Classical Planning

By
S.Komal Kaur
Assistant Professor, CSE Dept.,
VCE, Hyderabad

Outline

- Definition of Classical Planning
- Example
- Algorithms for Planning as State-Space Search
- Planning Graphs
- The GRAPHPLAN Algorithm
- Other Classical Planning Approaches

Planning

- Classical planning is how an agent can take advantage of the structure of a problem to construct complex plans of action.
- Rational action of agents implies some form of **planning**.
- That is they devise a plan of action to achieve a goal.
- The problem-solving agent can find sequences of actions that result in a goal state.
- It deals with atomic representations of states
- Needs good domain-specific heuristics to perform well.

Definition Of Classical Planning

- Factored representation

 One in which a state of the world is represent ed by a collection of variables.

  Planning Domain Definition Language (PDDL)
  Can be used to define a search problem:
  1.Initial state
  2.Actions that are available in a state
  3.Result of applying an action
  4.The goal test

PDDL

State:

- Represented as conjunction of fluents that are ground, functionless atoms.
- Eg. Poor ^ Unknown
- Eg. At(Truck1, Melbourne) ^ At(Truck2, Sydney)
- Database semantics are used the closed world assumption means that any fluents that are not are mentioned are false, and a unique names assumption means that Truck1 and Truck2 are distinct.
- Set semantics is used logical inference or a set of fluents.

Planning Domain Definition Language P DDL

Actions

- Described by a set of action schemas that implicitly define the ACTIONS(S) and RESULT(S,A)functions needed to do a problem-solving search.
- Classical planning concentrates on problems where most actions leave most things unchan ged.

Planning Domain Definition Language P DDL

- Ex. The action of nudging an object causes that object to
  change its location by a vector V.
- A concise description of the action should mention only V ; it should not have to mention all the objects that stay in place.
- PDDL specifies the result of an action in terms of what changes; everything that stays the same is unmentioned

PDDL -Action

- Action schema
- A set of ground actions
- The schema lifts the level of reasoning from propositional logic to a restricted subset of first order logic is called lifted representation.
- **Action(Fly(p,from,to),**
  PRECOND: At(p,from) ^ Plane(p) ^ Airport(from) ^ Airport(to)
- **EFFECT: ¬At(p,from) ^ At(p,to)**
- The schema consists of action name, a list of all the variables used in the
  schema, a precondition and an effect.

PDDL

- The precondition and effect of an action are each conjunctions of literals (positive or negated atomic sentences).
- The precondition defines the states in which the action can be executed, and the effect defines the result of executing the action.
- An action a can be executed in state s if s entails the precondition of a.

PDDL

- Entailment can also be expressed with the set semantics: s |= q iff every positive literal in q is in s and every negated literal in q is not.
- We say that action a is applicable in state a if the preconditions are satisfied by s.

$$(a \in \text{ACTIONS}(s)) \Leftrightarrow s \models \text{PRECOND}(a),$$

where any variables in a are universally quantified. For example,

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

Example: Air Cargo transport

- 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, and
- At(x,a) means that object x (either plane or cargo) is at airport a.

Example: Air Cargo transport

$Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK)$
$\quad \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2)$
$\quad \land Airport(JFK) \land Airport(SFO))$
$Goal(At(C_1, JFK) \land At(C_2, SFO))$
$Action(Load(c, p, a),$
$\quad$ PRECOND: $At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
$\quad$ EFFECT: $\neg At(c, a) \land In(c, p))$
$Action(Unload(c, p, a),$
$\quad$ PRECOND: $In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
$\quad$ EFFECT: $At(c, a) \land \neg In(c, p))$
$Action(Fly(p, from, to),$
$\quad$ PRECOND: $At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)$
$\quad$ EFFECT: $\neg At(p, from) \land At(p, to))$

Figure 10.1 A PDDL description of an air cargo transportation planning problem.

Example: The spare tire problem

- Consider the problem of changing a flat tire. The goal is to have a good spare tire properly mounted onto the car's axle, where the initial state has a flat tire on the axle and a good spare tire in the trunk.
- There are just four actions: removing the spare from the trunk, removing the flat tire from the axle, putting the spare on the axle, and leaving the car unattended overnight.

Example: The spare tire problem

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

Figure 10.2 The simple spare tire problem.

Example: The Block World

- One of the most 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.

Example: The Block World

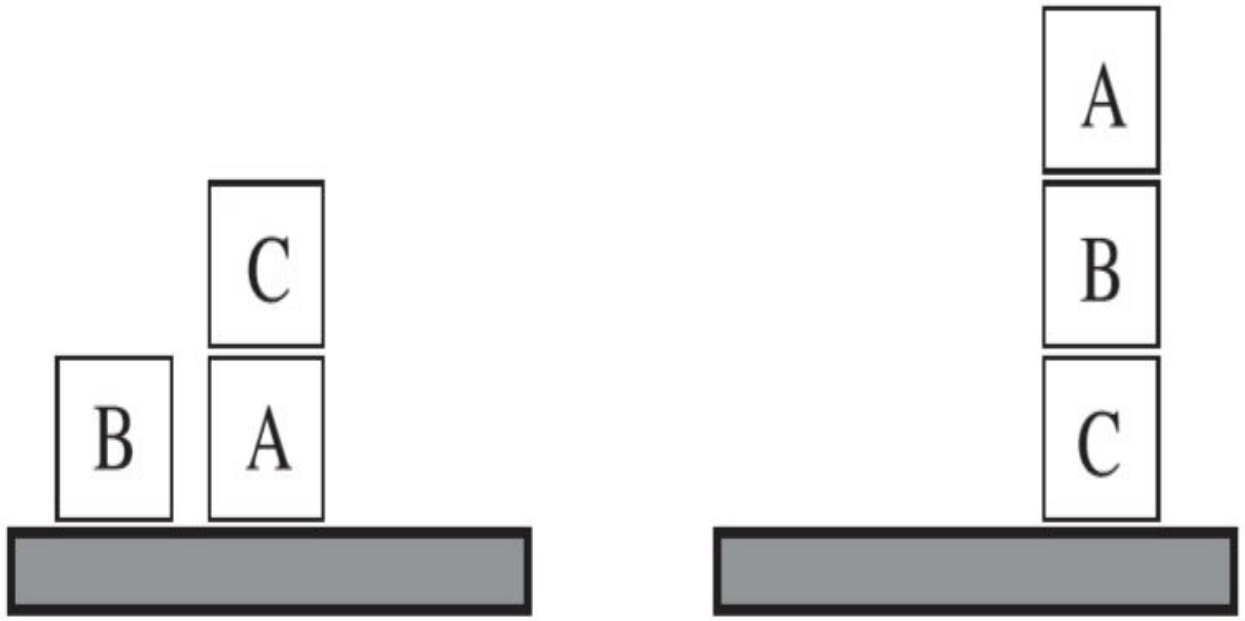

Figure 10.4 Diagram of the blocks-world problem in Figure 10.3.

Example: The Block World

$$Init(On(A, Table) \land On(B, Table) \land On(C, A) \land Block(A) \land Block(B) \land Block(C) \land Clear(B) \land Clear(C))$$

$$Goal(On(A, B) \land On(B, C))$$

$$Action(Move(b, x, y),$$
$$\text{PRECOND: } On(b, x) \land Clear(b) \land Clear(y) \land Block(b) \land Block(y) \land (b \neq x) \land (b \neq y) \land (x \neq y),$$
$$\text{EFFECT: } On(b, y) \land Clear(x) \land \neg On(b, x) \land \neg Clear(y))$$

$$Action(MoveToTable(b, x),$$
$$\text{PRECOND: } On(b, x) \land Clear(b) \land Block(b) \land (b \neq x),$$
$$\text{EFFECT: } On(b, Table) \land Clear(x) \land \neg On(b, x))$$

Figure 10.3 A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[MoveToTable(C, A), Move(B, Table, C), Move(A, Table, B)]$.

Algorithm For Planning as State-Space Search

- The description of a planning problem defines a search problem:
- Search from the initial state through the space of states, looking for a goal.
- An advantage of the declarative representation of action schemas is that we can also search backward from the goal,
  looking for the initial state.

Forward (Progression) state space search

- 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 (Progression) state space search

- Clearly, even this relatively small problem instance is hopeless without an accurate heuristic.
- Although many real-world applications of planning have relied on domain-specific heuristics, it turns out that strong domain-independent heuristics can be derived automatically; that is what makes forward search feasible.

Compare forward and backward searches

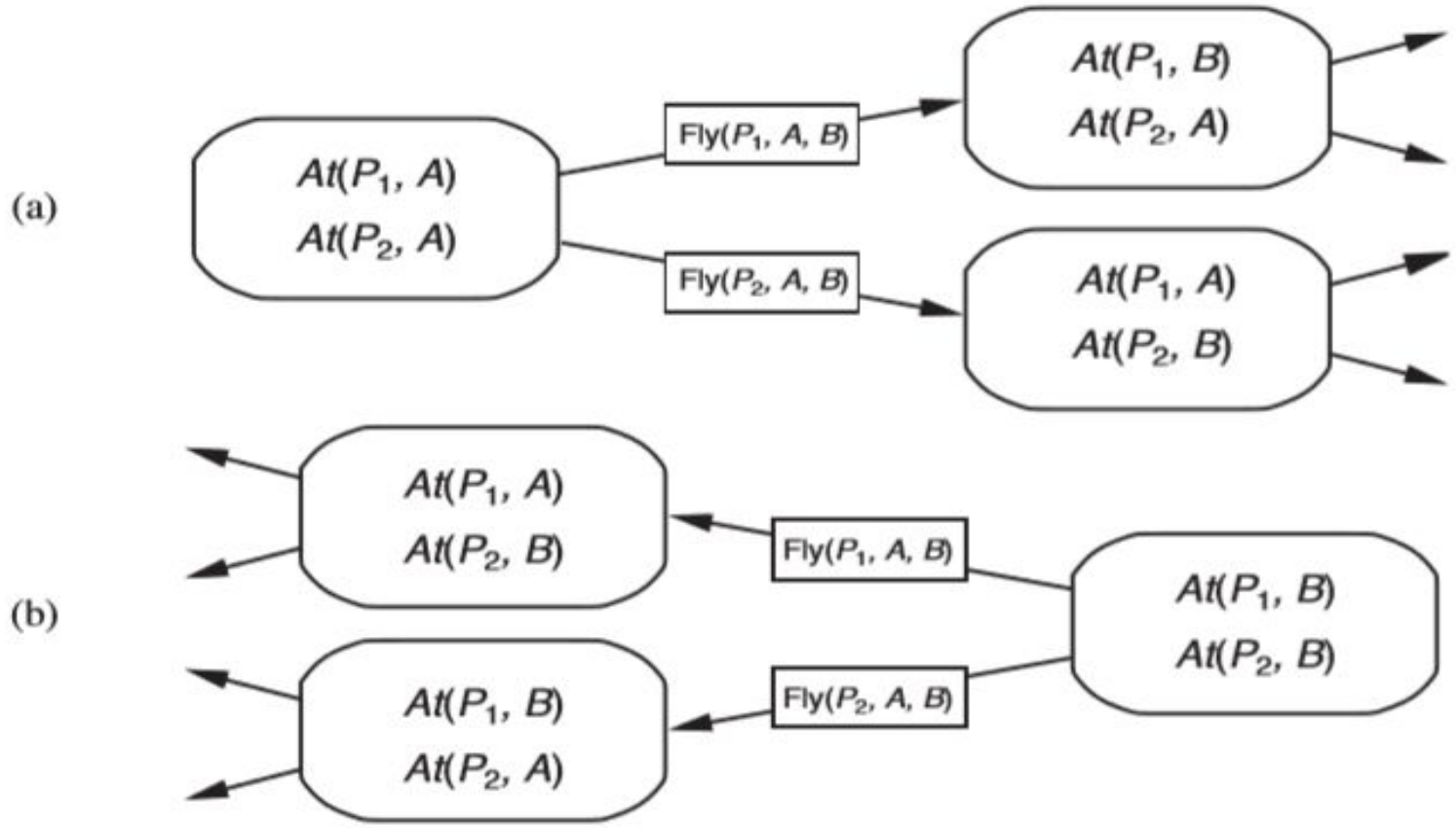

Figure 10.5 Two approaches to searching for a plan. (a) Forward (progression) search through the space of states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through sets of relevant states, starting at the set of states representing the goal and using the inverse of the actions to search backward for the initial state.

Backward (Regression) relevant states search

- In regression search we start at the goal and apply the actions backward until we find a sequence of steps that reaches the initial state.
- It is called relevant-states search because we only consider actions that are relevant to the goal (or current state).
- As in belief-state search, there is a set of relevant states to consider at each step, not just a single state.

Backward (Regression) relevant states search

- To get the full advantage of backward search, we need to deal with partially un-instantiated actions and states, not just ground ones.
- 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(C2,p,SFO),

PRECOND: In(C2,p) $\wedge$ At(p,SFO) $\wedge$ Cargo(C2) $\wedge$ Plane(p) $\wedge$ Airport(SFO)

EFFECT: At(C2,SFO) $\wedge \neg$In(C2,p)

The regressed state description is

G' = In(C2,p) $\wedge$ At(p,SFO) $\wedge$ Cargo(C2) $\wedge$ Plane(p) $\wedge$ Airport(SFO).

Heuristic for Planning

- Neither forward nor backward search is efficient without a good heuristic function
- A heuristic function h(s) estimates the distance from a state s to the goal and that if we can derive an admissible heuristic for this distance—one that does not overestimate—then we can use A* search to find optimal solutions.
- An admissible heuristic can be derived by defining a relaxed problem that is easier to solve.
- The exact cost of a solution to this easier problem then becomes the heuristic for the original problem.

Planning Graphs

- This section shows how a special data structure called a planning graph can be used to give better heuristic estimates.
- These heuristics can be applied to any of the search techniques we have seen so far.
- Alternatively, we can search for a solution over the space formed by the planning graph, using an algorithm called GRAPHPLAN.

Planning Graph

- A planning graph is a directed graph organized into levels:
- First a level S0 for the initial state, consisting of nodes representing each fluent that holds in S0;
- Then a level A0 consisting of nodes for each ground action that might be applicable in S0; then alternating levels Si followed by Ai; until we reach a termination condition.

Simple Planning Problem

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

Figure 10.7 The "have cake and eat cake too" problem.

Planning Graph

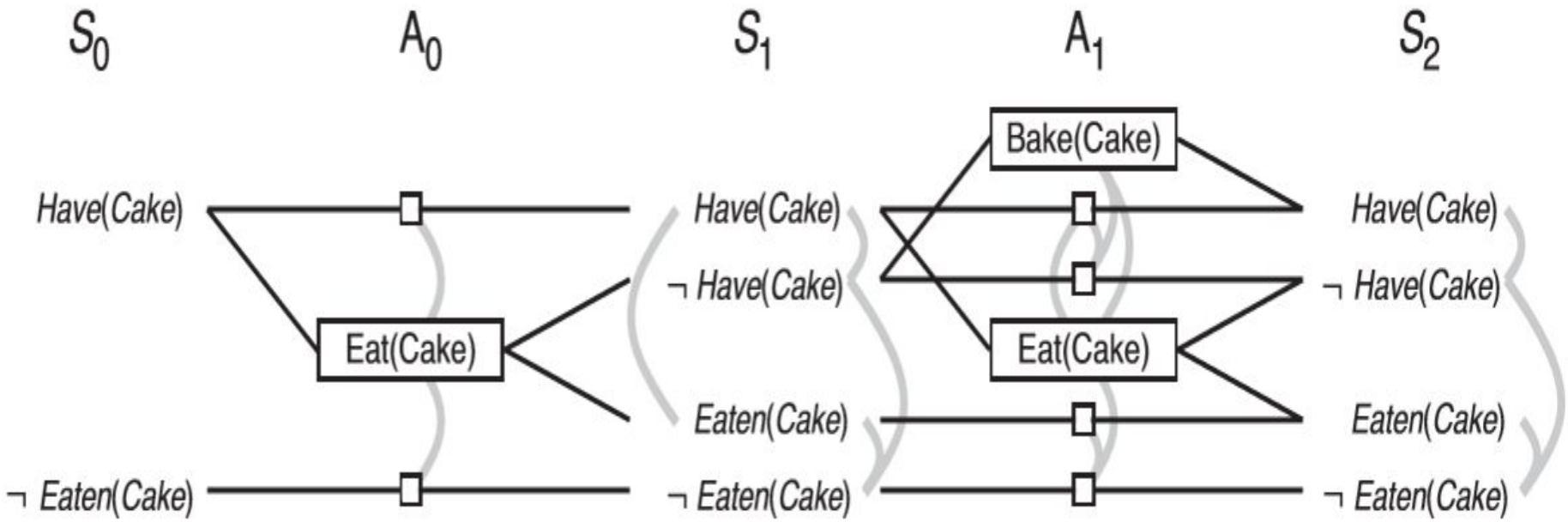

Figure 10.8 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions), and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines. Not all mutex links are shown, because the graph would be too cluttered. In general, if two literals are mutex at S_i, then the persistence actions for those literals will be mutex at A_i and we need not draw that mutex link.

Planning Graph

- Figure 10.7 shows a simple planning problem, and Figure 10.8 shows its planning graph.
- Each action at level Ai is connected to its preconditions at Si and its effects at Si+1.
- So a literal appears because an action caused it, but we also want to say that a literal can persist if no action negates it.
- This is represented by a persistence action (sometimes called a no-op).
- Level A0 in Figure 10.8 shows one “real” action, Eat(Cake), along with two persistence actions drawn as small square boxes.

Planning Graph

- Level A0 contains all the actions that could occur in state S0, but just as important it records conflicts between actions that would prevent them from occurring together.
- The gray lines in Figure 10.8 indicate mutual exclusion (or mutex) links.
- For example, Eat(Cake) is mutually exclusive with the persistence of either Have(Cake) or ¬Eaten(Cake).

Planning Graph

- Alternating between state level Si and action level Ai until we reach a point where two consecutive levels are identical.
- At this point, we say that the graph has leveled off. The graph in Figure 10.8 levels off at S2.

GRAPHPLAN ALGORITHM

- The GRAPHPLAN algorithm repeatedly adds a level to a planning graph with EXPAND-GRAPH.
- Once all the goals show up as non mutex in the graph, GRAPHPLAN calls EXTRACT-SOLUTION to search for a plan that solves the problem.
- If that fails, it expands another level and tries again, terminating with failure when there is no reason to go on.

GRAPHPLAN ALGORITHM

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

  graph ← INITIAL-PLANNING-GRAPH(problem)
  goals ← CONJUNCTS(problem.GOAL)
  nogoods ← an empty hash table
  for tl = 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)
```

Figure 10.9 The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is possible.

GRAPHPLAN ALGORITHM

- Let us now trace the operation of GRAPHPLAN on the spare tire problem.
- The first line of GRAPHPLAN initializes the planning graph to a one-level (S0) graph representing the initial state.
- The positive fluents from the problem description's initial state are shown, as are the relevant negative fluents.

GRAPHPLAN ALGORITHM

- The goal At(Spare,Axle) is not present in S0, so we need not call EXTRACT-SOLUTION— we are certain that there is no solution yet.
- Instead, EXPAND-GRAPH adds into A0 the three actions whose preconditions exist at level S0 (i.e., all the actions except PutOn(Spare,Axle)), along with persistence actions for all the literals in S0.
- The effects of the actions are added at level S1.
- EXPAND-GRAPH then looks for mutex relations and adds them to the graph.

Planning graph

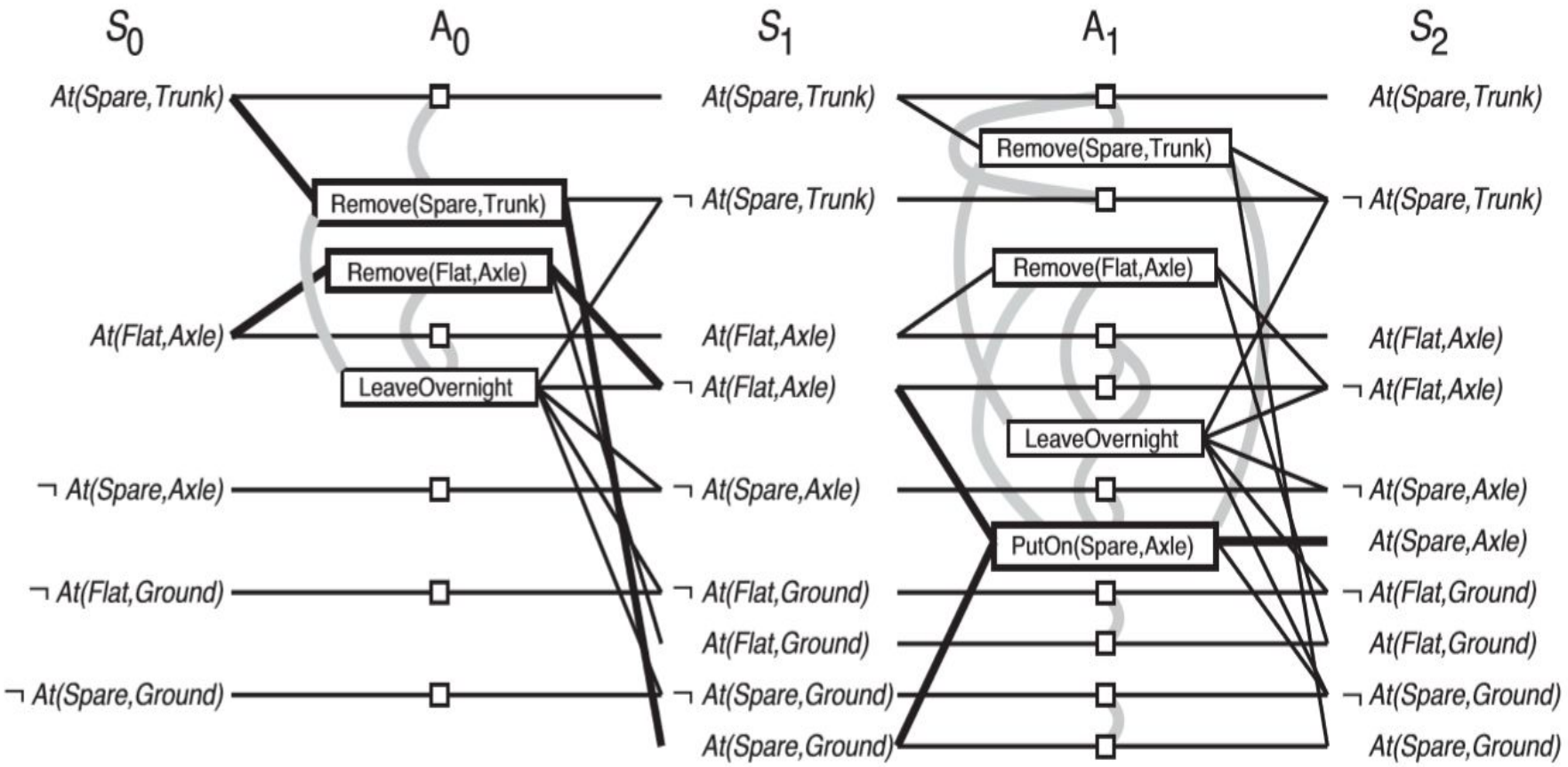

Figure 10.10 The planning graph for the spare tire problem after expansion to level S_2. Mutex links are shown as gray lines. Not all links are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.

Planning graph

- Alternatively, we can define EXTRACT-SOLUTION as a backward search problem, where each state in the search contains a pointer to a level in the planning graph and a set of unsatisfied goals.
- We define this search problem as follows:
- The initial state is the last level of the planning graph, Sn, along with the set of goals from the planning problem.

Planning graph

- The actions available in a state at level Si are to select any conflict-free subset of the actions in Ai−1 whose effects cover the goals in the state. The resulting state has level Si−1 and has as its set of goals the preconditions for the selected set of actions. By “conflict free,” we mean a set of actions such that no two of them are mutex and no two of their preconditions are mutex.

Planning graph

- The goal is to reach a state at level S0 such that all the goals are satisfied.
- The cost of each action is 1.

  For this particular problem, we start at S2 with the goal At(Spare,Axle). The only choice we have for achieving the goal set is PutOn(Spare,Axle). That brings us to a search state at S1 with goals At(Spare,Ground) and ¬At(Flat,Axle).

Planning graph

- The former can be achieved only by Remove(Spare,Trunk), and the latter by either Remove(Flat,Axle) or Leave Overnight.
- But LeaveOvernight is mutex with Remove(Spare,Trunk) , so the only solution is to choose Remove(Spare,Trunk) and Remove(Flat,Axle).
- That brings us to a search state at S0 with the goals At(Spare,Trunk) and At(Flat,Axle).
- Both of these are present in the state, so we have a solution: the actions Remove(Spare, Trunk) and Remove(Flat, Axle) in level A0, followed by PutOn(Spare, Axle) in A1.

GraphPlan

- In the case where EXTRACT-SOLUTION fails to find a solution for a set of goals at a given level, we record the (level,goals) pair as a no-good.
- Whenever EXTRACT-SOLUTION is called again with the same level and goals, we can find the recorded no-good and immediately return failure rather than searching again.

OTHER CLASSICAL PLANNING APPROACHES

- In this section we first describe the translation to a satisfiability problem and
- Then describe other influential approaches:
- planning as first-order logical deduction;
- and as plan refinement.

Classical planning as Boolean satisfiability

- Here we show how to translate a PDDL description into a form that can be processed by SATPLAN.
- The translation is a series of straightforward steps:

1. **Propositionalize the actions**: replace each action schema with a set of ground actions formed by substituting constants for each of the variables. These ground actions are not part of the translation, but will be used in subsequent steps.

Classical planning as Boolean satisfiability

2. **Define the initial state**: assert F0 for every fluent F in the problem's initial state, and ¬F for every fluent not mentioned in the initial state.
3. **Propositionalize the goal**: for every variable in the goal, replace the literals that contain the variable with a disjunction over constants.

For example, the goal of having block A on another block, On(A,x) $\wedge$ Block(x) in a world with objects A,B and C, would be replaced by the goal

(On(A,A) $\wedge$ Block(A)) $\vee$ (On(A,B) $\wedge$ Block(B)) $\vee$ (On(A,C) $\wedge$ Block(C)).

Classical planning as Boolean satisfiability

4. **Add successor-state axioms**: For each fluent F, add an axiom of the form.

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee \left(F^t \wedge \neg ActionCausesNotF^t\right),$$

where ActionCausesF is a disjunction of all the ground actions that have F in their add list, and ActionCausesNotF is a disjunction of all the ground actions that have F in their delete list.

5. **Add precondition axioms**: For each ground action A, add the axiom $A_t \Rightarrow PRE(A)_t$, that is, if an action is taken at time t, then the preconditions must have been true.
6. **Add action exclusion axioms**: say that every action is distinct from every other action.

Planning as first-order logical deduction: Situation calculus

- First-order logic lets us get around this limitation by replacing the notion of linear time with a notion of branching situations, using a representation called situation calculus that works like this:

1. The initial state is called a situation. If s is a situation and a is an action, then RESULT(s,a) is also a situation. There are no other situations. Thus, a situation corresponds to a sequence, or history, of actions. You can also think of a situation as the result of applying the actions, but note that two situations are the same only if their start and actions are the same: (RESULT(s,a)=RESULT(s,a)) $\Leftrightarrow$ (s = s $\wedge$ a = a).

Planning as first-order logical deduction: Situation calculus

- A function or relation that can vary from one situation to the next is a fluent. By convention, the situation s is always the last argument to the fluent, for example At(x,l,s) is a relational fluent that is true when object x is at location l in situation s, and Location is a functional fluent such that Location(x,s) = l holds in the same situations as At(x,l,s).

Planning as first-order logical deduction: Situation calculus

- Each action's preconditions are described with a possibility axiom that says when the action can be taken.
- An example from the wumpus world says that it is possible to shoot if the agent is alive and has an arrow:

  Alive(Agent,s) $\wedge$ Have(Agent,Arrow,s) $\Rightarrow$ Poss(Shoot,s)

Situation Calculus

- Each fluent is described with a successor-state axiom that says what happens to the fluent, depending on what action is taken.
- Action is possible $\Rightarrow$

  (Fluent is true in result state $\Leftrightarrow$ Action's effect made it true $\vee$ It was true before and action left it alone) .

Situation Calculus

- For example, the axiom for the relational fluent Holding says that the agent is holding some gold g after executing a possible action if and only if the action was a Grab of g or if the agent was already holding g and the action was not releasing it:
- Poss(a,s) $\Rightarrow$

(Holding(Agent,g,Result(a,s)) $\Leftrightarrow$ a=Grab(g) $\vee$ (Holding(Agent,g,s) $\wedge$ a != Release(g))) .

Situation Calculus

- We need unique action axioms so that the agent can deduce that, for example, a!=Release(g).
- For each distinct pair of action names Ai and Aj we have an axiom that says the actions are different:

  $$Ai(x,...) = Aj(y,...)$$

  and for each action name Ai we have an axiom that says two uses of that action name are equal if and only if all their arguments are equal:

  $$Ai(x1,...,xn)=Ai(y1,...,yn) \Leftrightarrow x1 = y1 \wedge ... \wedge xn = yn .$$
- A solution is a situation (and hence a sequence of actions) that satisfies the goal.

Situation Calculus

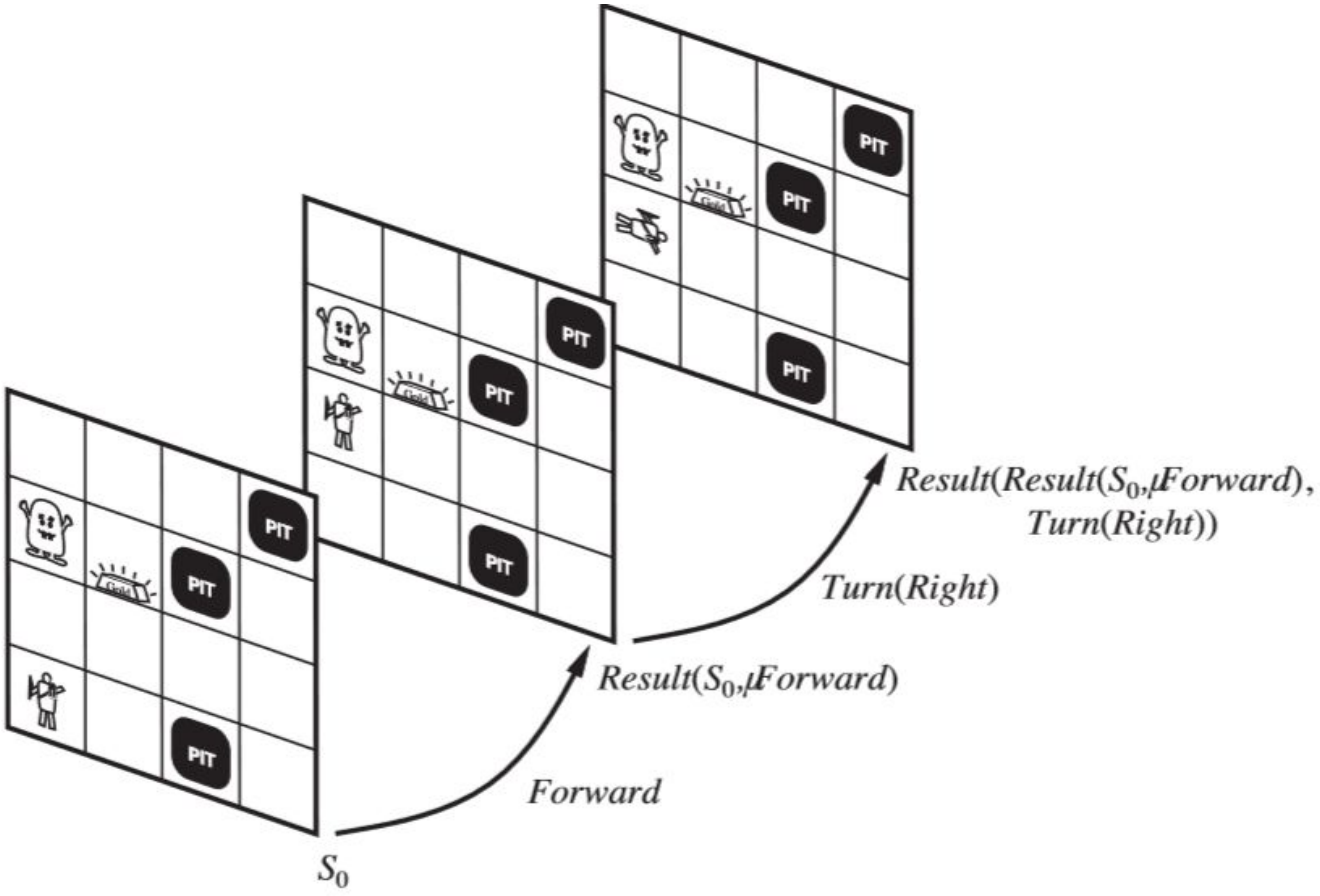

Figure 10.12 Situations as the results of actions in the wumpus world.

Planning as refinement of partially ordered plans

- All the approaches we have seen so far construct totally ordered plans consisting of a strictly linear sequences of actions. This representation ignores the fact that many subproblems are independent.
- An alternative is to represent plans as partially ordered structures: a plan is a set of actions and a set of constraints of the form Before(ai,aj) saying that one action occurs before another.

Planning as refinement of partially ordered plans

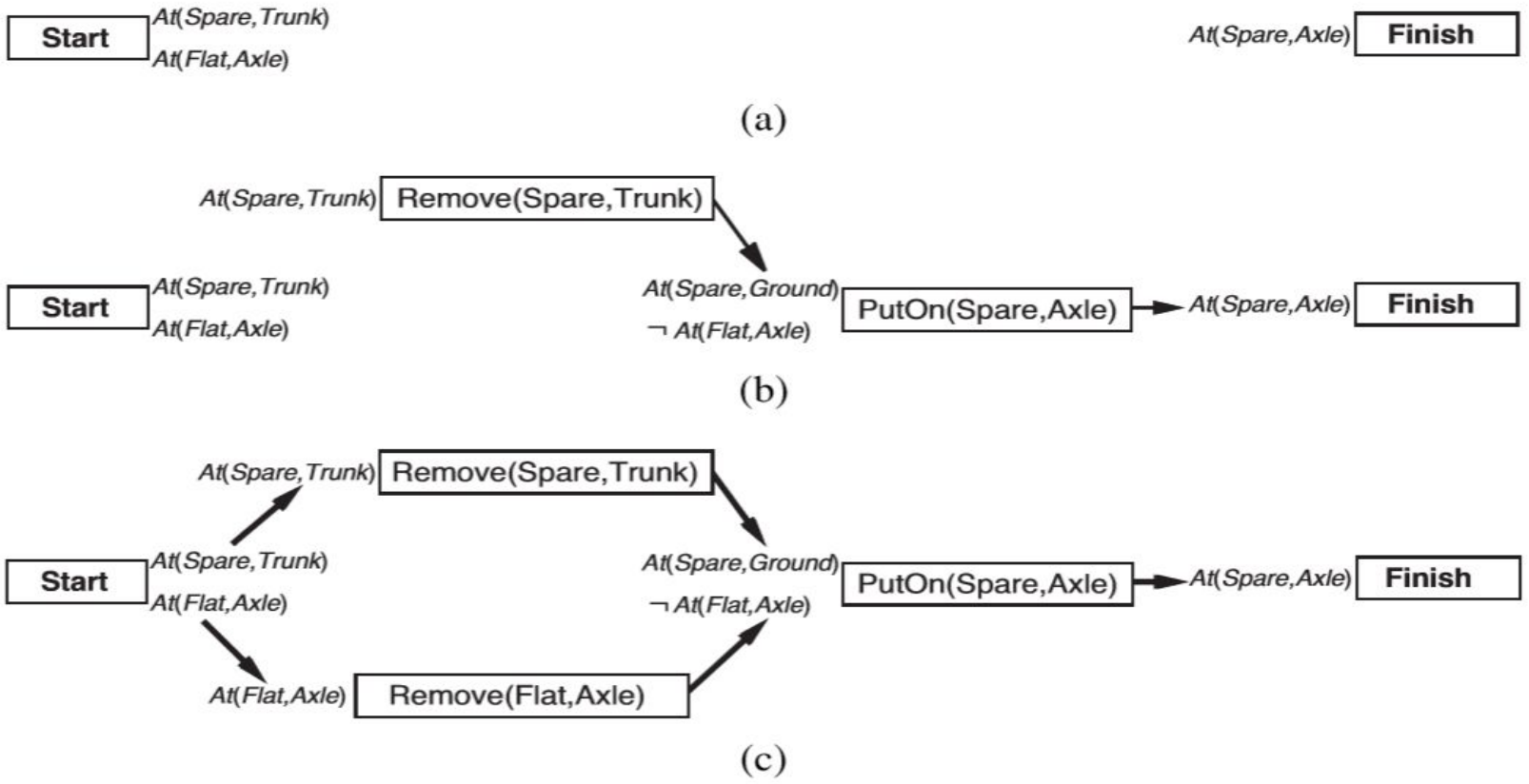

Figure 10.13 (a) the tire problem expressed as an empty plan. (b) an incomplete partially ordered plan for the tire problem. Boxes represent actions and arrows indicate that one action must occur before another. (c) a complete partially-ordered solution.

Planning as refinement of partially ordered plans

- In the bottom of Figure10.13, we see a partially ordered plan that is a solution to the spare tire problem.
- Actions are boxes and ordering constraints are arrows.
- Note that Remove(Spare,Trunk) and Remove(Flat,Axle) can be done in either order as long as they are both completed before the PutOn(Spare,Axle) action.

Thank You!