Planning

- Search vs. planning
- STRIPS operators
- Partial-order planning

What we have so far

- Can TELL KB about new percepts about the world
- KB maintains model of the current world state
- Can ASK KB about any fact that can be inferred from KB

How can we use these components to build a planning agent,

i.e., an agent that constructs plans that can achieve its goals, and that then executes these plans?

Example: Robot Manipulators

- Example: (courtesy of Martin Rohrmeier)

Remember: Problem-Solving Agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT( p) returns an action
    inputs: p, a percept
    static: s, an action sequence, initially empty
            state, some description of the current world state
            g, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, p)
    if s is empty then
        g ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, g)
        s ← SEARCH( problem)
    action ← RECOMMENDATION(s, state)
    s ← REMAINDER(s, state)
    return action
```


Note: This is *offline* problem-solving. *Online* problem-solving involves acting w/o complete knowledge of the problem and environment

Simple planning agent

- Use percepts to build model of current world state

- IDEAL-PLANNER: Given a goal, algorithm generates plan of action

- STATE-DESCRIPTION: given percept, return initial state description in format required by planner

- MAKE-GOAL-QUERY: used to ask KB what next goal should be

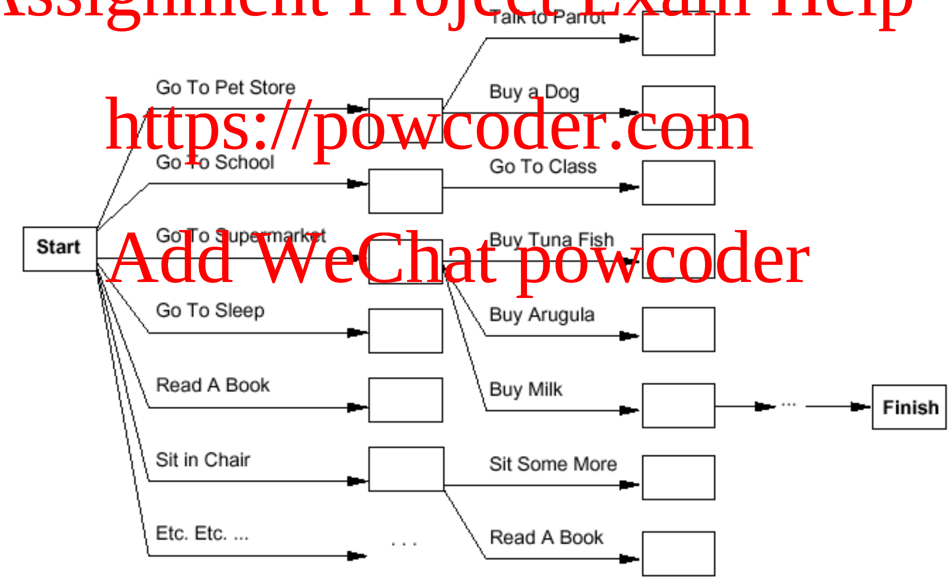
A Simple Planning Agent

```
function SIMPLE-PLANNING-AGENT(percept) returns an action
    static:        KB, a knowledge base (includes action descriptions)
                   p, a plan (initially, NoPlan)
                   t, a time counter (initially 0)
    local variables: G, a goal
                   current, a current state description
    TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
    current ← STATE-DESCRIPTION(KB, t)
    if p = NoPlan then
            G ← ASK(KB, MAKE-GOAL-QUERY(t))
            p ← IDEAL-PLANNER(current, G, KB)
    if p = NoPlan or p is empty then
            action ← NoOp
    else
            action ← FIRST(p)
            p ← REST(p)          Like popping from a stack
    TELL(KB, MAKE-ACTION-SENTENCE(action, t))
    t ← t+1
    return action
```

Search vs. planning

Consider the task *get milk, bananas, and a cordless drill*

Standard search algorithms seem to fail miserably:



After-the-fact heuristic/goal test inadequate

Search vs. planning

Planning systems do the following:

1) open up action and goal representation to allow selection
2) divide-and-conquer by subgoaling
3) relax requirement for sequential construction of solutions

	Search	**Planning**
States	Lisp data structures	Logical sentences
Actions	Lisp code	Preconditions/outcomes
Goal	Lisp code	Logical sentence (conjunction)
Plan	Sequence from S_0	Constraints on actions

Planning in situation calculus

$PlanResult(p, s)$ is the situation resulting from executing p in s

