Planning in AI

Sukanta Ghosh

LOVELY PROFESSIONAL UNIVERSITY

What is AI Planning ?

- Planning is the task of finding a procedural course of action for a declaratively described system to reach its goals while optimizing overall performance measures.

- Planning is the task of coming up with a sequence of actions that will achieve a goal.

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 vs. problem solving

- Planning and problem solving methods can often solve the same sorts of problems
- Planning is more powerful because of the representations and methods used
- States, goals, and actions are decomposed into sets of sentences (usually in first-order logic)
- Subgoals can be planned independently, reducing the complexity of the planning problem
- Search often proceeds through a much smaller *plan space* rather than *state space* (though there are also state-space planners) by considering only relevant actions

Planning vs. Problem Solving

- Planning agent is very similar to problem solving agent
  - Constructs plans to achieve goals, then executes them

- Planning agent is different from problem solving agent in:
  - Representation of goals, states, actions
  - Use of explicit, logical representations
  - Way it searches for solutions

Planning vs. Problem Solving

- Planning systems do the following:
- divide-and-conquer
- relax requirement for sequential construction of solutions

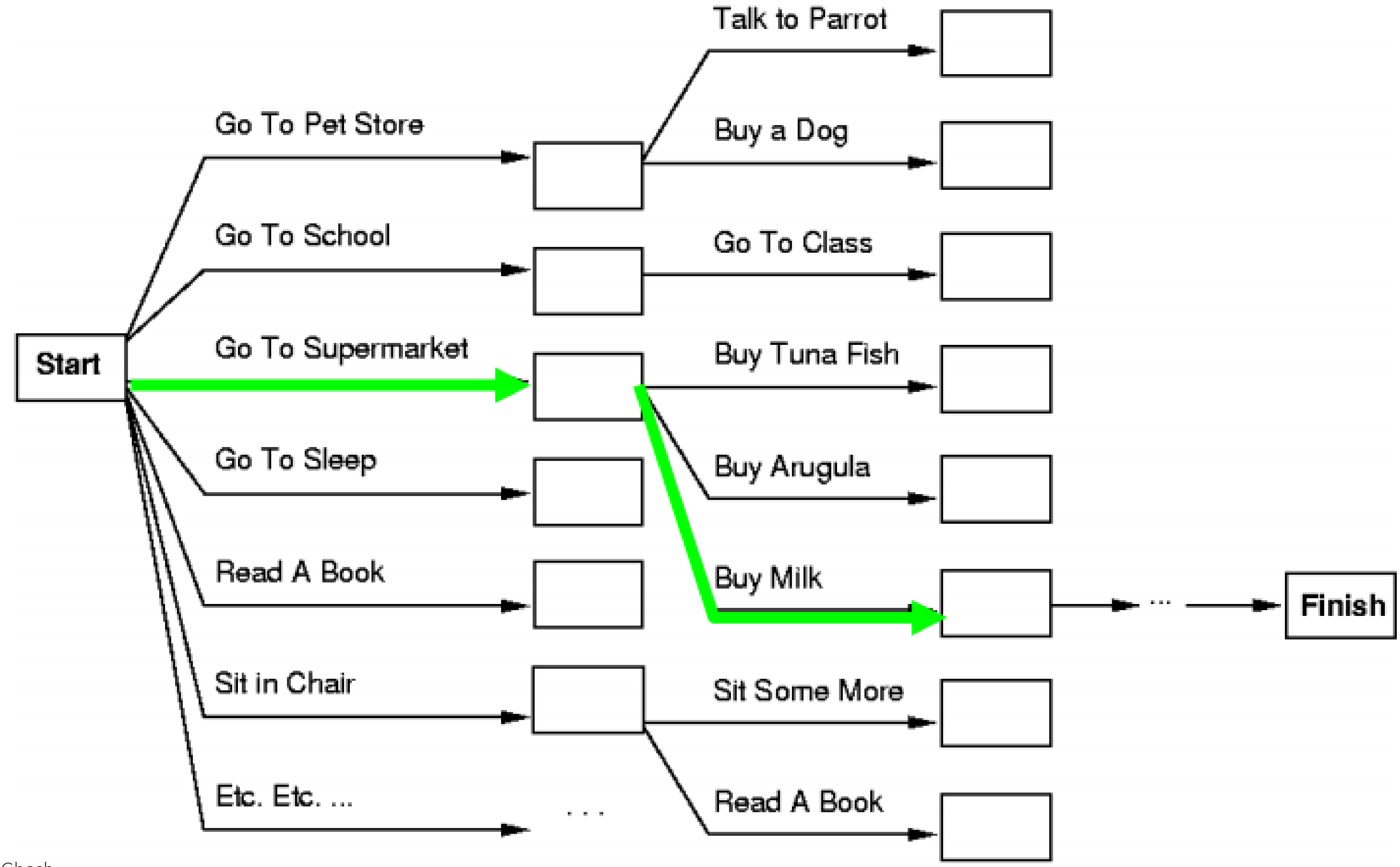
	Problem Sol.	Planning
States	data structures	logical sentences
Actions	code	preconditions/outcomes
Goal	code	logical sentences
Plan	sequence from s0	constraints on actions

Problems with Standard Search

- Overwhelmed by irrelevant actions
- Finding a good heuristic function is difficult
- Cannot take advantage of problem decomposition

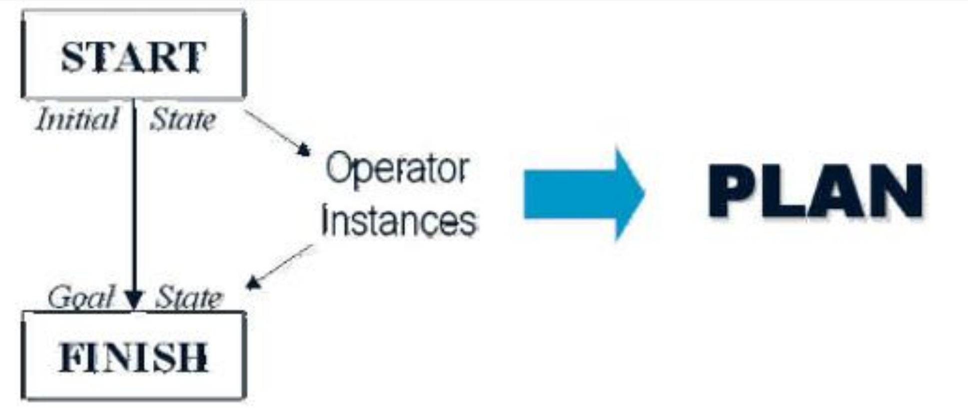
Example

- Consider the task: get milk, bananas, and a cordless drill
- Standard search algorithms seem to fail miserably
- Why? Huge branching factor & heuristics


Start
Go To Pet Store
Go To School
Go To Supermarket
Go To Sleep
Read A Book
Sit in Chair
Etc. Etc. ...
Talk to Parrot
Buy a Dog
Go To Class
Buy Tuna Fish
Buy Arugula
Buy Milk
Sit Some More
Read A Book
Finish

Purpose of Planning

- The purpose of planning is to find a sequence of actions that achieves a given goal when performed starting in a given state. In other words, given a set of operator instances (defining the possible primitive actions by the agent), an initial state description, and a goal state description or predicate, the planning agent computes a plan.



What is a plan?

- 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.

A Simple Planning Agent

- Earlier we saw that problem-solving agents are able to plan ahead - to consider the consequences of sequences of actions - before acting.
- We also saw that a knowledge based agents can select actions based on explicit, logical representations of the current state and the effects of actions.
- This allows the agent to succeed in complex, inaccessible environments that are too difficult for a problem-solving agent
- Problem Solving Agents + Knowledge-based Agents = Planning Agents

Algorithm of a simple planning agent

- Generate a goal to achieve
- Construct a plan to achieve goal from current state
- Execute plan until finished
- Begin again with new goal

Assumptions

- A simple planning agent create and use plans based on the following assumptions:
- Atomic time: each action is indivisible
- No concurrent actions allowed
- Deterministic actions: result of each actions is completely determined by the definition of the action, and there is no uncertainty in performing it in the world.
- Agent is the sole cause of change in the world.
- Agent is omniscient: has complete knowledge of the state of the world
- Closed world assumption: everything known to be true in the world is included in a state description. Anything not listed is false

Basic Element of Searching Approach

- Representations of actions: programs that develop successor state descriptions which represent actions.
- Representation of state: every state description is complete. This is because a complete description of the initial state is given, and actions are represented by a program that creates complete state descriptions.
- Representation of goals: a problem solving agent has only information about it's goal, which is in terms of a goal test and the heuristic function.
- Representation of plans: in problem solving, the solution is a sequence of actions.

