ARTIFICIAL INTELLIGENCE

L T P C

BCSE306L - 3 0 0 3

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Dr. S M SATAPATHY

Associate Professor Sr.,
School of Computer Science and Engineering,
VIT Vellore, TN, India – 632 014.

Module – 6

PLANNING

1. Classical Planning
2. Planning Graphs
3. Hierarchical Planning
4. Planning and acting in Nondeterministic domain

PLANNING

Planning :: Introduction

- In AI, planning refers to the process of creating a sequence of actions (a plan) for an intelligent agent to achieve a specific goal in a particular environment.

- The environment can be the real world, a simulation, or an abstract space.

- Planning algorithms consider factors like:
  - The agent's capabilities (what actions it can perform)
  - The state of the environment (current situation)
  - The desired goal (what the agent wants to achieve)

Planning :: Types

- There are various approaches to planning in AI, each suited for different scenarios and complexities.
  - Classical Planning
  - State-Space Planning
  - Plan-Space Planning
  - Probabilistic Planning
  - Hierarchical Planning
  - Planning with Sensing
  - Real-Time Planning

Classical Planning :: Definition

- [The Ideal World Scenario]: This is a foundational type of planning and is defined as the task of finding a sequence of actions to accomplish a goal in a discrete, deterministic, static, fully observable environment.
- Example: Imagine a robot needs to clean a room (environment). Classical planning would assume:
  - The robot can perfectly sense the dirt locations (observable world).
  - Picking up dirt always removes it (deterministic actions).
  - The robot has unlimited time to clean (no time constraints).
- Classical planning algorithms would then search for a sequence of actions (move to location A, pick up dirt, move to location B, pick up dirt...) that leads the robot from a dirty room (start state) to a clean room (goal state).

State-Space Planning :: Definition

- [Navigating the Possibilities]: This technique represents the environment as a set of discrete states (room layouts) and actions as transitions between those states (moving between rooms). The planning process involves searching through this state space to find a path from the initial state (e.g., dirty room) to the goal state (e.g., clean room).
- State-space planning can handle partial observability by considering possible states the agent might be in, but it can become computationally expensive for large state spaces.
- Example: A self-driving car navigating a city can leverage state-space planning. Each intersection represents a state, and possible actions are turns or straight movements. The planner searches for a path through this state space that leads the car to its destination while avoiding obstacles.

Plan-Space Planning :: Definition

- [Building the Steps Incrementally]: This approach reasons about the effects of actions rather than the states themselves.
- It builds a plan step-by-step, considering the preconditions (requirements for an action to happen) and post-conditions (resulting state after the action) of each action..
- Example: A robotic chef might use plan-space planning to cook a meal. The goal is a prepared dish. The plan might involve subgoals like "chop vegetables" and "cook meat," each with preconditions (having vegetables and meat available) and postconditions (chopped vegetables and cooked meat).

Probabilistic Planning :: Definition

- [Considering the Uncertainties]: This approach tackles the uncertainty present in real-world scenarios. It acknowledges that actions might not always succeed (e.g., grasping an object) or the environment might be partially observable (e.g., hidden objects).
- Probabilistic planning uses techniques like probability distributions to reason about the likelihood of different outcomes.
- Example: A search and rescue robot in a disaster zone might use probabilistic planning. The environment might have debris (partially observable), and grasping objects might have a chance of failure. The planner considers these probabilities to find the most likely path to locate survivors.

Hierarchical Planning :: Definition

- [Breaking Down Complexity]: This approach tackles complex problems by decomposing them into smaller, more manageable sub-problems.
- It creates a hierarchy of plans, where high-level plans define the overall goal (e.g., win a game) and sub-plans define the specific actions for achieving subgoals (e.g., capture enemy pieces in chess).
- Example: A robot competing in a household chores competition might use hierarchical planning. The high-level goal is to clean the house. Sub-plans might involve cleaning the kitchen (with further sub-plans for washing dishes and wiping counters) and vacuuming the living room.

Planning with Sensing :: Definition

- Real-World Integration [Adapting as You Go]: This approach incorporates the need for the agent to gather information about the environment as it acts.
- Sensors provide real-time data, and the plan might need to be adapted based on this information.
- Example: An autonomous vacuum cleaner might use planning with sensing. The initial plan might be to clean a room in a straight line. Sensors might detect obstacles, and the plan would adapt to navigate around them, ensuring a clean room despite unexpected situations.

Real-Time Planning :: Definition

- [Reacting to Change]: This approach enables re-planning as the environment changes or new information becomes available.
- The agent continuously evaluates its progress and adjusts its plan based on the current situation.
- Example: A self-driving car encountering unexpected traffic might use real-time planning. The initial plan might involve following a specific route. If there's a sudden traffic jam, the planner would need to replan the route to reach the destination efficiently.

Terminologies

- Factored representation: Imagine breaking down a world into smaller, independent pieces like furniture in a room. It's a way to represent the state of the world using these independent pieces.
  - Example: "At(Robot, Kitchen)" and "Clean(Floor)" are factored pieces describing the robot's location and the floor's cleanliness.

- Fluent: These are the basic building blocks of the factored representation, like individual pieces of furniture. They describe properties of the world that can change.
  - Example: "At(Robot, Kitchen)" and "Clean(Floor)" are fluents. "Robot" and "Kitchen" are not fluents because they don't change.

Terminologies

- State: This is a snapshot of the world at a specific time, like the arrangement of furniture at a specific moment. It's a combination of all the current fluents.
  - Example: The state might be { "At(Robot, Kitchen)", "Clean(Floor)" } if the robot is in the kitchen and the floor is clean.
- Grounds: These are specific instances of fluents, like a "RedChair" instead of just "Chair." We use them when talking about a particular situation.
  - Example: "Clean(Floor)" is a fluent, but "Clean(KitchenFloor)" would be a ground if we're specifically talking about the kitchen floor.
- Functionless atoms: These are fluents that don't involve any functions. They simply state a fact about the world, like "Clean(Floor)."
  - Example: "Clean(Floor)" is a functionless atom because it just describes the state of the floor.

Terminologies

- Actions: These are things the agent (like a robot) can do to change the world, like moving from room to room or picking up objects.
  - Example: "Move(Robot, Kitchen)" is an action representing the robot moving to the kitchen.
- Action schema: This is a general template for an action, like a blueprint for building different types of furniture. It describes the preconditions (things that need to be true before the action can happen) and the effects (changes the action makes to the world).
  - Example: An action schema for "Move(Robot, Location)" might have the precondition "At(Robot, CurrentLocation)" and the effect "¬At(Robot, CurrentLocation) ∧ At(Robot, Location)" (meaning the robot is no longer in the current location and is now in the new location).

Terminologies

- Lifted representation: This is a way to write action schemas that use variables instead of specific values, like using "Location" instead of "Kitchen" in the action schema. It allows the schema to be applied to different situations.
  - Example: The action schema "Move(Robot, Location)" is a lifted representation because it uses the variable "Location" which can be any valid location.

Planning

- Planning systems are problem-solving algorithms that operate on explicit propositional or relational representations of states and actions
- Planning problem: find a plan that is guaranteed (from any of the initial states) to generate a sequence of actions that leads to one of the goal states
- Planning problems often have large state spaces.
- Automated Planning:
  - Two popular and effective current approaches to automated classical planning:
    - Forward state-space search with heuristics
    - Translating to a Boolean satisfiability problem
  - Other approaches, e.g. planning graphs: data structures to give better heuristic estimates than other methods, and also used to search for a solution over the space formed by the planning graph

Planning Problem

- Agent Environment States are represented as value of state variables.
- An action can be represented as a procedure or a program
- The procedures are used to compute values of state variables
- After the execution of procedure (i.e. after the action), the environment state will be changed, towards the goal.
- A planning problem is defined by a quadruple <S, A, G, initialState>, where:
  - S is a set of states representing the possible configurations of the environment.
  - A is a set of actions that the agent can take.
  - G is a goal state, a specific desirable configuration in S.
  - initialState is the current state of the environment from which the planning process begins.

Representation of a Planning Problem

- Search based problem-solving agents
  - Find sequences of actions that result in a goal state BUT deal with atomic states so need good domain specific heuristics to perform well
- Planning represented by factored representation
  - Represent a state by a collection of variables
- Planning Domain Definition Language (PDDL)
  - Allows expression of all actions with one schema
  - Inspired by earlier STRIPS planning language
- Defining a Search Problem : Define a search problem through:
  1) Initial state
  2) Actions available in a state
  3) Result of action
  4) Goal test

PDDL

- Planning Domain Definition Language (PDDL) is a factored representation.
- Basic PDDL can handle classical planning domains, and extensions can handle non-classical domains that are continuous, partially observable, concurrent, and multi-agent.
- State: represented as a conjunction of ground atomic fluents
- These are ground, functionless atoms
  - Example: At(Truck1,Manchester) ∧ At(Truck2,Warrington)
- Unique names assumption (Truck1 distinct from Truck2)
- uses database semantics: the closed-world assumption means that any fluents that are not mentioned are false.
- Actions described by a set of action schemas that implicitly define Actions(s) and Result(s,a) functions

PDDL :: Action Schema

- An action schema represents a family of ground actions
  - The schema consists of the action name, a list of all the variables used in the schema, a precondition and an effect.
  - Example: action schema for flying a plane from one location to another

    Action(Fly(p, from, to),

    PRECOND: At(p, from) ∧ Plane(p) ∧Airport(from) ∧ Airport(to)

    EFFECT: ¬At(p, from) ∧ At(p, to))
- Free to choose whatever values we want to instantiate variables
- Precondition and effect of an action are each conjunctions of literals (positive or negated atomic sentences)
  - Precondition defines states in which action can be executed
  - Effect defines result of action

PDDL :: Action Schema

- Action a can be executed in state s if s entails the precondition of a

$$(a \in \text{Actions}(s)) \Leftrightarrow s \models \text{Precond}(a)$$

where any variables in a are universally quantified

- Example:

$$\forall p,\text{from},\text{to}\ (\text{Fly}(p,\text{from},\text{to}) \in \text{Actions}(s)) \Leftrightarrow$$
$$s \models (\text{At}(p,\text{from}) \wedge \text{Plane}(p) \wedge \text{Airport}(\text{from}) \wedge \text{Airport}(\text{to}))$$

- We say that a is applicable in s if the preconditions are satisfied by s
- Result of executing action *a in state s (s')*

$$\text{Result}(s,a)=(s-\text{Del}(a)) \cup \text{Add}(a)$$

- Delete list (Del(a)): fluents that appear as negative literals in action's effect
- Add list (Add(a)): fluents that appear as positive literals in action's effect

PDDL :: Planning Domain

- A set of action schemas serves as a definition of a planning domain. A specific problem within the domain is defined with the addition of an initial state and a goal
- The initial state is a conjunction of ground fluents
- The goal is just like a precondition: a conjunction of literals (positive or negative) that may contain variables.
  - e.g. At(p,LPL) ∧ Plane(p)
- Problem solved when we find sequence of actions that end in a state that entails the goal
  - e.g. Plane(Plane1) ∧ At(Plane1,LPL) entails the goal At(p,LPL) ∧ Plane(p)

PDDL :: Air Cargo Transport Problem

- Consider an air cargo transport problem involving loading and unloading cargo and flying it from place to place.
- The problem can be defined with three actions: Load, Unload, and Fly.
- The actions affect two predicates:
  - In(c,p) means that cargo c is inside plane p,
  - At(x, a) means that object x (either plane or cargo) is at airport a.
- Note that some care must be taken to make sure the At predicates are maintained properly.

PDDL :: Planning Domain

Example: Air Cargo Transport

Init(At(C_1,SFO) ∧ At(C_2,JFK) ∧ At(P_1,SFO) ∧ At(P_2,JFK) ∧
Cargo(C_1) ∧ Cargo(C_2) ∧ Plane(P_1) ∧ Plane(P_2) ∧
Airport(JFK) ∧ Airport(SFO))

Goal(At(C_1,JFK) ∧ At(C_2,SFO))

Action(Load(c,p,a),
PRECOND*: At(c,a) ∧ At(p,a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)*
EFFECT: ¬*At(c,a) ∧ In(c,p))*

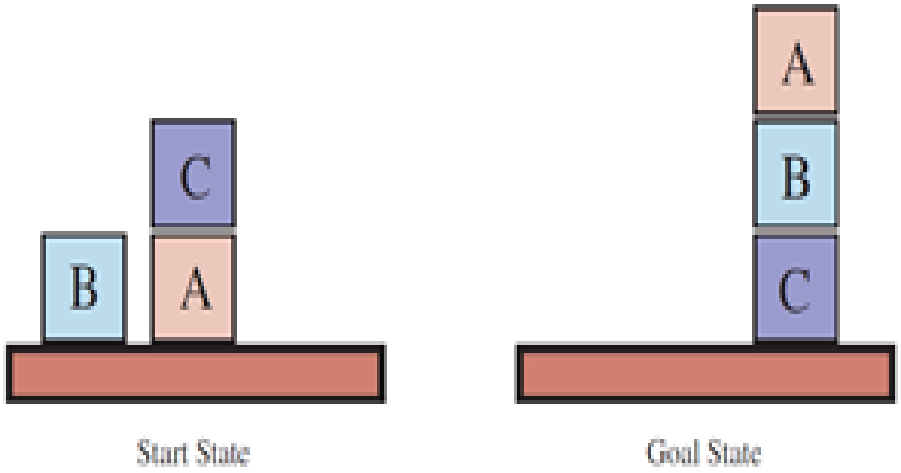
Action(Unload(c,p,a),
PRECOND*: In(c,p) ∧ At(p,a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)*
EFFECT: *At(c,a) ∧* ¬*In(c,p))*

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

PDDL :: Air Cargo Transport Problem

- Problem defined with 3 actions and Actions affect 2 predicates
- When a plane flies from one airport to another, all cargo inside goes too
  - in PDDL we have no explicit universal quantifier to say this as part of the Fly action
  - so instead we use the load/unload actions:
    - cargo seizes to be At the old airport when it is loaded
    - and only becomes At the new airport when it is unloaded
- A solution plan:
  - *[Load(C1,P1,SFO),Fly(P1,SFO,JFK),Unload(C1,P1,JFK),*
  - *Load(C2.P2.JFK),Fly(P2,JFK,SFO),Unload(C2,P2,SFO)].*
- Problem – spurious actions like *Fly(P1,JFK,JFK) have contradictory effects [*Add inequality preconditions ∧ *(from ≠ to)]*

PDDL :: The Blocks World Problem



- One of the famous planning domains is known as the blocks world. This domain consists of a set of cube-shaped blocks sitting on a table.
- The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it to another position, either on the table or on top of another block.
- The arm can pick up only one block at a time, so it cannot pick up a block that has another one on it.

PDDL :: The Blocks World Problem

$$Init(On(A, Table) \land On(B, Table) \land On(C, A)$$
$$\land Block(A) \land Block(B) \land Block(C) \land Clear(B) \land Clear(C))$$
$$Goal(On(A, B) \land On(B, C))$$
$$Action(Move(b, x, y),$$
$$\text{PRECOND: } On(b, x) \land Clear(b) \land Clear(y) \land Block(b) \land Block(y) \land$$
$$(b \neq x) \land (b \neq y) \land (x \neq y),$$
$$\text{EFFECT: } On(b, y) \land Clear(x) \land \neg On(b, x) \land \neg Clear(y))$$
$$Action(MoveToTable(b, x),$$
$$\text{PRECOND: } On(b, x) \land Clear(b) \land Block(b) \land (b \neq x),$$
$$\text{EFFECT: } On(b, Table) \land Clear(x) \land \neg On(b, x))$$

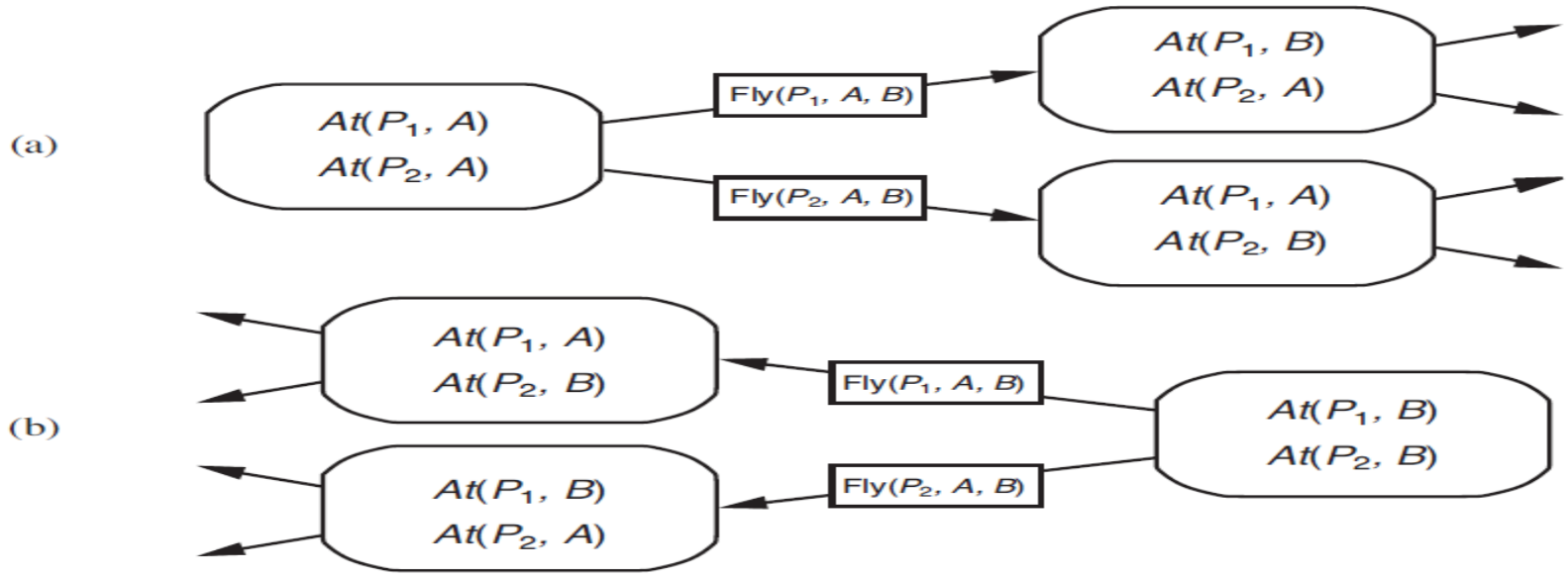
PDDL :: The Blocks World Problem

- A planning problems in the blocks world: building a three-block tower.
- One solution is the sequence

  [MoveToTable(C,A), Move(B,Table,C), Move(A,Table,B)].

Planning with State Space Search

- The most straightforward approach of planning algorithm, is state-space search
  - Forward state-space search (Progression)
  - Backward state-space search (Regression)
- The descriptions of actions in a planning problem, and specify both preconditions and effects.



Forward State Space Search (Progression)

Problem Formulation:

- Initial state:
  - Initial state of the planning problem
- Actions:
  - Applicable to the current state.
  - First actions' preconditions are satisfied, Successor states are generated
  - Add positive literals to add list and negative literals to delete list.
- Goal test:
  - Whether the state satisfies the goal of the planning
- Step cost:
  - Each action is 1 (assumed)

Forward State Space Search (Progression)

Algorithm:

Forward-search (O, s0, g)

1. s ← s0
2. π ← The empty plan
3. loop
   i. if s satisfies g then return π
   ii. applicable ← {a | a is a ground instance of an operator in O, and precond(a) is true in s}
   iii. if applicable = Ø then return failure
   iv. Non-deterministically choose an action a € applicable
   v. s ← y(s, a)
   vi. π ← π.a

Forward State Space Search (Progression)

- Forward search is prone to exploring irrelevant actions.

- Consider the noble task of buying a copy of AI: A Modern Approach from an online bookseller. Suppose there is an action schema Buy(isbn) with effect Own(isbn).
- ISBNs are 10 digits, so this action schema represents 10 billion ground actions.

- An uninformed forward-search algorithm would have to start enumerating these 10 billion actions to find one that leads to the goal.

Forward State Space Search (Progression)

- Planning problems often have large state spaces.
- Consider an air cargo problem with 10 airports, where each airport has 5 planes and 20 pieces of cargo. The goal is to move all the cargo at airport A to airport B.
- There is a simple solution to the problem: load the 20 pieces of cargo into one of the planes at A, fly the plane to B, and unload the cargo.

Forward State Space Search (Progression)

- Finding the solution can be difficult because the average branching factor is huge: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded) or loaded into any plane at its airport (if it is unloaded).
- So in any state there is a minimum of 450 actions (when all the packages are at airports with no planes) and a maximum of 10,450 (when all packages and planes are at the same airport).
- On average, let's say there are about 2000 possible actions per state, so the search graph up to the depth of the obvious solution has about 2000^{41} nodes.

Backward State Space Search (Regression)

- Backward search works only when we know how to regress from a state description to the predecessor state description.
- For example, it is hard to search backwards for a solution to the n-queens problem because there is no easy way to describe the states that are one move away from the goal.
- Happily, the PDDL representation was designed to make it easy to regress actions—if a domain can be expressed in PDDL, then we can do regression search on it.
- Given a ground goal description g and a ground action a, the regression from g over a gives us a state description g' defined by.

$$g' = (g - ADD(a)) \cup Precond(a)$$

Backward State Space Search (Regression)

Algorithm:

Backward-search (O, s0, g)

1. π ← The empty plan
2. loop
   i. if s0 satisfies g then return π
   ii. applicable ← {a | a is a ground instance of an operator in O that is relevant for g}
   iii. if applicable = Ø then return failure
   iv. Non-deterministically choose an action a € applicable
   v. π ← a.π
   vi. g ← y-1(g, a)

Backward State Space Search (Regression)

- For example, suppose the goal is to deliver a specific piece of cargo to SFO: At(C2, SFO).
- That suggests the action Unload(C2, p', SFO):

$$Action(Unload(C_2, p', SFO),$$
$$\text{PRECOND: } In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO)$$
$$\text{EFFECT: } At(C_2, SFO) \wedge \neg In(C_2, p') .$$

Heuristics for Planning

- An admissible heuristic can be derived by defining a relaxed problem that is easier to solve.
- A technique for admissible heuristics: problem relaxation

  ⇒ h(s): the exact cost of a solution to the relaxed problem
- Forms of problem relaxation exploiting problem structure
  - Add arcs to the search graph ⇒ make it easier to search
    - ignore-preconditions heuristics
    - ignore-delete-lists heuristics
  - Clustering nodes (aka state abstraction) ⇒ reduce search space
    - ignore less-relevant fluents

Heuristics for Planning

Ignore-Preconditions Heuristics

- Ignore all preconditions drops all preconditions from actions
  - every action is applicable in any state
  - any single goal literal can be satisfied in one step (or there is no solution)
  - fast, but over-optimistic
- Remove all preconditions & effects, except literals in the goal
  - more accurate
  - NP-complete, but greedy algorithms efficient
- Ignore some selected (less relevant) preconditions
  - relevance based on heuristics or domain-depended criteria

Heuristics for Planning

Ignore-Preconditions Heuristics

Action(Slide(t, s1, s2),

PRECOND: *On(t, s1) ∧ Tile(t) ∧ Blank(s2) ∧ Adjacent(s1, s2)*

EFFECT: *On(t, s2) ∧ Blank(s1) ∧ ¬On(t, s1) ∧ ¬Blank(s2))*

- Remove the precondition Blank(s2)

  ⇒ we get the number-of-misplaced-tiles heuristics

- Remove the preconditions Blank(s2) ∧ Adjacent(s1, s2)

  ⇒ we get the Manhattan-distance heuristics

Heuristics for Planning

Ignore Delete-list Heuristics

- Assumption: goals & preconditions contain only positive literals
  - reasonable in many domains
- Idea: Remove the delete lists from all actions
  - No action will ever undo the effect of actions,

  ⇒ there is a monotonic progress towards the goal
- Still NP-hard to find the optimal solution of the relaxed problem
  - can be approximated in polynomial time, with hill-climbing
- Can be very effective for some problems

Heuristics for Planning

Ignore Delete-list Heuristics

- Consider a scenario where you need to paint a room red and blue. You have red and blue paint cans, but you can only hold one at a time.
- Action Schema:
  - Precondition: Holding a paint can (Holding(PaintCan))
  - Effect: Paint the wall the color of the held can (Painted(Wall, Color)) and empty the can (Empty(PaintCan))
- Ignore-Delete-List Heuristic:
  - We ignore the fact that painting one wall empties the paint can (Ignores the need to refill the paint).
  - This allows us to consider painting both walls with the same can (even though it's not possible).

Heuristics for Planning

Ignoring Less-Relevant Fluents

- This approach focuses on a subset of fluents those are most critical for achieving the goal (ignores less relevant fluents).
- Consider a robot that needs to navigate a maze to reach a specific location (goal).
- Fluents:
  - RobotAt(Location): Location of the robot.
  - Wall(Location1, Location2): Existence of a wall between two locations.
  - Color(Location): Color of the wall at a specific location (might be irrelevant).
  - BatteryLevel(Robot): Battery level of the robot (might be irrelevant for basic navigation).

Heuristics for Planning

Ignoring Less-Relevant Fluents

- This approaches simplify the planning problem, leading to faster computation of heuristic values.
- Reduced Accuracy: Ignoring negative effects or irrelevant fluents can lead to inaccurate estimates of the effort or steps needed to reach the goal.

Planning Graph

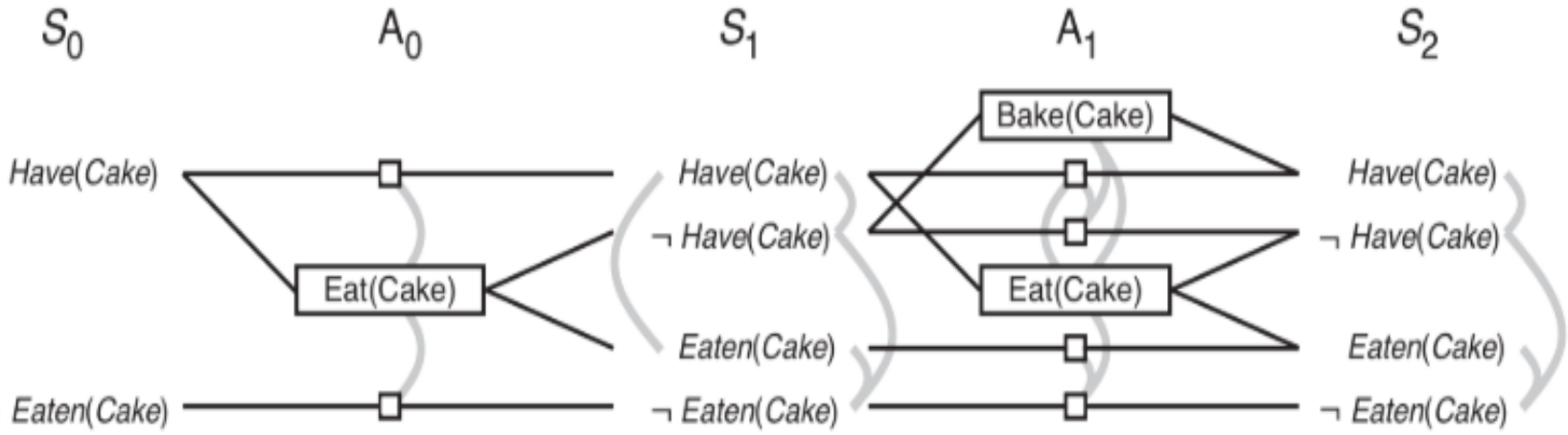
- A directed graph, built forward and organized into levels
  - level S_0: contain each ground fluent that holds in the initial state
  - level A_0: contains each ground action applicable in S_0
  - ...
  - level A_i: contains all ground actions with preconditions in S_{i-1}
  - level S_{i+1}: all the effects of all the actions in A_i
    - each S_i may contain both P_j and $\neg P_j$
- Persistence actions (aka maintenance actions, no-ops)
  - say that a literal l persists if no action negates it
- Mutual exclusion links (mutex) connect
  - incompatible pairs of actions
  - incompatible pairs of literals

Planning Graph

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg\, Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg\, Have(Cake)$
EFFECT: $Have(Cake))$

You would like to eat your cake and still have a cake. Fortunately, you can bake a new one.

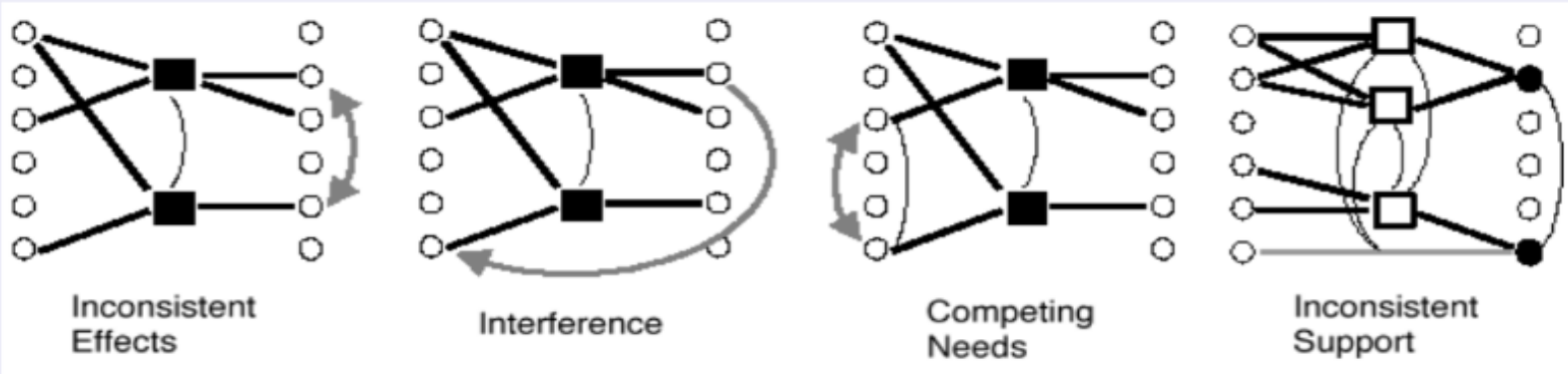
Rectangles indicate actions
Small squares persistence actions (**no-ops**)
Straight lines indicate preconditions and effects
Mutex links are shown as curved gray lines



Planning Graph :: Mutex Computation

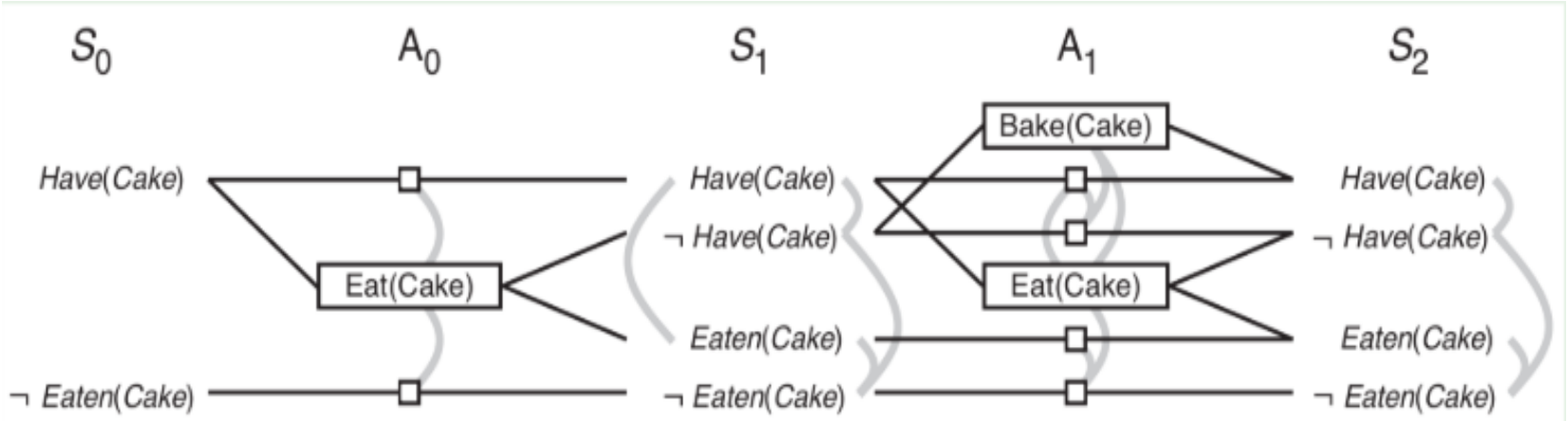
- Two actions at the same action-level have a mutex relation if
  - Inconsistent effects: an effect of one negates an effect of the other
  - Interference: one deletes a precondition of the other
  - Competing needs: they have mutually exclusive preconditions
- Otherwise they don't interfere with each other

  ⇒ both may appear in a solution plan
- Two literals at the same state-level have a mutex relation if
  - inconsistent support: one is the negation of the other
  - all ways of achieving them are pairwise mutex



Planning Graph :: Mutex Computation

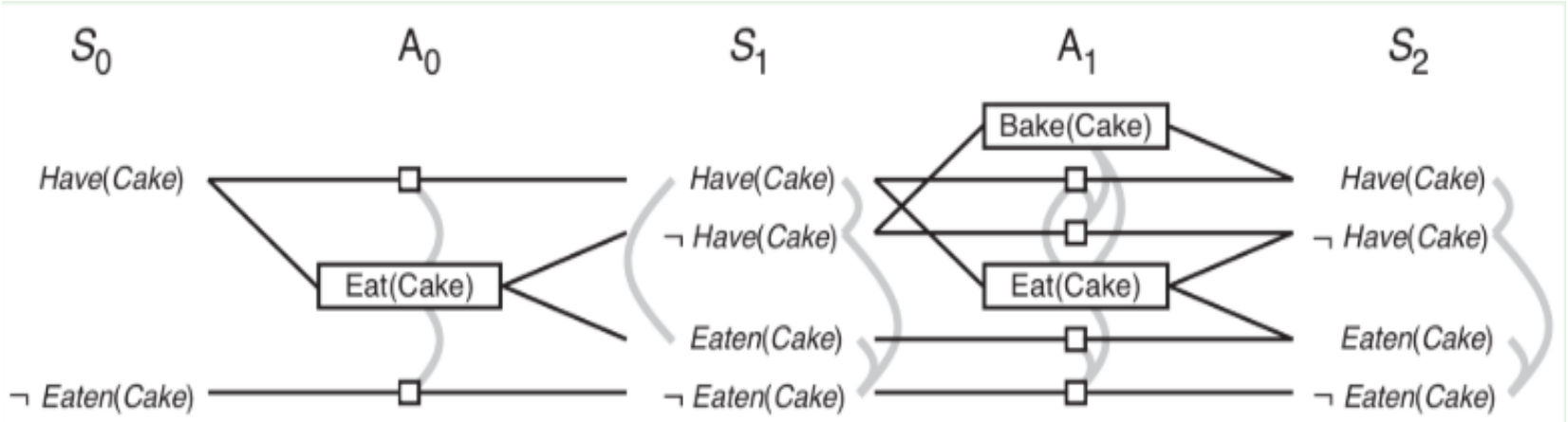
- Inconsistent effects: an effect of one negates an effect of the other
  - ex: persistence of Have(Cake), Eat(Cake) have competing effects
  - ex: Bake(Cake), Eat(Cake) have competing effects
- Interference: one deletes a precondition of the other
  - ex: Eat(Cake) interferes with the persistence of Have(Cake)
- Competing needs: they have mutually exclusive preconditions
  - ex: Bake(Cake) and Eat(Cake)



(© S. Russell & P. Norwig, AIMA)

Planning Graph :: Mutex Computation

- inconsistent support: one is the negation of the other
  - ex.: Have(Cake), ¬Have(Cake)
- all ways of achieving them are pairwise mutex
  - ex.: (S1): Have(Cake) in mutex with Eaten(Cake) because
  - persist. of Have(Cake), Eat(Cake) are mutex



(© S. Russell & P. Norwig, AIMA)

Planning Graph :: Complexity

- A planning graph is polynomial in the size of the problem:
  - a graph with ***n*** levels, ***a*** actions, ***l*** literals, has size $O(n(a + l)^2)$
  - time complexity is also $O(n(a + l)^2)$

⇒ The process of constructing the planning graph is very fast

  - does not require choosing among actions

GraphPlan Algorithm

```
function GRAPHPLAN(problem) returns solution or failure

    graph ← INITIAL-PLANNING-GRAPH(problem)
    goals ← CONJUNCTS(problem.GOAL)
    nogoods ← an empty hash table
    for t = 0 to ∞ do
        if goals all non-mutex in S_t of graph then
            solution ← EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
            if solution ≠ failure then return solution
        if graph and nogoods have both leveled off then return failure
        graph ← EXPAND-GRAPH(graph, problem)
```

Hierarchical Planning

- Real-World planning problems often too complex to handle
- Hierarchical Planners manage the creation of complex plans at different levels of abstraction, by considering the simplest details only after finding a solution for the most difficult ones.
- Hierarchical plan: hierarchy of action sequences (or partial orders) at distinct abstraction levels
  - each action, in turn, can be decomposed further, until we reach the level of actions that can be directly executed
  - designed by hierarchical decomposition (like, e.g., SW design)

Hierarchical Planning

- Ex (vacation plan): “Go to San Francisco airport; take Hawaiian Airlines flight 11 to Honolulu; do vacation for two weeks; take Hawaiian Airlines flight 12 back to San Francisco; go home.”
- “Go to San Francisco airport” can be viewed as a planning task

  ⇒ “Drive to the long-term parking lot; park; take the shuttle to the terminal.” or (simplier): “take a taxi to San Francisco airport”

  - “Drive to the long-term parking lot”: plan a route

Hierarchical Planning

Hierarchical Task Networks (HTN)

- We assume full observability, determinism and the availability of a set of actions (primitive actions, PAs)
- High-level action (HLA):
  - has one or more possible refinements
  - each refinement is a sequence (or p.o.) of actions (PAs or HLAs)
  - may be recursive
- A high-level plan achieves the goal from a given state if at least one of its implementations achieves the goal from that state.

Hierarchical Planning

$Refinement(Go(Home, SFO),$
STEPS: $[Drive(Home, SFOLongTermParking),$
$Shuttle(SFOLongTermParking, SFO)]$)
$Refinement(Go(Home, SFO),$
STEPS: $[Taxi(Home, SFO)]$)

$Refinement(Navigate([a, b], [x, y]),$
PRECOND: $a = x \ \wedge \ b = y$
STEPS: [])
$Refinement(Navigate([a, b], [x, y]),$
PRECOND: $Connected([a, b], [a - 1, b])$
STEPS: $[Left, Navigate([a - 1, b], [x, y])]$)
$Refinement(Navigate([a, b], [x, y]),$
PRECOND: $Connected([a, b], [a + 1, b])$
STEPS: $[Right, Navigate([a + 1, b], [x, y])]$)
...

Hierarchical Planning

```
function HIERARCHICAL-SEARCH(problem, hierarchy) returns a solution, or failure
  frontier ← a FIFO queue with [Act] as the only element
  loop do
    if EMPTY?(frontier) then return failure
    plan ← POP(frontier)  /* chooses the shallowest plan in frontier */
    hla ← the first HLA in plan, or null if none
    prefix,suffix ← the action subsequences before and after hla in plan
    outcome ← RESULT(problem.INITIAL-STATE, prefix)
    if hla is null then  /* so plan is primitive and outcome is its result */
      if outcome satisfies problem.GOAL then return plan
    else for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
      frontier ← INSERT(APPEND(prefix, sequence, suffix), frontier)
```

REFINEMENTS(HLA,OUTCOME,HIERARCHY)

- returns a set of action sequences, one for each refinement of the HLA, whose preconditions are satisfied by the specified state: outcome.

Planning and Acting in Non-Deterministic Domains

- Assumptions so far:
  - the environment is deterministic
  - the environment is fully observable
  - the environment is static
  - the agent knows the effects of each action

⇒ The agent does not need perception:

- can calculate which state results from any sequence of actions
- always knows which state it is in

Planning and Acting in Non-Deterministic Domains

- In the real world, the environment may be uncertain
  - partially observable and/or nondeterministic environment
  - incorrect information (differences between world and model)

⇒ If one of the above assumptions does not hold, use percepts

  - the agent's future actions will depend on future percepts
  - the future percepts cannot be determined in advance

- Use percepts:
  - perceive the changes in the world
  - act accordingly
  - adapt plan when necessary

Handling Indeterminacy

Sensorless planning (aka conformant planning):

- find plan that achieves goal in all possible circumstances

  (regardless of initial state and action effects)

  - for environments with no observations

Conditional planning (aka contingency planning):

- construct conditional plan with different branches for possible contingencies

  - for partially-observable and nondeterministic environments

Execution monitoring and replanning:

- while constructing plan, judge whether plan requires revision

  - for partially-known or evolving environments

Open-World vs. Closed-World Assumption

- Classical Planning based on Closed-World Assumption (CWA)
  - states contain only positive fluents
  - we assume that every fluent not mentioned in a state is false
- Sensorless & Partially-observable Planning based on Open-World Assumption (OWA)
  - states contain both positive and negative fluents
  - if a fluent does not appear in the state, its value is unknown
- A belief state is represented by a logical formula

  (instead of an explicitly-enumerated set of states)

  ⇒ The belief state corresponds exactly to the set of possible worlds that satisfy the formula representing it
- The unknown information can be retrieved via sensing actions (aka percept actions) added to the plan

Case Study

- The table & chair painting problem
  - Given a chair and a table, the goal is to have them of the same color.
  - In the initial state we have two cans of paint, but the colors of the paint and the furniture are unknown.
  - Only the table is initially in the agent's field of view

Case Study

- The table & chair painting problem

- Initial state:

$Init(Object(Table) \wedge Object(Chair) \wedge Can(C1) \wedge Can(C2) \wedge InView(Table))$

- Goal: $Goal(Color(Chair, c) \wedge Color(Table, c))$
  - recall: in goal, variable c existentially quantified
- Actions:

$Action(RemoveLid(can),$
$Precond : Can(can)$
$Effect : Open(can))$
$Action(Paint(x, can),$
$Precond : Object(x) \wedge Can(can) \wedge Color(can, c) \wedge Open(can)$
$Effect : Color(x, c))$

c not part of action's variable list (partially observable only)

- Add an action causing objects to come into view (one at a time):

$Action(LookAt(x),$
$Precond : InView(y) \wedge (x \neq y)$
$Effect : InView(x) \wedge \neg InView(y))$

Case Study

- The table & chair painting problem
  - Partially-Observable Problems:
    need to reason about percepts obtained during action

  $\Longrightarrow$ Augment PDDL with percept schemata for each fluent. Ex:

    - $Percept(Color(x, c)$,
      $Precond : Object(x) \wedge InView(x)$
      "if an object is in view, then the agent will perceive its color"

    $\Longrightarrow$ perception will acquire the truth value of $Color(x, c)$, for every x, c

    - $Percept(Color(can, c)$,
      $Precond : Can(can) \wedge InView(can) \wedge Open(can)$
      "if an open can is in view, then the agent perceives the color of the paint in the can"

    $\Longrightarrow$ perception will acquire the truth value of $Color(can, c)$, f.e. can, c

  - Fully-Observable Problems:

  $\Longrightarrow$ Percept schemata with no preconditions for each fluent. Ex:

    - $Percept(Color(x, c))$

  - Sensorless Agent: no percept schema

Handling Indeterminacy

Sensorless planning (aka conformant planning):

- find plan that achieves goal in all possible circumstances

  (regardless of initial state and action effects)

  - for environments with no observations
  - ex: “Open any can of paint and apply it to both chair and table”

Conditional planning (aka contingency planning):

- construct conditional plan with different branches for possible contingencies

  - for partially-observable and nondeterministic environments
  - ex: “Sense color of table and chair;

    if they are the same, then finish, else sense can paint;

    if color(can) =color(furniture) then apply color to other piece;

    else apply color to both”

Handling Indeterminacy

Execution monitoring and replanning:

- while constructing plan, judge whether plan requires revision
  - for partially-known or evolving environments
  - ex: Same as conditional, and can fix errors

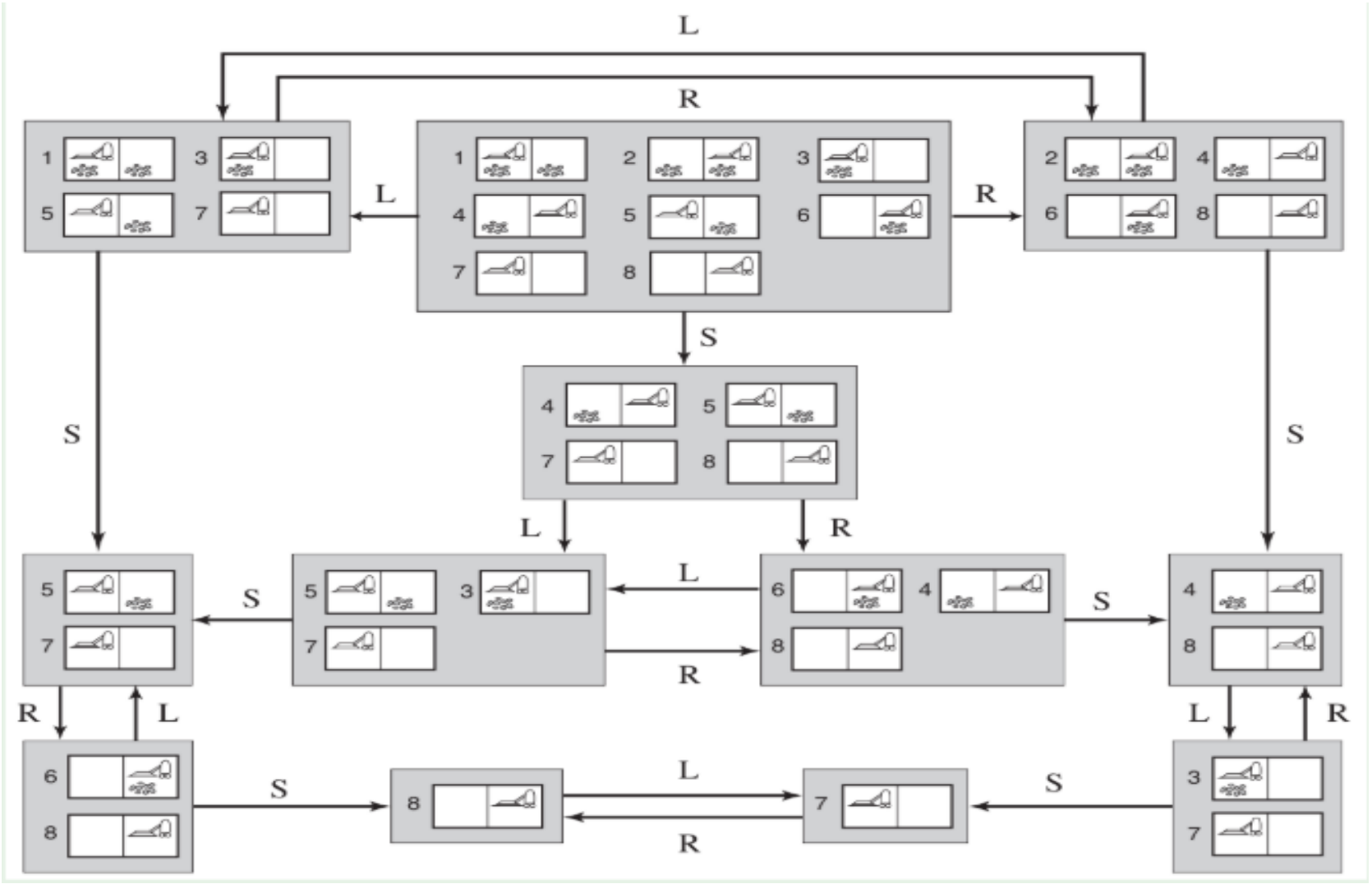
Sensorless planning (aka conformant planning)

Search with No Observation:

- aka Sensorless Search or Conformant Search
- Idea: To solve sensorless problems, the agent searches in the space of belief states rather than in that of physical states
  - fully observable, because the agent knows its own belief space
  - solutions are always sequences of actions (no contingency plan), because percepts are always empty and thus predictable
- Main drawback: 2^N candidate states rather than N

Belief-State Problem Formulation

Sensorless Vacuum Cleaner: Belief State Space



Sensorless planning (aka conformant planning)

- Main differences:
  - planners deal with factored representations rather than atomic
  - physical transition model is a collection of action schemas
  - the belief state represented by a logical formula instead of an explicitly-enumerated set of states
- Background knowledge: Object(Table) ∧ Object(Chair) ∧ Can(C1) ∧ Can(C2)
- Initial state b_0: Color(x, C(x)) any object has a color
- States must also include negative literals. A possible plan is:
- [RemoveLid(Can1), Paint(Chair, Can1), Paint(Table, Can1)]

Sensorless planning (aka conformant planning)

- Let's execute the plan:
- RemoveLid(Can1) in $b_0 \rightarrow b_1$ = Color(x,C(x)) ∧ Open(Can1)
- Paint(Chair, Can1) in $b_1 \rightarrow b_2$ = Color(x,C(x)) ∧ Open(Can1) ∧ Color(Chair, C(Can1))
- Paint(Table, Can1) in $b_2 \rightarrow b_3$ = Color(x,C(x)) ∧ Open(Can1) ∧ Color(Chair,C(Can1)) ∧ Color(Table, C(Can1))
- The final belief state satisfies the goal.
- b_3 |= Color(Table, c) ∧ Color(Chair, c)

⇒ [RemoveLid(Can1), Paint(Chair, Can1), Paint(Table, Can1)] - valid conformant plan

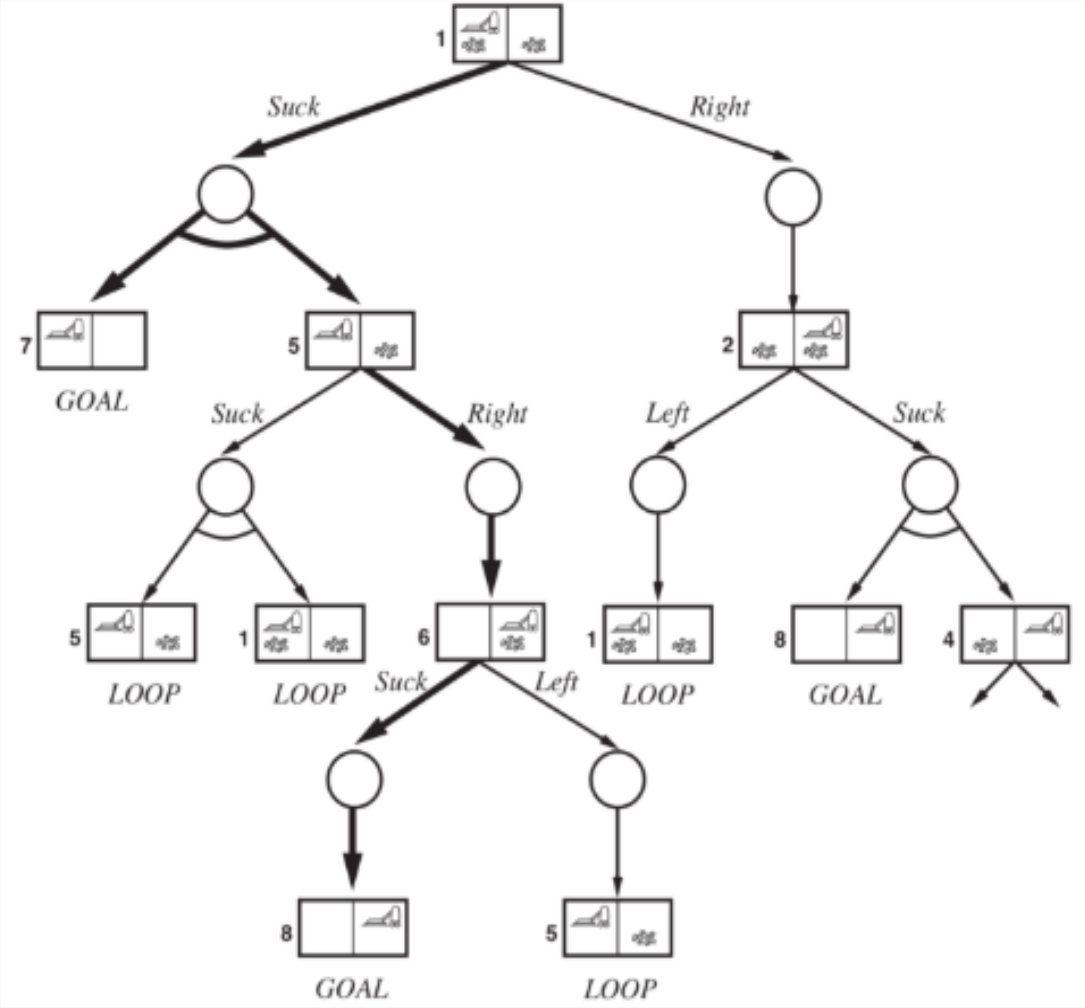
Conditional planning (aka contingency planning)

- Contingent planning is the generation of plans with conditional branching based on percepts.
- It is appropriate for environments with partial observability, nondeterminism, or both.
- The action is selected on the basis of perceptions.
- Different from conditional planning.

$$
\begin{array}{l}
[LookAt(Table), LookAt(Chair), \\
\quad \textbf{if } Color(Table, c) \wedge Color(Chair, c) \textbf{ then } NoOp \\
\qquad \textbf{else } [RemoveLid(Can_1), LookAt(Can_1), RemoveLid(Can_2), LookAt(Can_2), \\
\qquad\quad \textbf{if } Color(Table, c) \wedge Color(can, c) \textbf{ then } Paint(Chair, can) \\
\qquad\quad \textbf{else if } Color(Chair, c) \wedge Color(can, c) \textbf{ then } Paint(Table, can) \\
\qquad\quad \textbf{else } [Paint(Chair, Can_1), Paint(Table, Can_1)]]]
\end{array}
$$

Conditional planning (aka contingency planning)

Solution for [SUCK, IF STATE = 5 THEN [RIGHT, SUCK] ELSE [ ]]



Conditional planning (aka contingency planning)

- Computation of the new belief state is done in two steps:
  1) Compute the belief state as a result of actions (using add and delete lists)
  2) Update belief state according to perceptions.

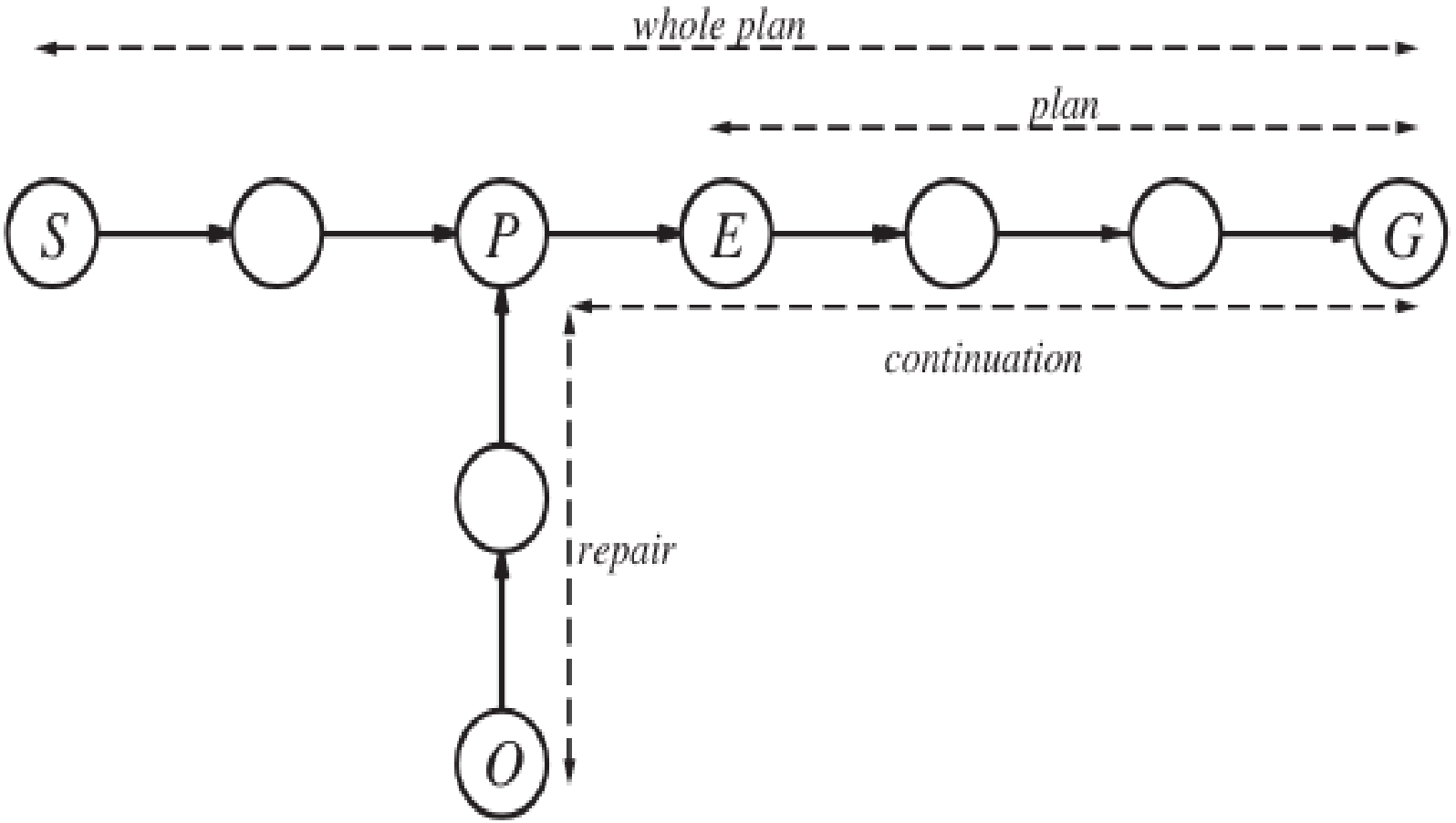
- Main differences:
  - planners deal with factored representations rather than atomic
  - physical transition model is a collection of action schema
  - the belief state represented by a logical formula instead of an explicitly-enumerated set of states
  - sets of belief states represented as disjunctions of logical formulas representing belief states

Online replanning

- In contingent lanning you have to consider all the contingencies that may arise.
- An alternative, or addition, to contingent planning is replanning during execution.
- Different reasons for replanning:
  - Inaccurate model (missing preconditions, missing effects, missing state variables …)
  - Exogenous events
- It requires some execution monitoring:
  - Action monitoring (preconds ok?)
  - Plan monitoring (rest of plan still ok?)
  - Goal monitoring (better goal?)

Online replanning

- Plan execution monitoring also allows for serendipity, accidental achievement of the goal by other means.



Multiagent Planning

- In multi-agent planning problems an agent tries to achieve its own goals with the help or despite the interference of others.
- There are intermediate scenarios in the spectrum from a single agent to a multi-agent scenario made of autonomous agents:
  - multi-effectors planning: the centralized plan has to control different effectors
  - multi-body planning: overall plan executed by different bodies, decentralized planning
  - multi-agent planning:
    - same goal but different plans, coordination is needed;
    - different goals as players in opposite teams in competitive environments , e.g. tennis or soccer.
    - a mixture …

Multiagent Planning

- The issues involved in multi-agent planning:
  1) representing and planning for multiple simultaneous actions
  2) cooperation, coordination, competition in multi-agent planning
- Multi-actor planning: where actors is a generic term covering effectors, bodies, and agents themselves.
- The different entities share a common goal and collaborate to achieve it.
- The planning is done centrally.
- We assume perfect synchronization.

Multiagent Planning

- Transition model: The single action a is replaced by a joint action $\langle a_1, \ldots, a_n \rangle$, where a_i is the action taken by the i^{th} actor.
- Two big problems, given that b is the number of possible actions:
  1) we have to describe the transition model for b^n different joint actions;
  2) we have a joint planning problem with a branching factor of b^n.
- The major concern is to decouple the actors so that each one can work independently to one sub-problem.

Loosely coupled sub-problems

- The standard approach to loosely coupled problems is to pretend the problems are completely decoupled and then fix up the interactions.
- Double tennis problem:

$$
\begin{aligned}
&Actors(A, B)\\
&Init(At(A, LeftBaseline) \wedge At(B, RightNet) \wedge\\
&\qquad Approaching(Ball, RightBaseline)) \wedge Partner(A, B) \wedge Partner(B, A)\\
&Goal(Returned(Ball) \wedge (At(a, RightNet) \vee At(a, LeftNet))\\
&Action(Hit(actor, Ball),\\
&\qquad \text{PRECOND:} Approaching(Ball, loc) \wedge At(actor, loc)\\
&\qquad \text{EFFECT:} Returned(Ball))\\
&Action(Go(actor, to),\\
&\qquad \text{PRECOND:} At(actor, loc) \wedge to \neq loc,\\
&\qquad \text{EFFECT:} At(actor, to) \wedge \neg At(actor, loc))
\end{aligned}
$$

Controlling interactions

- The following joint plan works:

PLAN 1:
A : [$Go(A, RightBaseline)$, $Hit(A, Ball)$]
B : [$NoOp(B)$, $NoOp(B)$]

- But in general, to restrict unwanted interactions we need to augment action schemas with a concurrent action list stating which actions must or must not be executed concurrently.
- The Hit action can be described as follows:

$Action(Hit(a, Ball)$,
CONCURRENT: $b \neq a \Rightarrow \neg Hit(b, Ball)$
PRECOND: $Approaching(Ball, loc) \wedge At(a, loc)$
EFFECT: $Returned(Ball))$

- Conversely, for some actions the desired effect is achieved only when another action occurs concurrently.
  - Example: carrying a heavy piece of furniture.

Multiple agent: cooperation and coordination

- Each agent makes its own plan. We assume that the goals and knowledge base are shared.
- There may be different plans for one goal.

PLAN 1:

$A : [Go(A, RightBaseline), Hit(A, Ball)]$

$B : [NoOp(B), NoOp(B)]$

PLAN 2:

$A : [Go(A, LeftNet), NoOp(A)]$

$B : [Go(B, RightBaseline), Hit(B, Ball)]$

- If agents choose different plans, the combined effect does not work. There is an issue of coordination.
- Different ways:
  1) Conventions and social laws, i.e. “stick to your side of the court”
  2) Communication, i.e. a tennis player could shout “Mine!”
  3) Plan recognition: recognize the joint plan that the other agent is starting to execute and behave accordingly.

Evolutionary conventions and flocking

- In both cases the behavior of animals in nature is a source of inspiration.

1) Conventions developed through evolutionary processes.
   - Colonies of ants execute very elaborate joint plans without any centralized control.
   - They do so playing very specific roles and strategies apparently with limited communication.

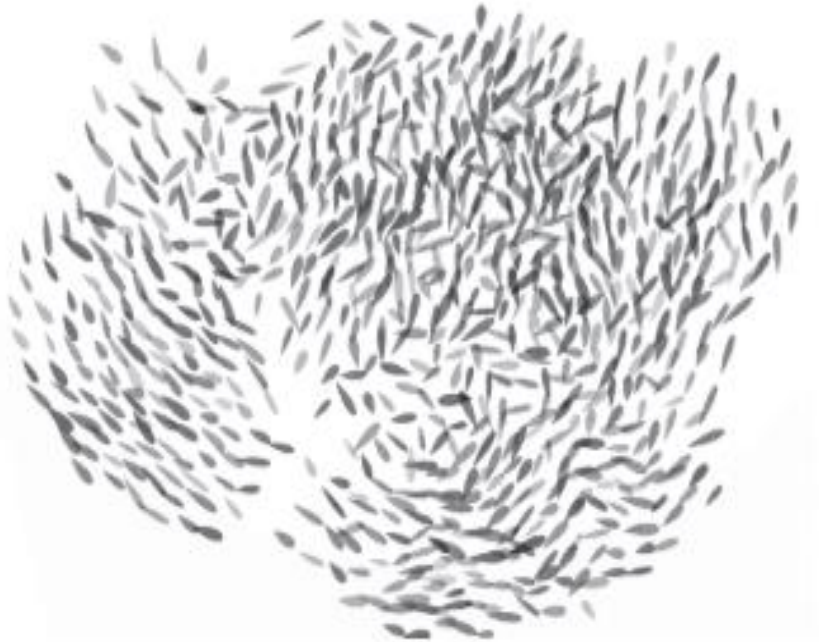
Evolutionary conventions and flocking

2) Flocking behavior of birds Maximize the weighted sum of three components:

- ❑ Cohesion: a positive score for getting closer to the average position of the neighbours
- ❑ Separation: a negative score for getting too close to any one neighbour
- ❑ Alignment: a positive score for getting closer to the average heading of the neighbours

❖ If all the boids execute this policy, the flock exhibits the EMERGENT BEHAVIOR of flying as a pseudo-rigid body with roughly constant density that does not disperse over time, and that occasionally makes sudden swooping motions.

Evolutionary conventions and flocking



(a)

Flocking simulation by local interactions

(b)

Flocking behavior

Disclaimer

The material for the presentation has been compiled from various sources such as prescribed text books by Russell and Norvig and other tutorials and lecture notes. The information contained in this lecture/ presentation is for educational purpose only.

Thank You *for* Your Attention !