$$PlanResult([], s) = s$$
$$PlanResult([a|p], s) = PlanResult(p, Result(a, s))$$

Initial state $At(Home, S_0) \land \neg Have(Milk, S_0) \land \ldots$

Actions as Successor State axioms

$Have(Milk, Result(a, s)) \Leftrightarrow$
$[(a = Buy(Milk) \land At(Supermarket, s)) \lor (Have(Milk, s) \land a \neq \ldots)]$

Query

$$s = PlanResult(p, S_0) \land At(Home, s) \land Have(Milk, s) \land \ldots$$

Solution

$$p = [Go(Supermarket), Buy(Milk), Buy(Bananas), Go(HWS), \ldots]$$

Principal difficulty: unconstrained branching, hard to apply heuristics

Basic representation for planning

- Most widely used approach: uses STRIPS language

- states: conjunctions of function-free ground literals (I.e., predicates applied to constant symbols, possibly negated); e.g.,

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

- goals: also conjunctions of literals; e.g.,

  $At(Home) \wedge Have(Milk) \wedge Have(Bananas) \wedge Have(Drill)$

  but can also contain variables (implicitly universally quant.); e.g.,

  $At(x) \wedge Sells(x, Milk)$

Planner vs. theorem prover

- Planner: ask for sequence of actions that makes goal true if executed

- Theorem prover: ask whether query sentence is true given KB

STRIPS operators

Tidily arranged actions descriptions, restricted language

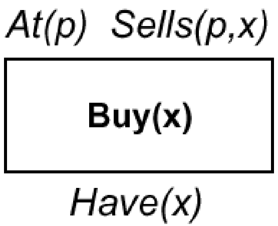
ACTION: $Buy(x)$
PRECONDITION: $At(p), Sells(p, x)$
EFFECT: $Have(x)$

[Note: this abstracts away many important details!]

Restricted language $\Rightarrow$ efficient algorithm

Precondition: conjunction of positive literals

Effect: conjunction of literals



Types of planners

- Situation space planner: search through possible situations

- Progression planner: start with initial state, apply operators until goal is reached

  Problem: high branching factor!

- Regression planner: start from goal state and apply operators until start state reached

  Why desirable? usually many more operators are applicable to initial state than to goal state.

  Difficulty: when want to achieve a conjunction of goals

Initial STRIPS algorithm: situation-space regression planner

State space vs. plan space

Standard search: node = concrete world state

Planning search: node = partial plan

Defn: open condition is a precondition of a step not yet fulfilled

Operators on partial plans:

- add a link from an existing action to an open condition
- add a step to fulfill an open condition
- order one step wrt another

Gradually move from incomplete/vague plans to complete, correct plans

Operations on plans

- Refinement operators: add constraints to partial plan

- Modification operator: every other operators

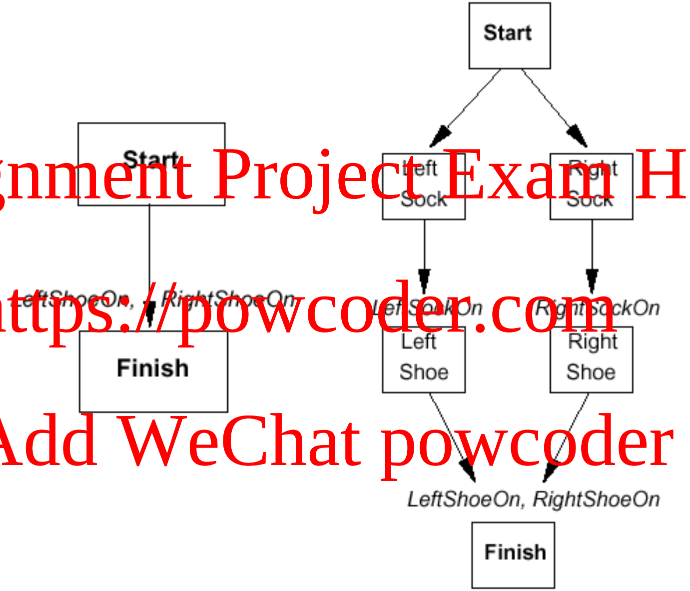
Types of planners

- Partial order planner: some steps are ordered, some are not

- Total order planner: all steps ordered (thus, plan is a simple list of steps)

- Linearization: process of deriving a totally ordered plan from a partially ordered plan.

Partially ordered plans



A plan is complete iff every precondition is achieved

A precondition is achieved iff it is the effect of an earlier step and no possibly intervening step undoes it

Plan

We formally define a plan as a data structure consisting of:

- Set of plan steps (each is an operator for the problem)
- Set of step ordering constraints

  e.g. $\prec$ B means "A before B"

- Set of variable binding constraints

  e.g., v = x where v variable and x constant or other variable

- Set of causal links

  e.g., $A \xrightarrow{c} B$ means "A achieves c for B"

POP algorithm sketch

function POP(*initial*, *goal*, *operators*) **returns** *plan*

    plan ← MAKE-MINIMAL-PLAN(*initial*, *goal*)
    loop do
        if SOLUTION?(*plan*) **then return** *plan*
        S_{need}, c ← SELECT-SUBGOAL(*plan*)
        CHOOSE-OPERATOR(*plan*, *operators*, S_{need}, c)
        RESOLVE-THREATS(*plan*)
    end

---

function SELECT-SUBGOAL(*plan*) **returns** S_{need}, c

    pick a plan step S_{need} from STEPS(*plan*)
        with a precondition c that has not been achieved
    return S_{need}, c

POP algorithm (cont.)

procedure CHOOSE-OPERATOR(*plan*, *operators*, S_{need}, *c*)

  choose a step S_{add} from *operators* or STEPS(*plan*) that has *c* as an effect
  if there is no such step **then fail**
  add the causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS(*plan*)
  add the ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS(*plan*)
  if S_{add} is a newly added step from *operators* **then**
    add S_{add} to STEPS(*plan*)
    add $Start \prec S_{add} \prec Finish$ to ORDERINGS(*plan*)

---

procedure RESOLVE-THREATS(*plan*)

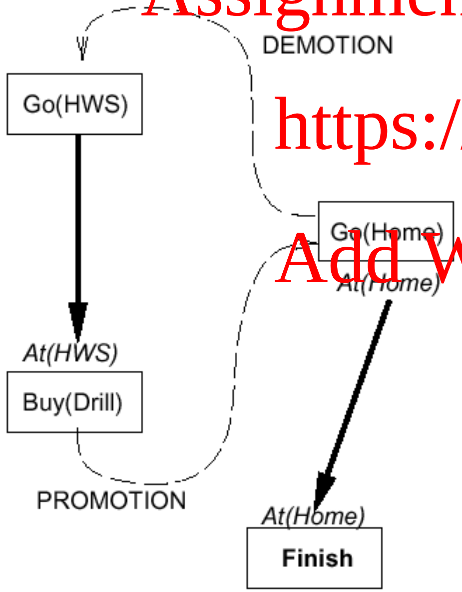
  for each S_{threat} that threatens a link $S_i \xrightarrow{c} S_j$ in LINKS(*plan*) **do**
    choose either
      Demotion: Add $S_{threat} \prec S_i$ to ORDERINGS(*plan*)
      Promotion: Add $S_j \prec S_{threat}$ to ORDERINGS(*plan*)
    if not CONSISTENT(*plan*) **then fail**
  end

POP is sound, complete, and systematic (no repetition)

Extensions for disjunction, universals, negation, conditionals

Clobbering and promotion/demotion

A <u>clobberer</u> is a potentially intervening step that destroys the condition achieved by a causal link. E.g., $Go(Home)$ clobbers $At(HWS)$:

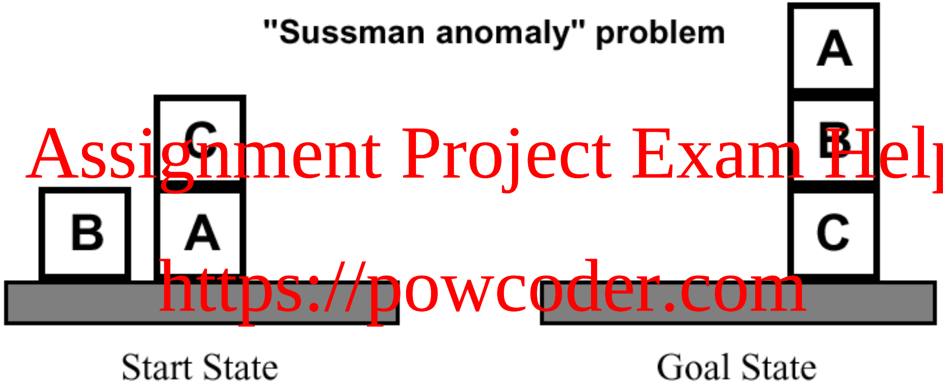


<u>Demotion</u>: put before $Go(HWS)$

<u>Promotion</u>: put after $Buy(Drill)$

Example: block world

"Sussman anomaly" problem



Clear(x) On(x,z) Clear(y)

PutOn(x,y)

~On(x,z) ~Clear(y)
Clear(z) On(x,y)

Clear(x) On(x,z)

PutOnTable(x)

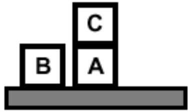
~On(x,z) Clear(z) On(x,Table)

+ several inequality constraints

Example (cont.)

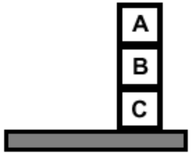
START

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

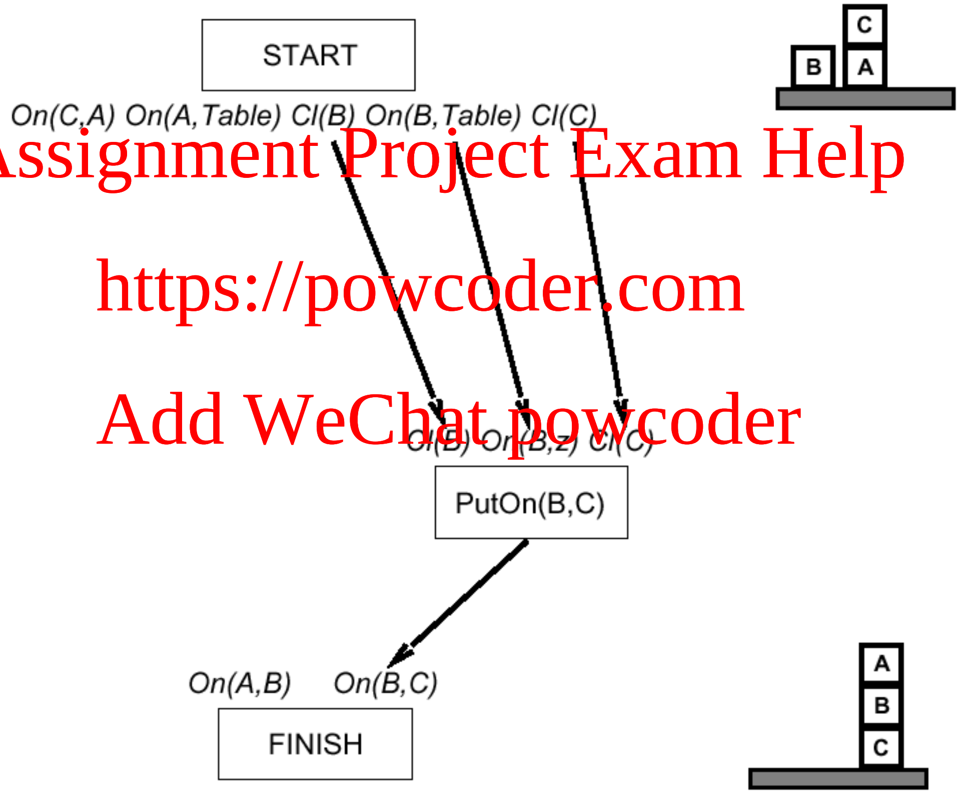


On(A,B) *On(B,C)*

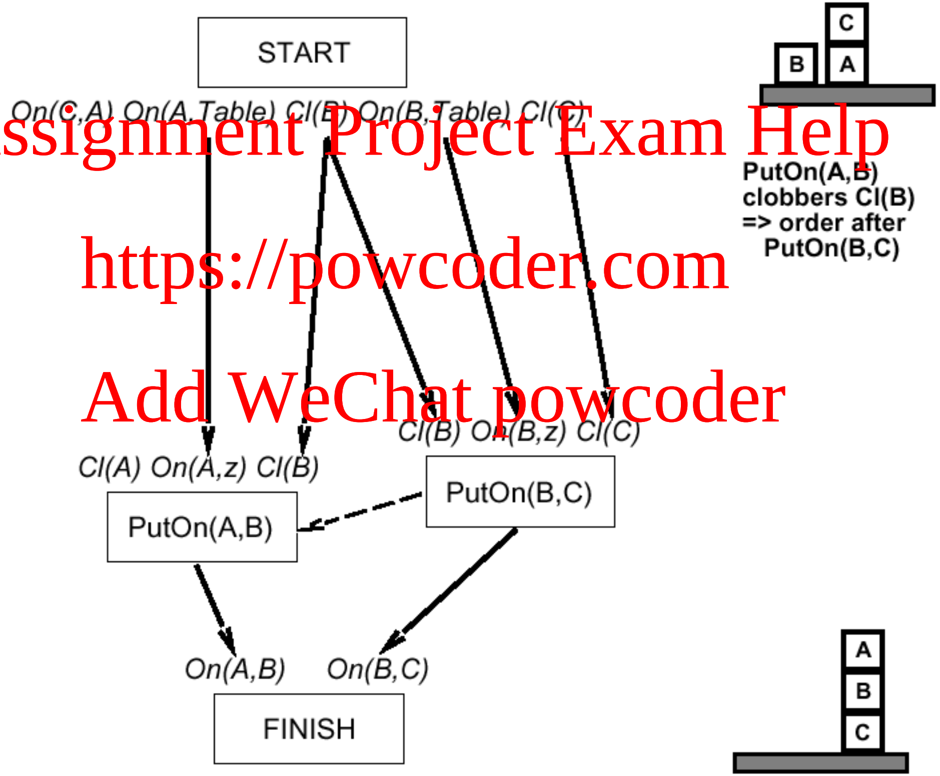
FINISH



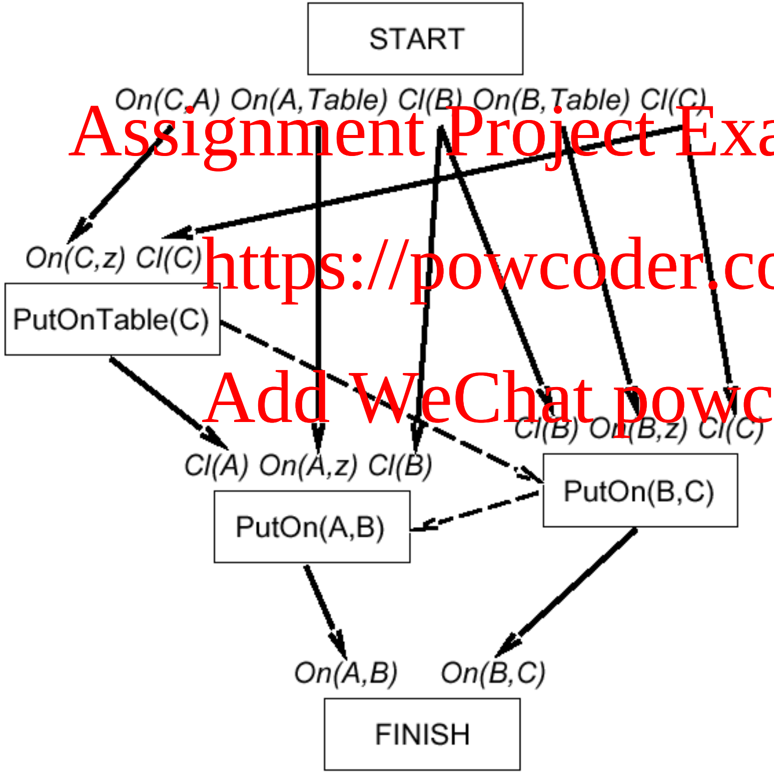
Example (cont.)

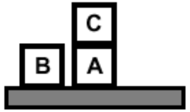


Example (cont.)



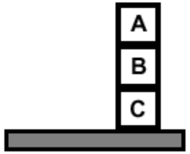
Example (cont.)



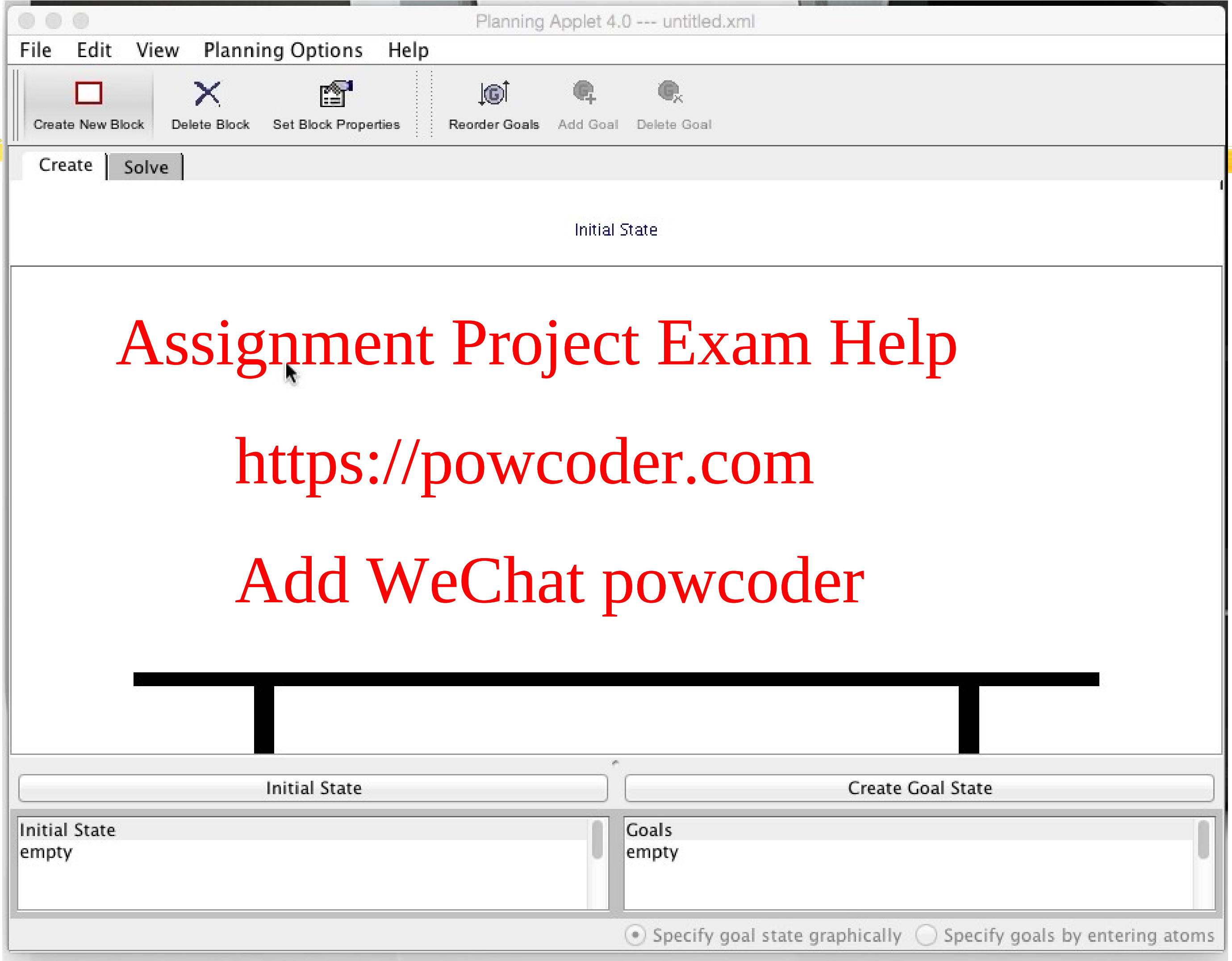


PutOn(A,B)
clobbers Cl(B)
=> order after
PutOn(B,C)

PutOn(B,C)
clobbers Cl(C)
=> order after
PutOnTable(C)



Demo


Planning Applet 4.0 --- untitled.xml
File
Edit
View
Planning Options
Help
Create New Block
Delete Block
Set Block Properties
Reorder Goals
Add Goal
Delete Goal
Create
Solve
Initial State
Initial State
Create Goal State
Initial State
empty
Goals
empty
Specify goal state graphically
Specify goals by entering atoms