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)

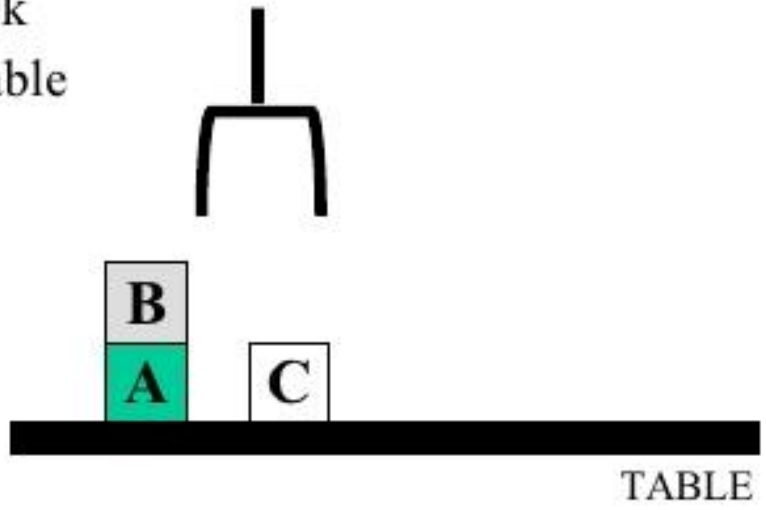
ontable(c)

on(b,a)

handempty

clear(b)

clear(c)



Major approaches

- GPS / STRIPS
- Situation calculus
- Partial order planning
- Hierarchical decomposition (HTN planning)
- Planning with constraints (SATplan, Graphplan)
- Reactive planning

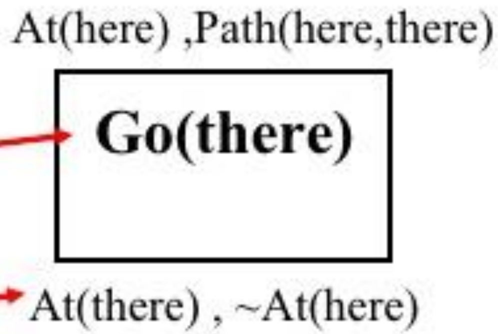
Basic representations for planning

- Classic approach first used in the **STRIPS** planner circa 1970
- States represented as a conjunction of ground literals
  - at(Home) ^ ~have(Milk) ^ ~have(bananas) ...
- Goals are conjunctions of literals, but may have variables which are assumed to be existentially quantified
  - at(?x) ^ have(Milk) ^ have(bananas) ...
- Do not need to fully specify state
  - Non-specified either don't-care or assumed false
  - Represent many cases in small storage
  - Often only represent changes in state rather than entire situation
- Unlike theorem prover, not seeking whether the goal is true, but is there a sequence of actions to attain it

Operator/action representation

- Operators contain three components:
  - **Action description**
  - **Precondition** - conjunction of positive literals
  - **Effect** - conjunction of positive or negative literals which describe how situation changes when operator is applied
- Example:

  Op[Action: Go(there),
  Precond: At(here) ^ Path(here,there),
  Effect: At(there) ^ ~At(here)]



- All variables are universally quantified
- Situation variables are implicit
  - preconditions must be true in the state immediately before operator is applied; effects are true immediately after

Representation of States and Goals in STRIPS

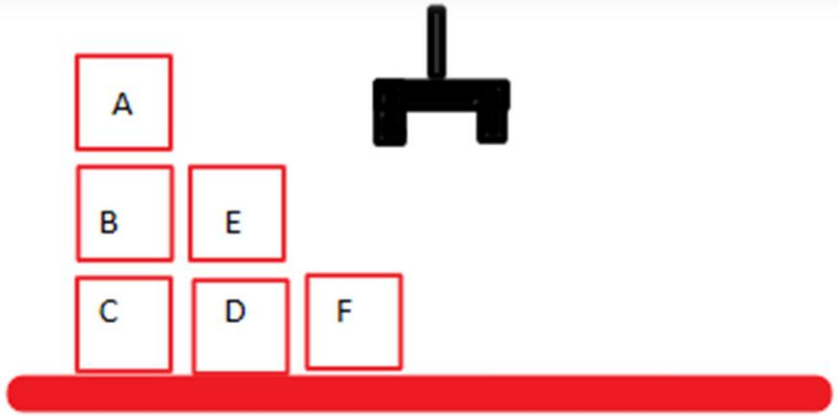
- States are represented by conjunctions of function-free ground literals, that is, predicates applied to constant symbols, possibly negated.
- An example of an initial state is:
- At(Home) /\ -Have(Milk) /\ -Have(Bananas) /\ -Have(Drill) /\ ...
- A state description does not have to be complete.
- We just want to obtain a successful plan to a set of possible complete states.
- But if it does not mention a given positive literal, then the literal can be assumed to be false.

Representation of Actions in Strips

- Strips operators consist of three components
  - action description: what an agent actually returns to the environment in order to do something.
  - precondition: conjunction of atoms (positive literals), that says what must be true before an operator can be applied.
  - effect of an operator: conjunction of literals (positive or negative) that describe how the situation changes when the operator is applied
- Op(ACTION:Go(there), PRECOND:At(here) /\ Path(here, there) EFFECT:At(there) /\ -At(here))

Block World Problem - Using STRIPS

- **Domain Predicate:** On(X,Y); OnTable(X); Holding(X); Clear (X); ArmEmpty()
- **Operators/Actions:**
- PickUp(X), PutDown(X)
- Precondition: OnTable(X); Clear(X); ArmEmpty(); On(X,Y)
- Add: Holding(X)
- Delete: Ontable(X); ArmEmpty(); On(X,Y)
- PutDown(X)
- Precondition: Holding(X)
- Add: OnTable(X); ArmEmpty()
- Delete: Holding(X)
- Stack(X,Y): Stack X on Y
- Unstack(X,Y): Unstack X from Y



How you will define it in terms of predicate?

OnTable(C) ^ OnTable(D) ^ OnTable(F) ^
On(B,C) ^ On(A,B) ^ On(E,D) ^
Clear(A) ^ Clear(E) ^ Clear(F) ^
ArmEmpty()

- Stack(X,Y):-
- Pre: Holding(X), Clear(Y)
- Effect+: On(X,Y), ArmEmpty(), Clear(X)
- Effect-: Holding(X), Clear(Y)

- UnStack(X,Y):-
- Pre: On(X,Y), ArmEmpty(), Clear(X)
- Effect+: Holding(X), Clear(Y)
- Effect-: On(X,Y), ArmEmpty(), Clear(X)

Planning Problem

- Planning Problem = (O, D, S, G)

- O: Set of Operators
- S: Start State
- D: Domain Description
- G: Goal State

- Let G = On(E,C) ^ On(C, F)

In other PDDL language

- Operators/Actions:
- PickUp(X)

• Precondition: OnTable(X); Clear(X); ArmEmpty()	Pre(a)
• Add: Holding(X)	Effect+(a)
• Delete: Ontable(X); ArmEmpty()	Effect-(a)

When we can say action(a) is applicable?

- Action(a) is applicable if pre(a) C (subset) S(any state)
- When a is applied we PROGRESS to get new state S' and is given by:
- S' <- (S\effect-(a)) U (effect+(a)).
- Plan ∏ is a sequence of action and denoted by: ∏ = (a1, a2,….,an).
- **How do I say a given state is a goal state?**
- A. S -> S', when after action a goal g (g is subset of S')

How to validate a test?

- THIS IS HOW I WILL VALIDATE IT
- S = OnTable(C), OnTable(D), OnTable(F), On(B,C), On(A,B), On(E,D), Clear(A), Clear(E), Clear(F), ArmEmpty()
- g = On(A,F)
- p = a1, a2 (a1 = UnStack(X,Y), a2 = Stack(X,Y))
- S' = OnTable(C), OnTable(D), OnTable(F), On(B,C), On(A,F), On(E,D), Clear(A), Clear(E), Clear(B), ArmEmpty()

- g is subset of S'

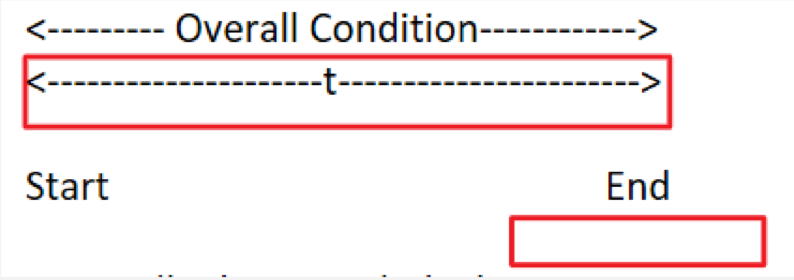
Features of Rich/High Level PDDL

- Domain: Moving from place A to B.

1. Conditional Effect

- Case: Moving a laptop bag from Hostel to Department
- Lappybag: loc(X) -> loc(Y)
- If Z(notebook) is in bag, then Z -> loc(Y)

- 2. Durative Actions
- Represented by duration(intervals).



- Allen's Interval Algebra
- Metric Value: Petrol Example

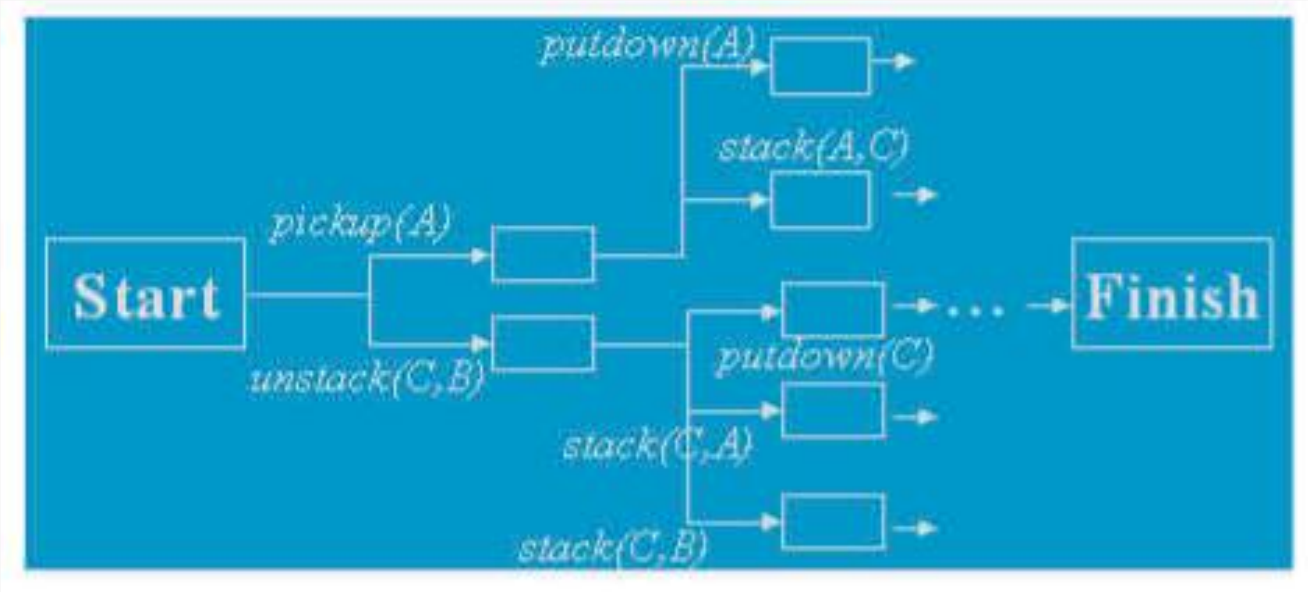
Planning as Search

- There are two main approaches to solving planning problems, depending on the kind of search space that is explored:

    1. Situation-space search
    2. Planning-space search

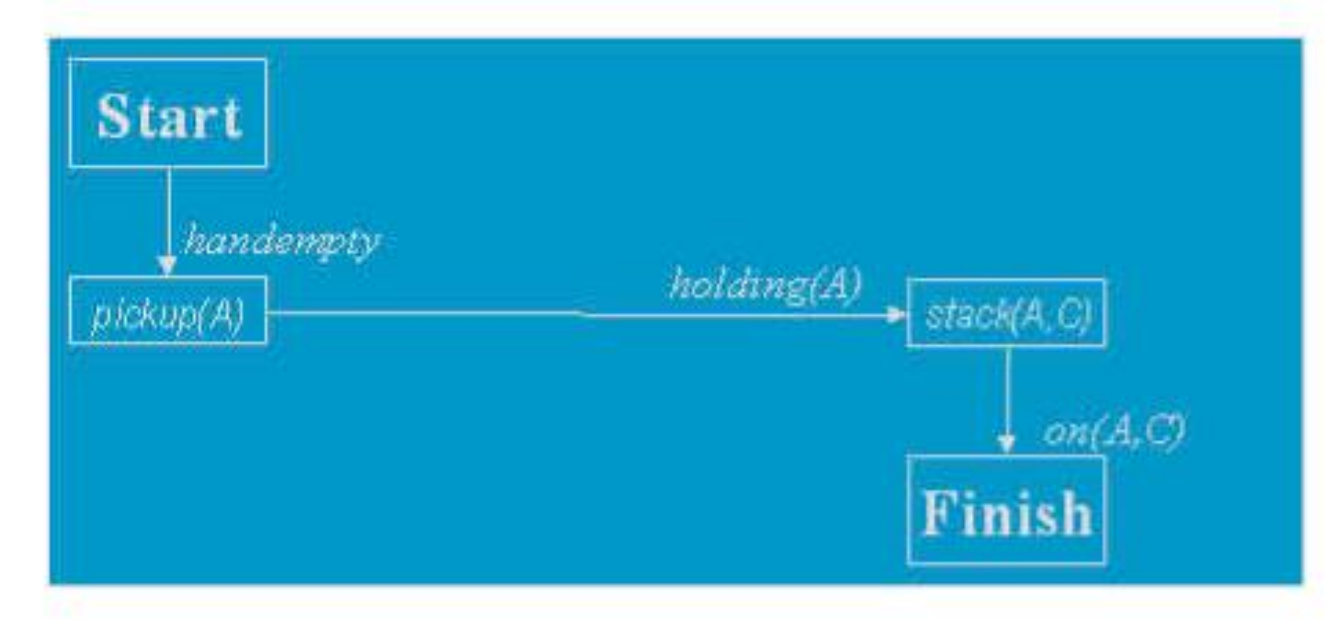
Situation-Space Search

- the search space is the space of all possible states or situations of the world
- initial state defines one node
- a goal node is a state where all goals in the goal state are satisfied
- a solution plan is the sequence of actions (e.g. operator instances) in the path from the start node to a goal node



Plan-Space Search

- the search space is the space of all possible plans
- a node corresponds to a partial plan
- initially we will specify an "initial plan" which is one node in this space
- a goal node is a node containing a plan which is complete, satisfying all of the goals in the goal state
- the node itself contains all of the information for determining a solution plan (e.g. sequence of actions)



Situation-Space Planning Algorithms

- There are 2 approaches to situation-space planning:
  1. Progression situation-space planning
  2. Regression situation-space planning

Progression Planning

- Forward-chaining from initial state to goal state
- Looks just like a state-space search except STRIPS operators are specified instead of a set of next-move functions
- You can use any search method you like (i.e. BFS, DFS, A*)

- Disadvantage: huge search space to explore, so usually very inefficient

Problem Formulation for Progression

- Initial State: Initial state of planning problem (S)
- Action: Applicable to current state. Preconditions must be satisfied.
- Goal Test: Weather new state(S') satisfy the goal of planning(G).
- Step Cost: Each action is 1

Algorithm

1. Start from initial state
2. Find all operators whose preconditions are true in the initial state
3. Compute effects of operators to generate successor states
4. Repeat steps #2-#3 until a new state satisfies the goal conditions

Blocks World example

Step	State	Applicable Operators	Operator Applied
#1	ontable(A) $\wedge$ ontable(B) $\wedge$ on(C, B) $\wedge$ clear(A) $\wedge$ clear(C) $\wedge$ handempty	pickup(A) unstack(C,B)	pickup(A)
#2	~ontable(A) $\wedge$ ontable(B) $\wedge$ on(C, B) $\wedge$ ~clear(A) $\wedge$ clear(C) $\wedge$ ~handempty $\wedge$ holding(A)	putdown(A) stack(A,C)	stack(A,C)
#3	ontable(B) $\wedge$ on(C, B) $\wedge$ on(A, C) $\wedge$ clear(A) $\wedge$ ~clear(C) $\wedge$ handempty $\wedge$ ~holding(A)	Matches goal state so STOP!	

Regression Planning

- Backward-chaining from goal state to initial state.
- A plan is a sequence of STRIPS operators
- The goal state must unify with at least one of the positive literals in the operator's effect.
- Its preconditions must hold in the previous situation, and these become subgoals which might be satisfied by the initial conditions
- Perform backward chaining from goal
- Again, this is just state-space search using STRIPS operators

- Regression situation-space planning is usually more efficient than progression because many operators are applicable at each state, yet only a small number of operators are applicable for achieving a given goal
- Hence, regression is more goal-directed than progression situation-space planning

- Disadvantage: cannot always find a plan even if one exists!

Algorithm

1. Start with goal node corresponding to goal to be achieved
2. Choose an operator that will add one of the goals
3. Replace that goal with the operator's preconditions
4. Repeat steps #2-#3 until you have reached the initial state
5. While backward-chaining is performed by STRIPS in terms of the generation of goals, sub-goals, sub-sub-goals, etc., operators are used in the forward direction to generate successor states, starting from the initial state, until a goal is found.

Blocks World example

Step	State	Stack	Plan	Note
#1	ontable(A) Λ ontable(B) Λ on(C, B) Λ clear(A) Λ clear(C) Λ handempty	achieve(on(A,C))		Stack contains original goal. State contains the initial state description.
#2	Same.	achieve(clear(C), holding(A), apply(Stack(A,C)) achieve(on(A,C))		Choose operator Stack to solve goal popped from top of goal stack.
#3	Same.	achieve(holding(A)) achieve(clear(C)) achieve(clear(C), holding(A), apply(Stack(A,C)) achieve(on(A,C))		Order sub-goals arbitrarily.

#4	Same.	achieve(ontable(A), clear(A), handempty), apply(pickup(A)) achieve(holding(A)) achieve(clear(C)) achieve(clear(C), holding(A), apply(Stack(A,C)) achieve(on(A,C))		Choose operator *pickup* to solve goal popped from top of goal stack.
#5	ontable(B) Λ on(C, B) Λ clear(C) Λ holding(A)	achieve(clear(C)) achieve(clear(C), holding(A), apply(Stack(A,C)) achieve(on(A,C))	Pickup(A)	Top goal is true in current state, so pop it and apply operator *pickup(A)*.
#6	ontable(B) Λ on(C, B) Λ on(A,C) Λ clear(A) Λ handempty	achieve(on(A,C))	pickup(A) stack(A,C)	Top goal *achieve(C)* true so pop it. Re-verify that goals that are the preconditions of the *stack(A,C)* operator still true, then pop that and the operator is applied.
#7	Same.	<empty>		Re-verify that original goal is true in current state, then pop and halt with empty goal stack and state description satisfying original goal.

Total-Order Planning

- Forward/backward state-space searches are forms of totally ordered plan search
- explore only strictly linear sequences of actions directly connected to the start or goal
- cannot take advantages of problem decomposition

Partial-Order Planning

- Idea:
- works on several subgoals independently
- solves them with subplans
- combines the subplans
- flexibility in ordering the subplans
- least commitment strategy:
  - delaying a choice during search
  - Example, leave actions unordered, unless they must be sequential

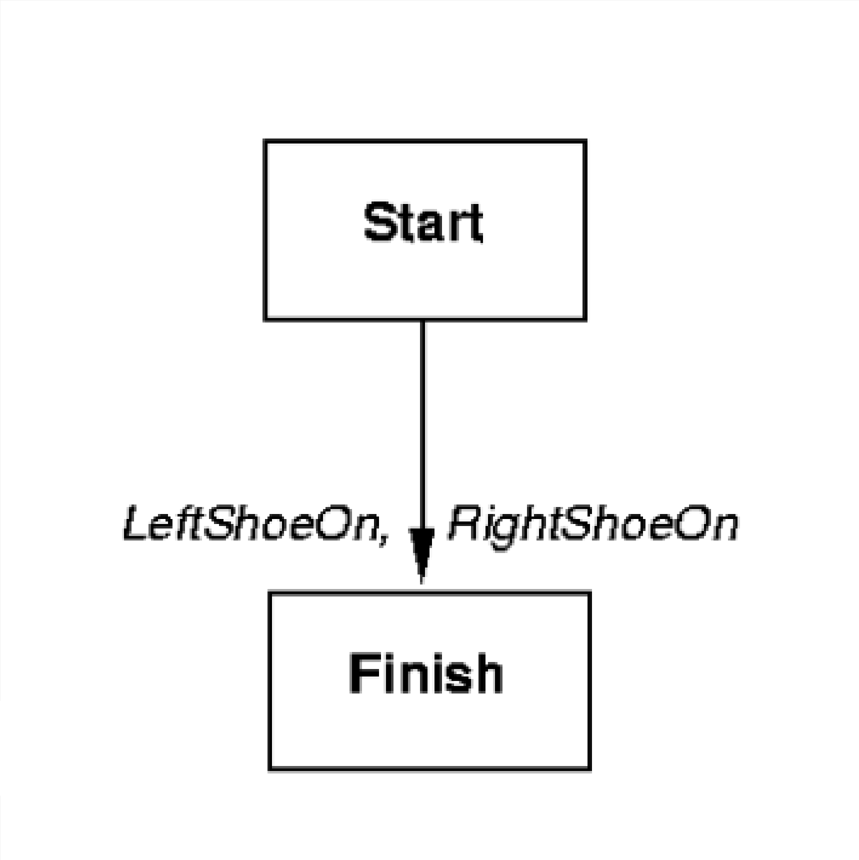
Principle of Least Commitment

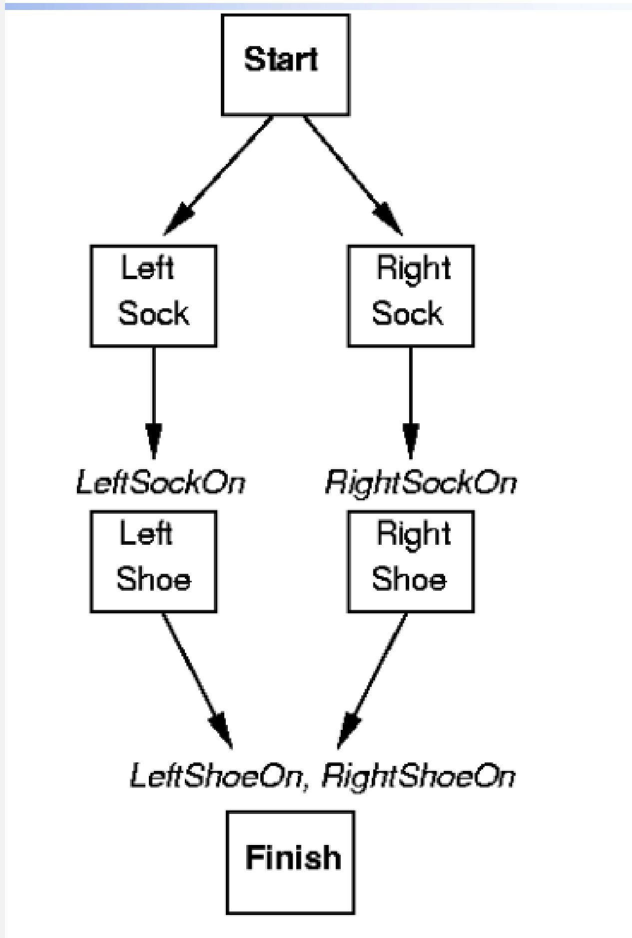
- The principle of least commitment is the idea of never making a choice unless required to do so.
- The advantage of using this principle is you won't have to backtrack later! In planning, one application of this principle is to never order plan steps unless it's necessary for some reason.
- So, partial-order planners exhibit this property because constraint ordering steps will only be inserted when necessary.
- On the other hand, situation-space progression planners make commitments about the order of steps as they try to find a solution and therefore may make mistakes from poor guesses about the right order of steps.

POP Example

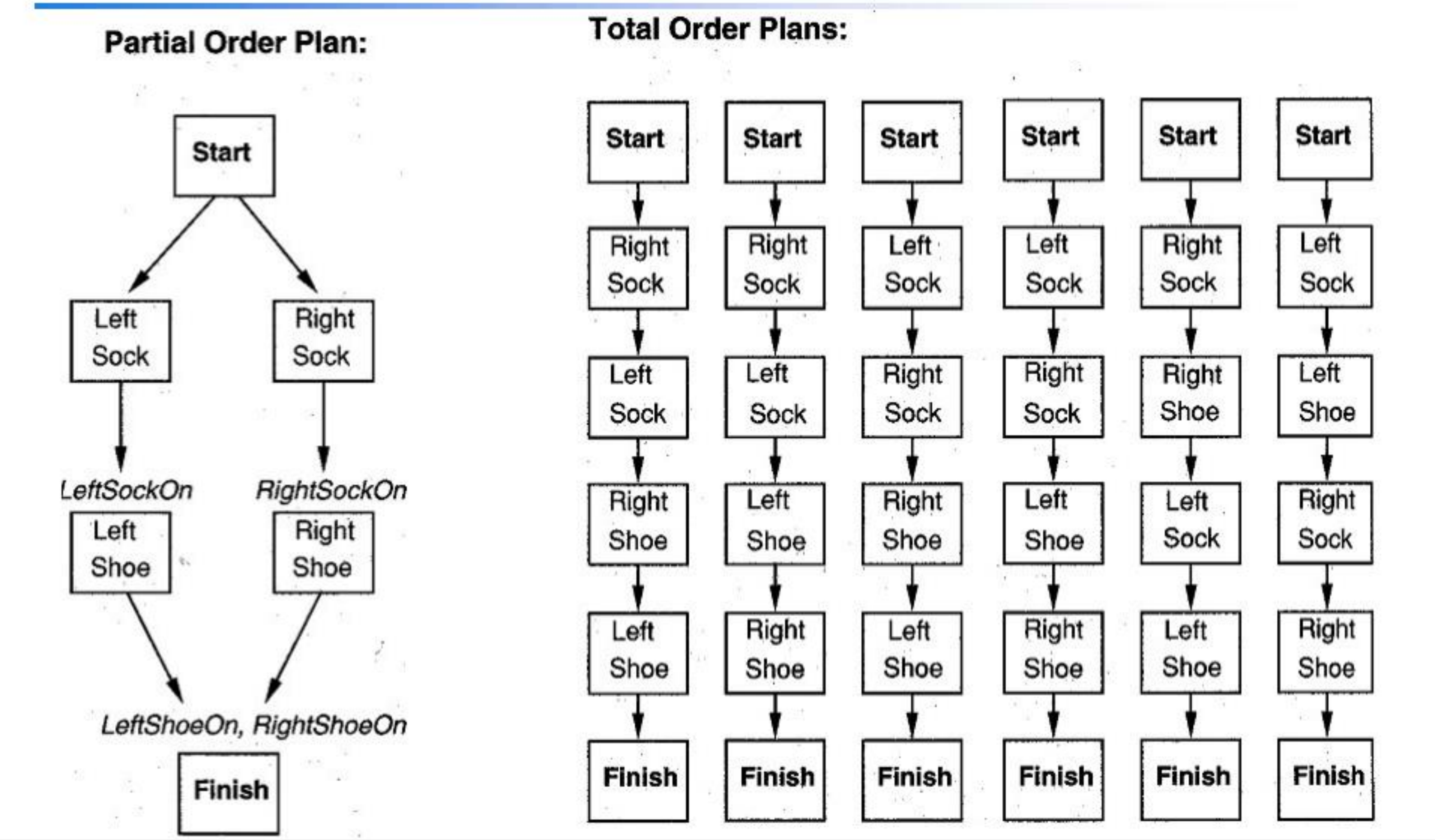
- Putting on a pair of shoes:
- Goal(RightShoeOn ^ LeftShoeOn)
- Init()
- Action: RightShoe
  - PRECOND: RightSockOn
  - EFFECT: RightShoeOn
- Action: RightSock
  - PRECOND: None
  - EFFECT: RightSockOn
- Action:LeftShoe
  - PRECOND: LeftSockOn
  - EFFECT: LeftShoeOn
- Action: LeftSock
  - PRECOND: None
  - EFFECT: LeftSockOn

POP Example


Start
LeftShoeOn, RightShoeOn
Finish


Start
Left Sock
Right Sock
LeftSockOn
RightSockOn
Left Shoe
Right Shoe
LeftShoeOn, RightShoeOn
Finish

Partial Order Plan to Total Order Plan



POP Example: Flat Tire

$Init(At(Flat, Axle) \land At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(Spare, Trunk),$
  PRECOND: $At(Spare, Trunk)$
  EFFECT: $\neg At(Spare, Trunk) \land At(Spare, Ground))$
$Action(Remove(Flat, Axle),$
  PRECOND: $At(Flat, Axle)$
  EFFECT: $\neg At(Flat, Axle) \land At(Flat, Ground))$
$Action(PutOn(Spare, Axle),$
  PRECOND: $At(Spare, Ground) \land \neg At(Flat, Axle)$
  EFFECT: $\neg At(Spare, Ground) \land At(Spare, Axle))$
$Action(LeaveOvernight,$
  PRECOND:
  EFFECT: $\neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk)$
  $\land \neg At(Flat, Ground) \land \neg At(Flat, Axle))$

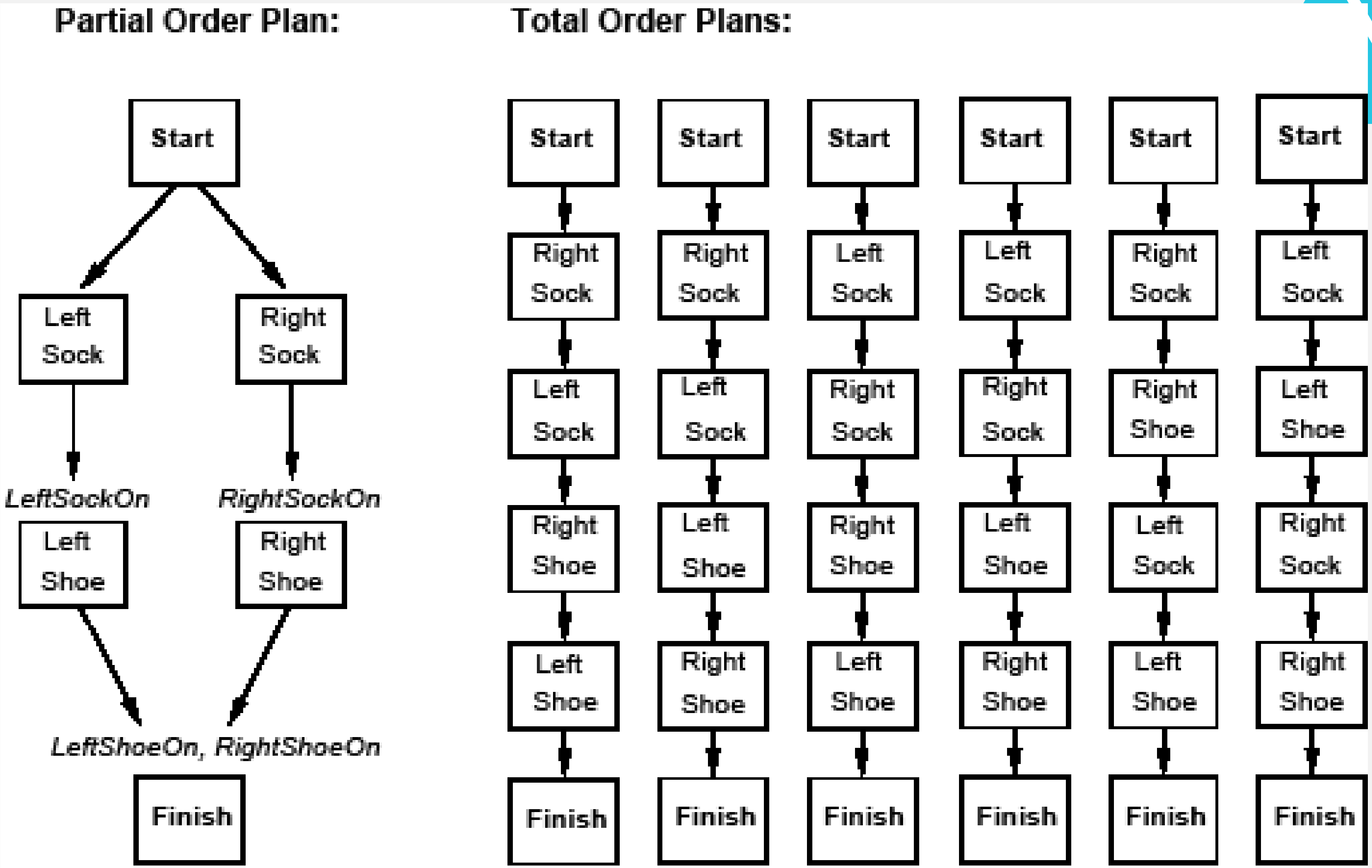
Representation of Plans

- A plan is formally defined as a data structure consisting of the following 4 components:
  1. A set of plan steps
  2. A set of step ordering constraints
  3. A set of variable binding constraints
  4. A set of causal links

Example

- Plan(STEPS:{S1:Op(ACTION: Start),
- S2:Op(ACTION: Finish,
- PRECOND: Ontable(c), On(b,c), On(a,b) },
- ORDERINGS: {S1 < S2},
- BINDINGS: {},
- LINKS: {})

Simple Sock/Shoe Example



Hierarchical planning

(Extra)

Hierarchical Planning

Principle

- hierarchical organization of 'actions'
- complex and less complex actions
- lowest level reflects directly executable actions

Procedure

- planning starts with complex action on top
- plan constructed through action decomposition
- substitute complex action with plan of less complex actions

Hierarchical Planning

Hierarchical Planning / Plan Decomposition

Plans are organized in a hierarchy. Links between nodes at different levels in the hierarchy denote a decomposition of a “complex action”

Example:

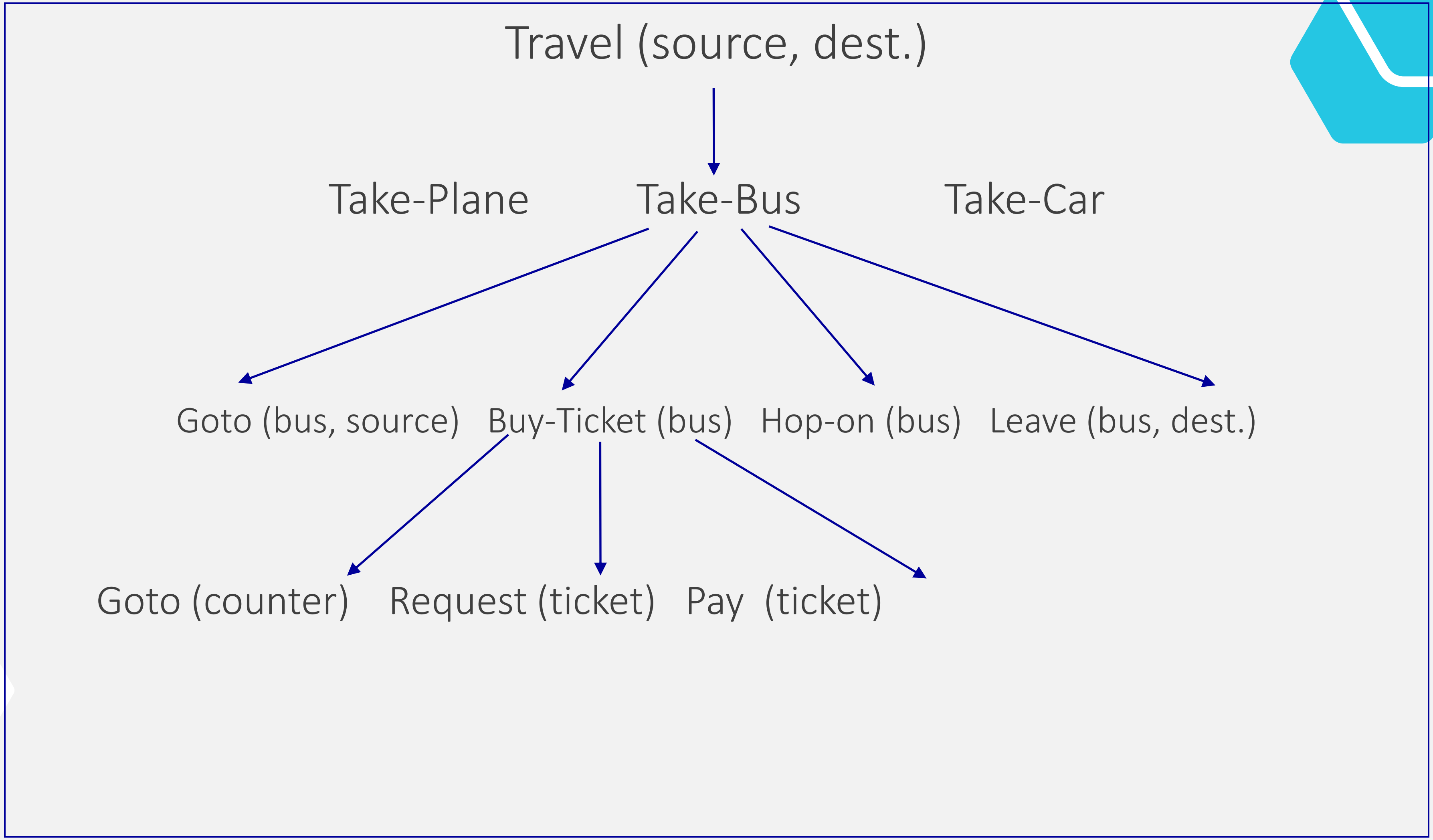
move (x, y, z)

operator expansion

pickup (x, y)   putdown (x, z)

The lowest level corresponds to executable actions of the agent.

Hierarchical Plan - Example



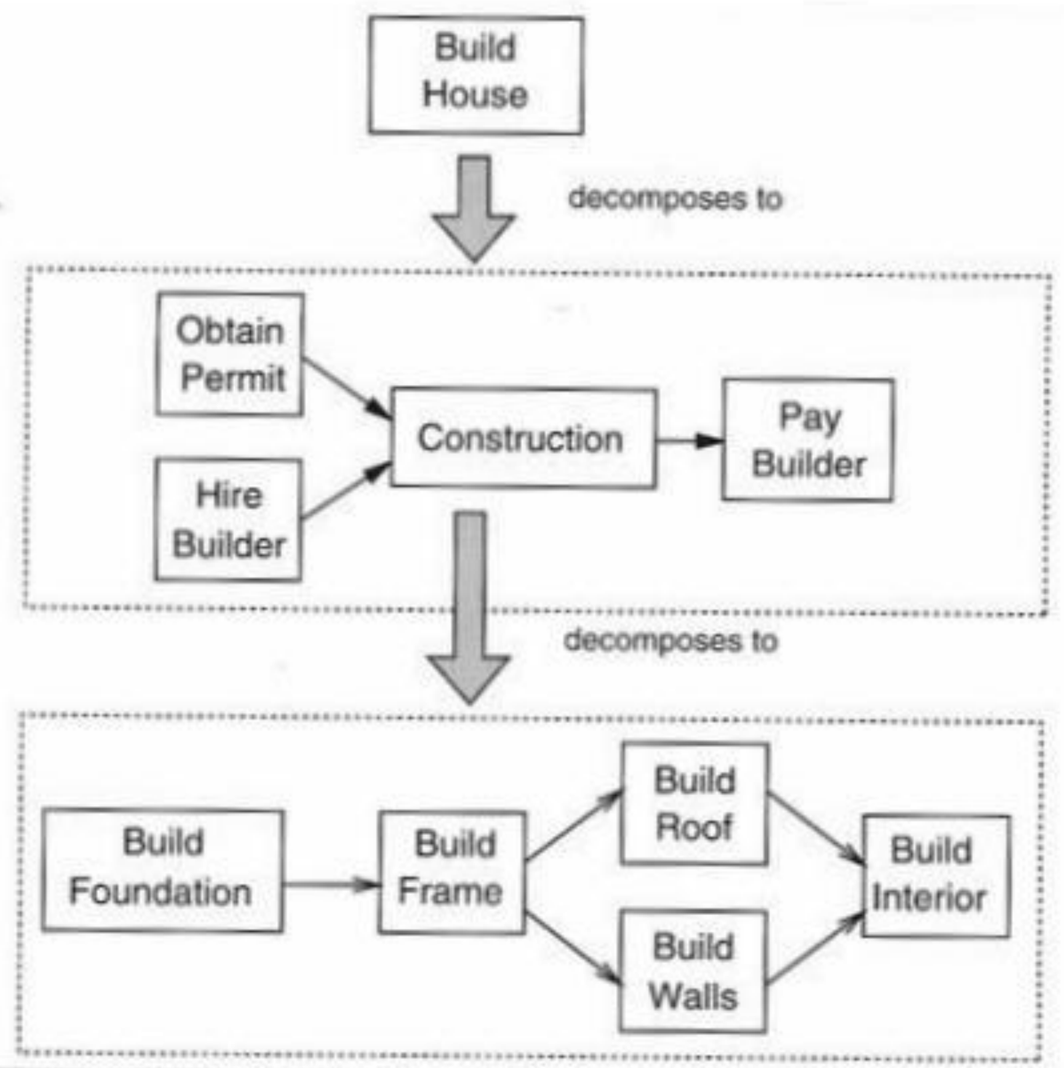


Figure 12.1 Hierarchical decomposition of the operator *Build House* into a partial-order plan.

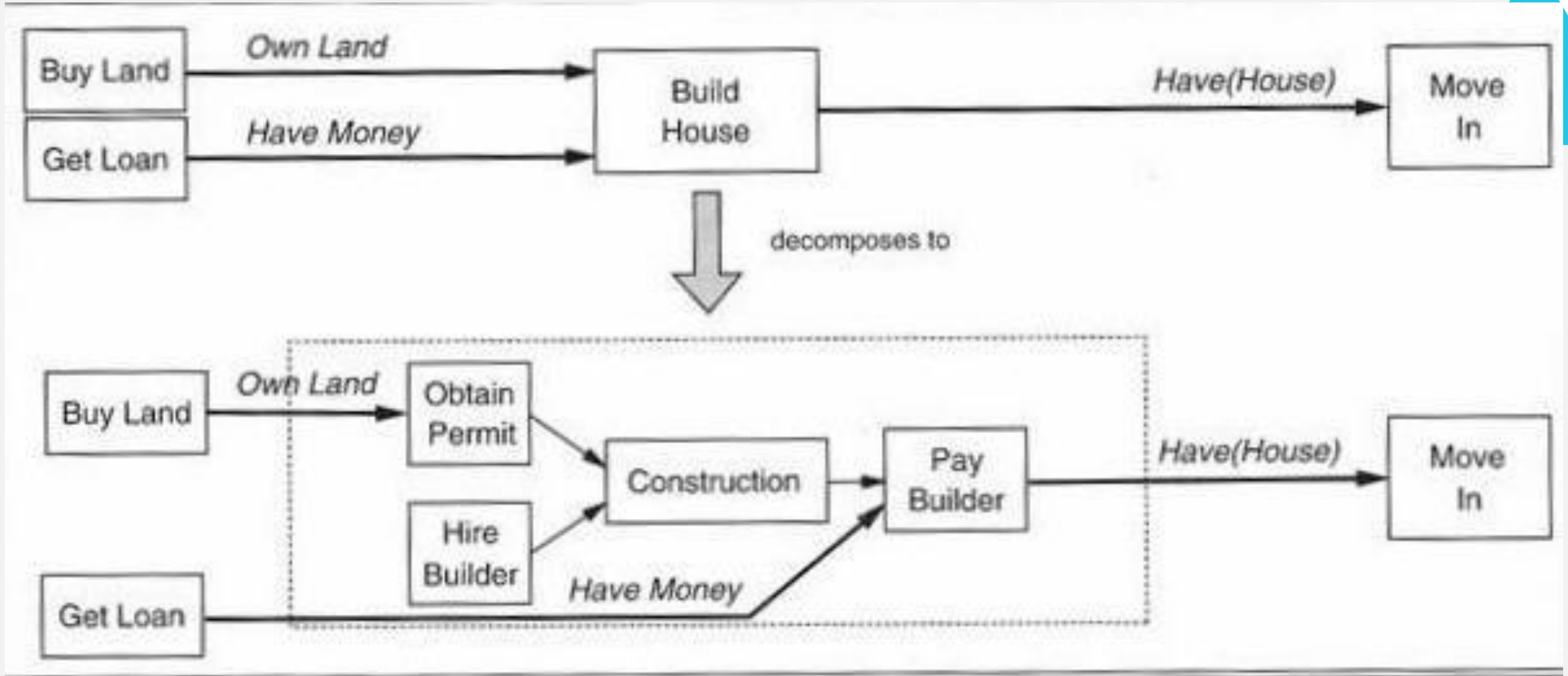


Figure 12.3 Detailed decomposition of a plan step.

Conditional Planning

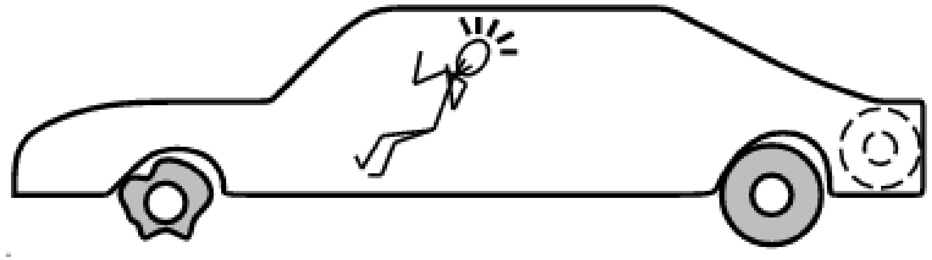
- Also called Contingency planning.
- It deals with uncertainty by checking the environment to see what is really happening.
- Used in fully observable environments.

Execution of conditional plan

$$[\ldots, \mathbf{If}(p, [thenPlan], [elsePlan]), \ldots]$$

Check *p* against current knowledge base, execute *thenPlan* or *elsePlan*

The Real World



On(x) ~Flat(x)

FINISH

START

~Flat(Spare) Intact(Spare) Off(Spare)
On(Tire1) Flat(Tire1)

On(x)

Remove(x)

Off(x) ClearHub

Off(x) ClearHub

Puton(x)

On(x) ~ClearHub

Intact(x) Flat(x)

Inflate(x)

~Flat(x)

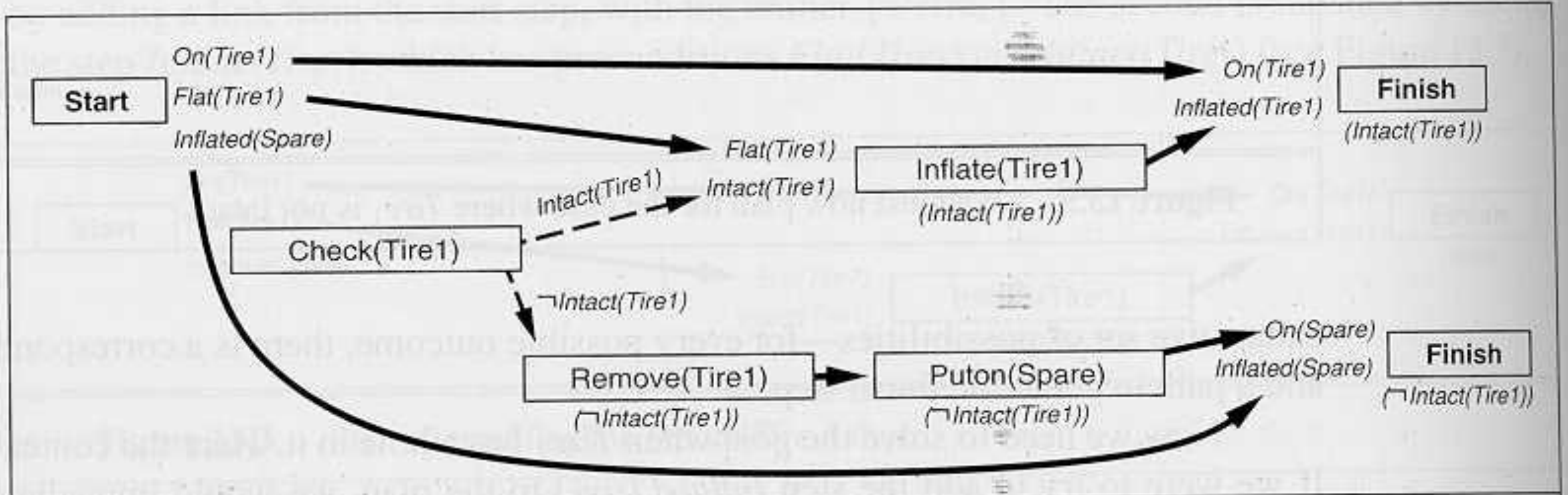


Figure 13.7 The complete plan for fixing the tire, showing causal and conditional links and contexts (in parentheses).

13.1 Conditional Planning

- Ex) "Fixing a flat tyre"

(1) Possible operators

- Op(ACTION:Remove(x),
  PRECOND:On(x),
  EFFECT:Off(x) $\wedge$ ClearHub(x) $\wedge$ $\neg$On(x))
- Op(ACTION:PutOn(y),
  PRECOND:Off(x) $\wedge$ ClearHub(x),
  EFFECT:On(y) $\wedge$ $\neg$ClearHub(y) $\wedge$ $\neg$Off(y))
- Op(ACTION:Inflate(x),
  PRECOND:Intact(x) $\wedge$ Flat(x),
  EFFECT:Inflated(x) $\wedge$ $\neg$Flat(x))

(2) goal

- On(y) $\wedge$ Inflated(y)

(3) Initial conditions

- Inflated(Spare) V Intact(Spare) V Off(Spare) V On($Tire_1$) V Flat($Tire_1$)

Resource constraint and Temporal Constraint

Activities

- An activity A is a task corresponds to a time interval i.e. [start(A), end(A)].

start(A) denotes the start time of the activity.

end(A) denotes the end time of the activity.

- Values of start(A) and end(A) are integer.

- Duration of activity A is a variable and is denoted by : dur(A)=end(A) – start(A)

Activities

- Depending on a problem, the duration may be known in advance or may be a decision variable.

Temporal Constraints

- A temporal constraint is a constraint in the form:

  d(min) <= ti -> tj <= d(max)

ti – actual start time of activity

tj – end time of activity

d(min) – time when the activity was initiated

d(max) – time when the activity was actually completed

Resources

Resources can be:

→Discrete Resource (has a known maximal capacity profile over time).

→Unary Resource (resource with unit capacity)

Resource Constraints

- A resource constraint defines how a given activity A will require and affect the availability of a given resource R.
- **Resource constraints consists of**

  (A,R,q,TE)

A is name of activity

R is a resource

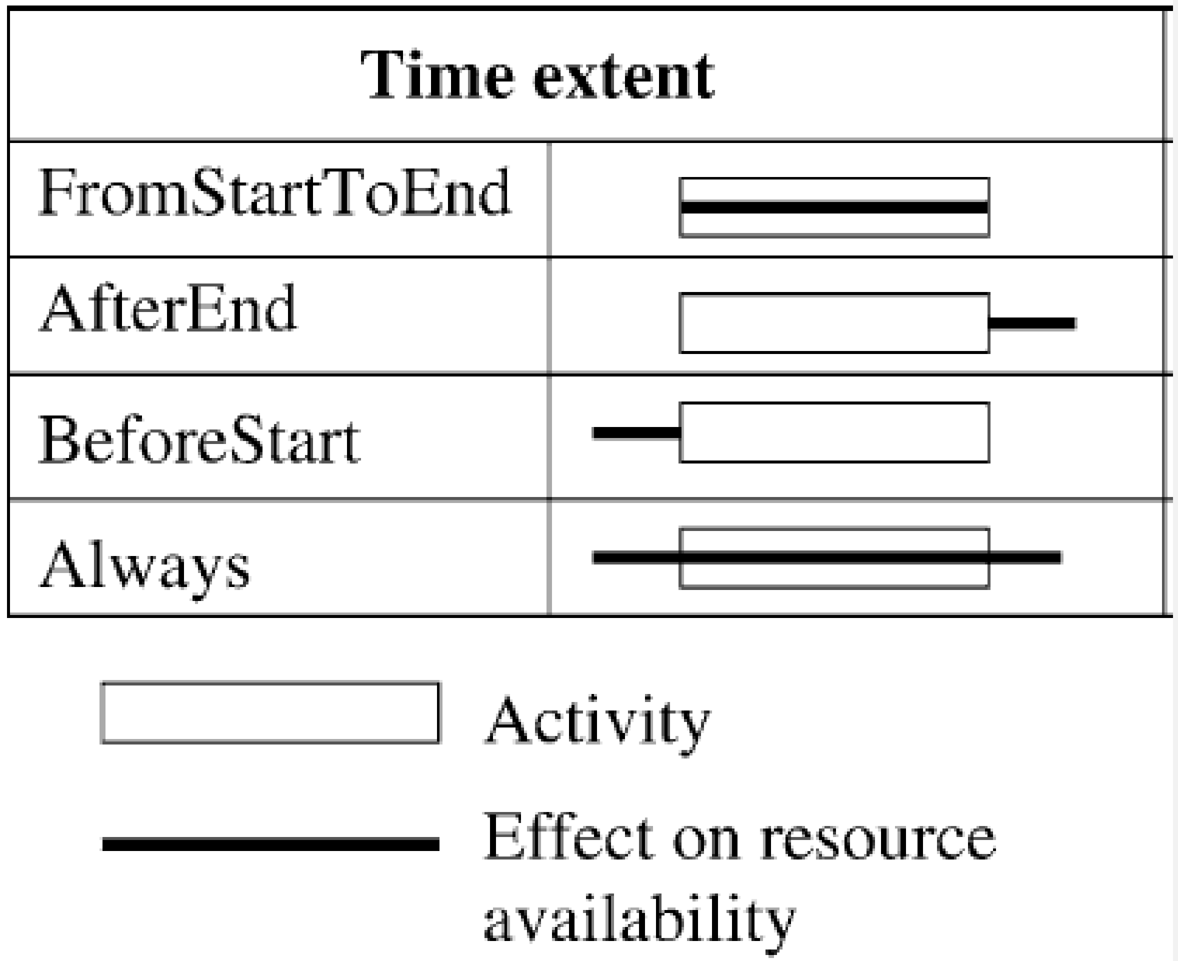
q is the quantity of the resource

TE is the time extent that defines the time interval where the availability of resource R is affected by the execution of activity A.

Resource Constraints

- For Example:

(A,R1,2,fromStartToEnd)


Time extent
FromStartToEnd
AfterEnd
BeforeStart
Always
Activity
Effect on resource availability

LOVELY PROFESSIONAL UNIVERSITY

Thank You

Sukanta Ghosh

+91-8699583576

Sukanta.19539@lpu.co.in