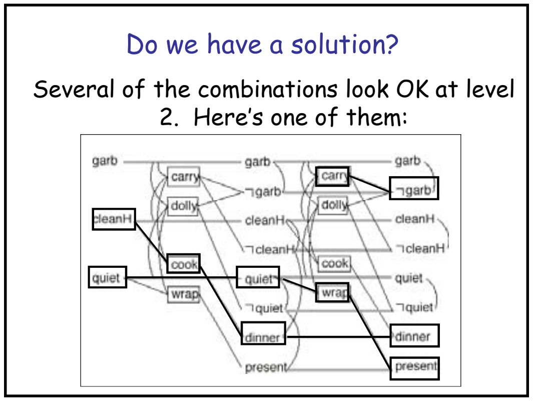
~quiet

dinner

present

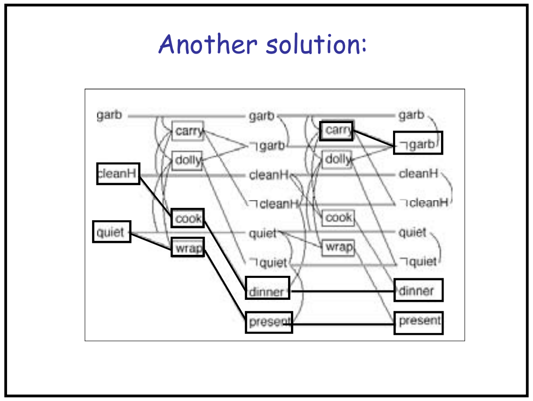


Add new step…



Do we have a solution?

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



Another solution: