PLANNING AND ACTING IN THE REAL WORLD

12.1 Time. Schedules. and Resources

12.2 Hierarchical Task Network Planning

12.3 Planning and Acting in Nondeterministic Domains

12.4 Conditional Planning

12.5 Execution Monitoring and Replanning

12.6 Continuous Planning

12.7 MultiAgent Planning

12.1 Time. Schedules. and Resources

The STRIPS representation talks about *what* actions do, but, because the representation is based on situation calculus, it cannot talk about *how long* an action takes or even about *when* an action occurs, except to say that it is before or after another action

Time is of the essence in the general family of appllications called **job shop scheduling.** Such tasks require completing a set of jobs, each of whi.ch consists of a sequence of actions, where each action has a given duration and might require some resources.

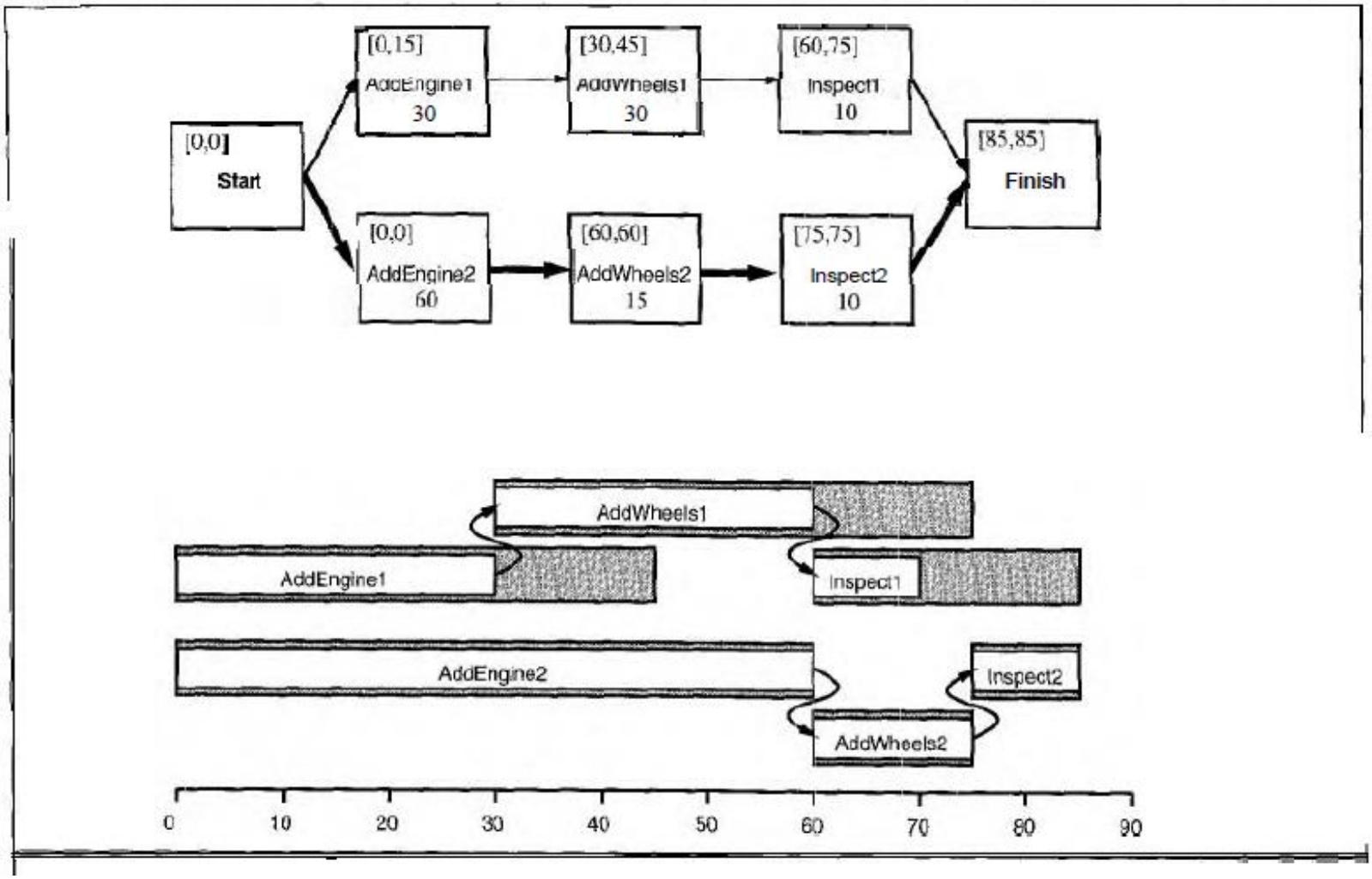
An example of a job shop scheduling problem is given in Figure 12.1. This is a highly simplified automobile assembly problem.

$Init(Chassis(C_1) \land Chassis(C_2)$
$\quad \land Engine(E_1, C_1, 30) \land Engine(E_2, C_2, 60)$
$\quad \land Wheels(W_1, C_1, 30) \land Wheels(W_2, C_2, 15))$
$Goal(Done(C_1) \land Done(C_2))$

$Action(AddEngine(e,c),$
$\quad$ PRECOND: $Engine(e, c, d) \land Chassis(c) \land \neg EngineIn(c),$
$\quad$ EFFECT: $EngineIn(c) \land Duration(d))$
$Action(AddWheels(w, c),$
$\quad$ PRECOND: $Wheels(w, c, d) \land Chassis(c) \land EngineIn(c),$
$\quad$ EFFECT: $WheelsOn(c) \land Duration(d))$
$Action(Inspect(c),$ PRECOND: $EngineIn(c) \land WheelsOn(c) \land Chassis(c),$
$\quad$ EFFECT: $Done(c) \land Duration(10))$

Figure 12.1 A job shop scheduling problem for assembling two cars. The notation $Duration(d)$ means that an action takes d minutes to execute. $Engine(E_1, C_1, 60)$ means that E_1 is an engine that fits into chassis C_1 and takes *60* minutes to install.

The problem in Figure 12.1 can be solved by any of the planners we have already seen. Figure 12.2 (if you ignore the numbers) shows the solution that the partial-order planner POP



The following formulas serve as a definition for *ES* and *LS* and also as the outline of a dynamic programming algorithm to compute them:

$$
\begin{aligned}
&ES(Start) = 0 \ . \\
&ES(B) = \max_{A \prec B} ES(A) + Duration(A) \ . \\
&LS(Finish) = ES(Finish) \ . \\
&LS(A) = \min_{A \prec B} LS(B) - Duration(A) \ .
\end{aligned}
$$

The complexity of the critical path algorithm is just O(Nb), where N is the number of actions and b is the maximum branching factor into or out of an action

Scheduling with resource constraints

Real scheduling problems are complicated by the presence of constraints on resources. For example, adding an engine to a car requires an engine hoist. If there is only one hoist, then we cannot simultaneously add engine *El* to car C1 and engine *E2* to car *Cz;*

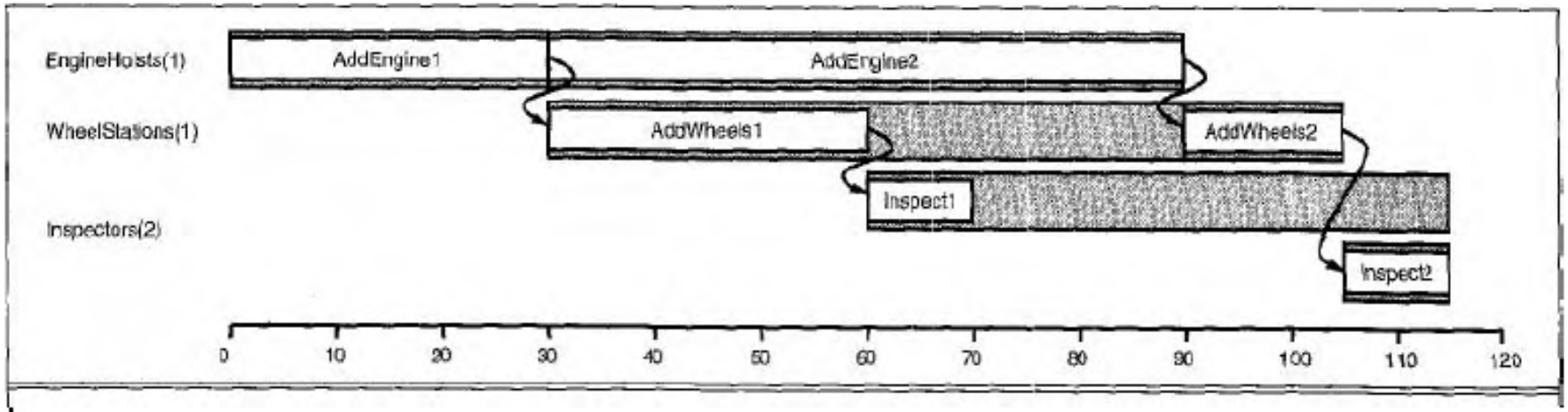


Figure 12.4 A solution to the job shop scheduling problem with resources from Figure 12.3. The left-hand margin lists the three resources, and actions are shown aligned horizontally with the resources they consume. There are two possible schedules, depending on which assembly uses the engine station first; we've shown the optimal solution, which takes 115 minutes.

12.2 Hierarchical Task Network Planning

One of the most pervasive ideas for dealing with complexity is **hierarchical decomposition.**

This section describes a planning method based on **hierarchical task networks** or HTNs.

In HTN planning, the initial plan, which describes the problem, is viewed as a very high-level description of what is to be done-

For example, building a house. Plans are refined by applying **action decompositions.**

Each action decomposition reduces a high-level action to a partially ordered set of lower-level actions.

For example, building a house might be reduced to obtaining a permit, hiring a contractor, doing the construction, and paying the contractor

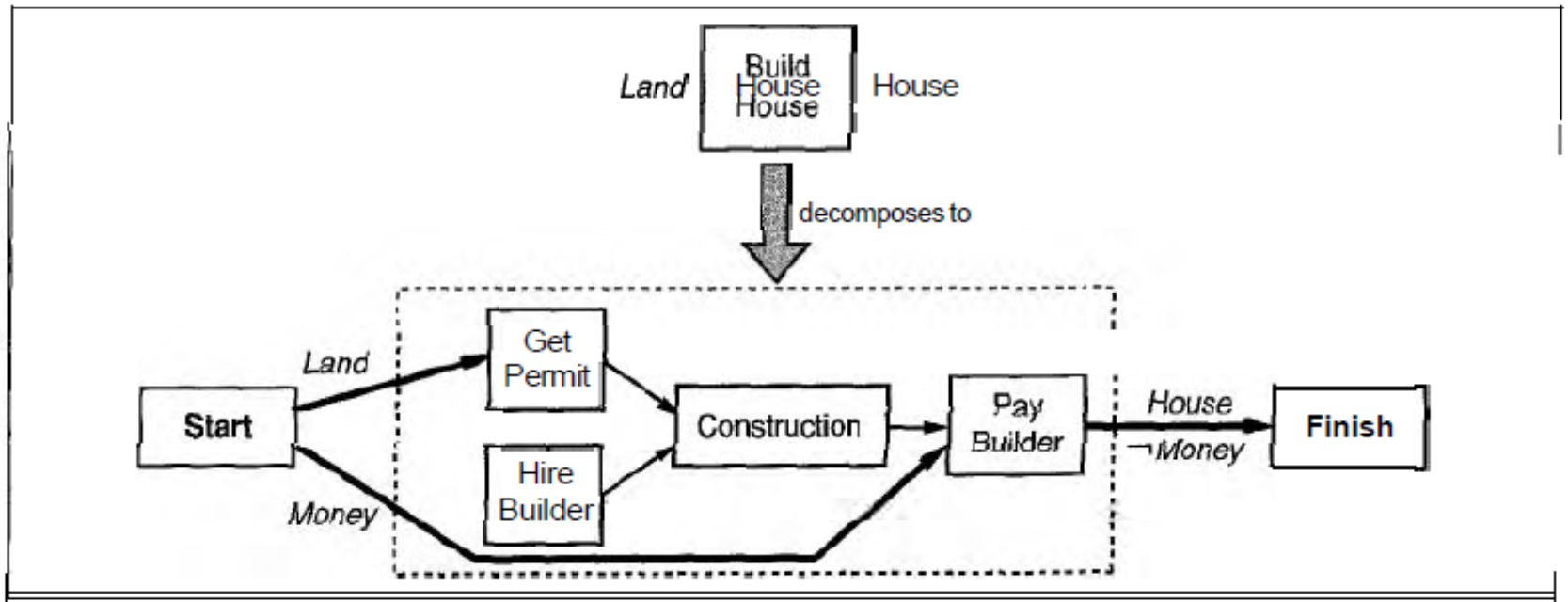


Figure 12.5 One possible decomposition for the *BuildHouse* action.

Action(*BuyLand*, PRECOND:*Money*, EFFECT:*Land* ∧ ¬ *Money*)
Action(*GetLoan*, PRECOND: *GoodCredit*, EFFECT:*Money* ∧ *Mortgage*)
Action(*BuildHouse*, PRECOND:*Land*, EFFECT:*House*)

Action(*GetPermit*, PRECOND: *Land*, EFFECT: *Permit*)
Action(*HireBuilder*, EFFECT: *Contract*)
Action(*Construction*, PRECOND:*Permit* ∧ *Contract*,
EFFECT:*HouseBuilt* ∧ ¬ *Permit*)
Action(*PayBuilder*, PRECOND:*Money* ∧ *HouseBuilt*,
EFFECT: ¬ *Money* ∧ *House* ∧ ¬ *Contract*)

Decompose(*BuildHouse*,
Plan(STEPS: $\{S_1 : GetPermit, S_2 : HireBuilder,$
$S_3 : Construction, S_4 : PayBuilder\}$
ORDERINGS: $\{Start \prec S_1 \prec S_3 \prec S_4 \prec Finish, \quad Start \prec S_2 \prec S_3\}$,
LINKS: $\{Start \xrightarrow{Land} S_1, Start \xrightarrow{Money} S_4,$
$S_1 \xrightarrow{Permit} S_3, S_2 \xrightarrow{Contract} S_3, S_3 \xrightarrow{HouseBuilt} S_4,$
$S_4 \xrightarrow{House} Finish, S_4 \xrightarrow{\neg Money} Finish\}$))

Representing action decompositions

General descriptions of action decomposition methods are stored in a **plan library,** from which they are extracted and instantiated to fit the needs of the plan being constructed

The *Start* action of the decomposition supplies all those preconditions of actions in the plan that are not supplied by other actions. We call these i.he **external preconditions.** In our example, the external preconditions of the deconnposition are Land and Money

external effects, which are the preconditions of Finish

the external effects of BuildHouse are House and ¬money.

primary effects, such as House, and

secondary effects, such as money.

Modifying the planner for decompositions

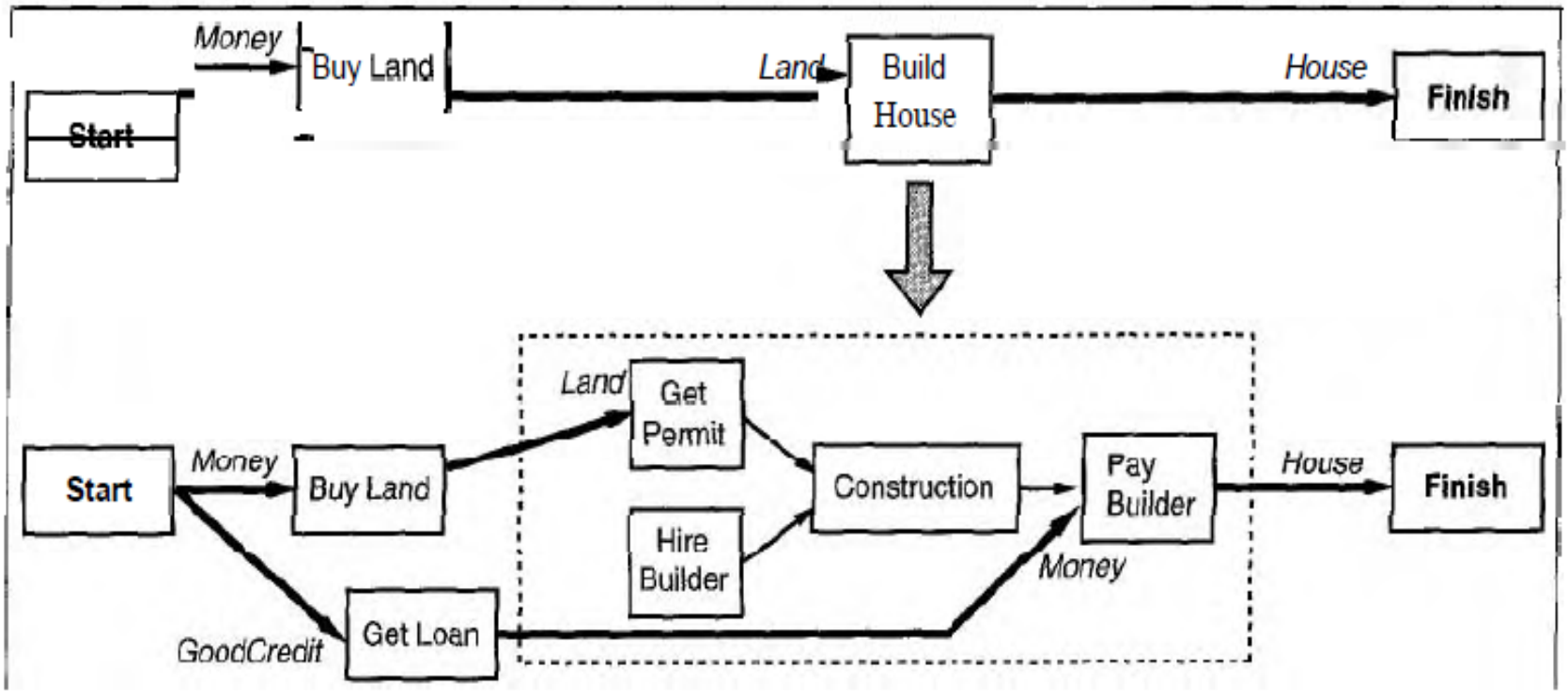


Figure 12.7 Decomposition of a high-level action within an existing plan. The *BuildHouse* action is replaced by the decomposition from Figure 12.5. The external precondition *Land* is supplied *by* the existing causal link from *BuyLand*. The external precondition *Money* remains open after the decomposition step, so we add a new action, *GetLoan*.

12.3 PLANNING AND ACTING IN NONDETERMINISTIC DOMAINS

So far we have considered only **classical planning** domains that are fully observable, static, and deterministic.

In an uncertain environment,

on the other hand, an agent must use its perceptsto discover what is happening while the plan is being executed and possibly modify or replace the plan if something unexpected happens.

The possibility of having complete or correct knowledge depends on *how much* indeterminacy there is in the world. With **bounded indeterminacy,** actions can have unpredictable

There are four planning methods for handling indeterminacy.

The first two are suitable for bounded indeterminacy, and the second two for unbounded indeterminacy:

1. Sensorless planning: Also called conformant pdanning, this method constructs standard, sequential plans that are to be executed without perception
2. Conditional planning: Also known as contingency planning, this approach deals with bounded indeterminacy by constructing a conditional plan with different branches for the different contingencies that could arise

3. Execution monitoring and replanning: In this approach, the agent can use any of the preceding planning techniques (classical, sensorless, or conditional) to construct a plan, but it also uses execution monitoring to judge whether the plan has a provision for the actual current situation or need to be revised.

4. Continuous planning: All the planners we have seen so far are designed to achieve a goal and then stop. A continuous planner is designed to persist over a lifetime.

Conditional planning is a way to deal with uncertainty by checking what is actually happening in the environment at predetermined points in the plan. Conditional planning is simplest to explain for fully observable environments, so we will begin with that case. The partially observable case is more difficult, but more interesting.

Conditional planning in fully observable environments

$$Action(Left, \text{PRECOND:}AtR, \text{EFFECT:}AtL \wedge \neg AtR)$$

must be modified to include a disjunctive effect:

$$Action(Left, \text{PRECOND:}AtR, \text{EFFECT:}AtL \vee AtR) .$$

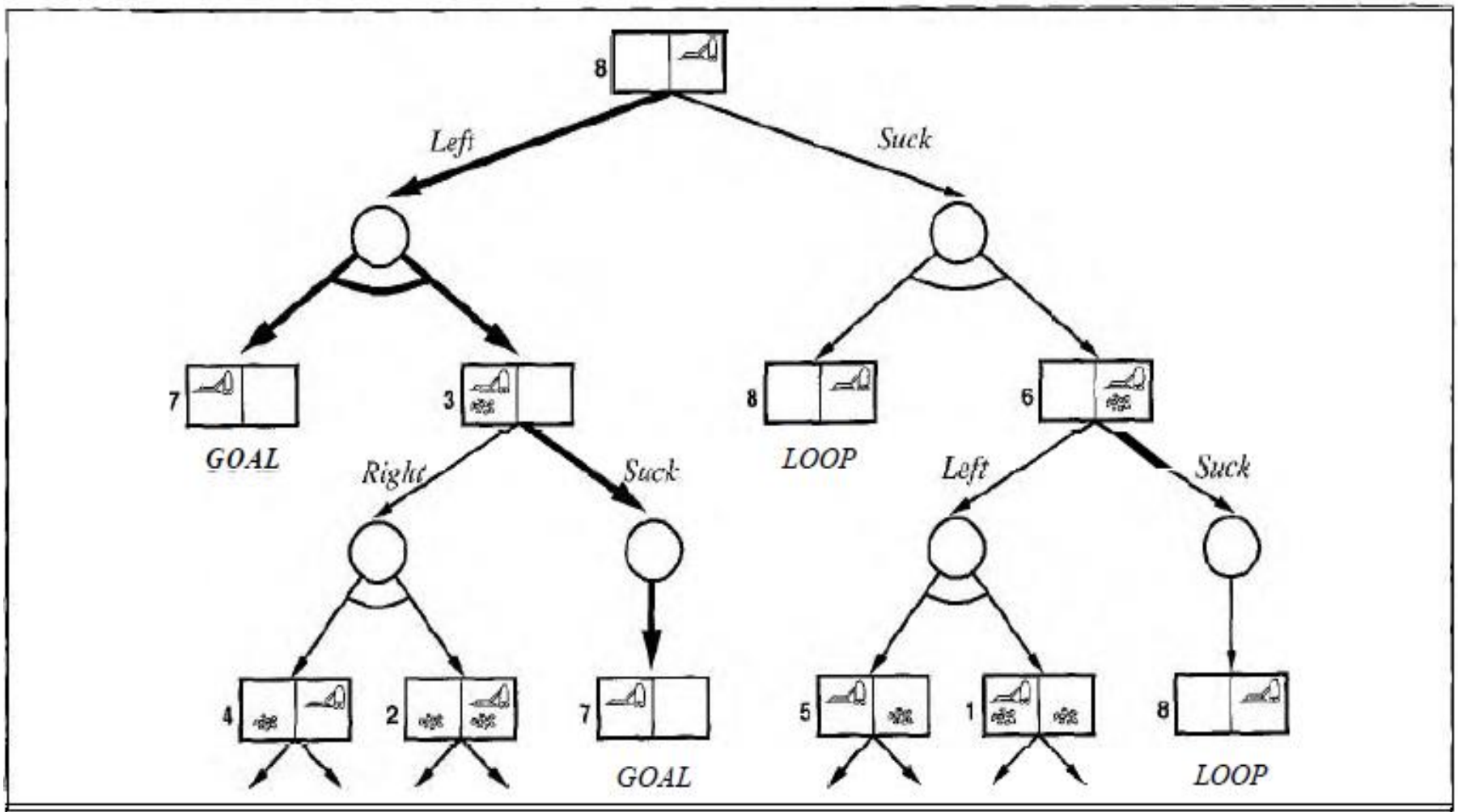


Figure 12.9 The first two levels of the search tree for the "double Murphy" vacuum world. State nodes are OR nodes where some action must be chosen. Chance nodes, shown as circles, are AND nodes where every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution is shown in bold lines.

Conditional planning in partially observable environments

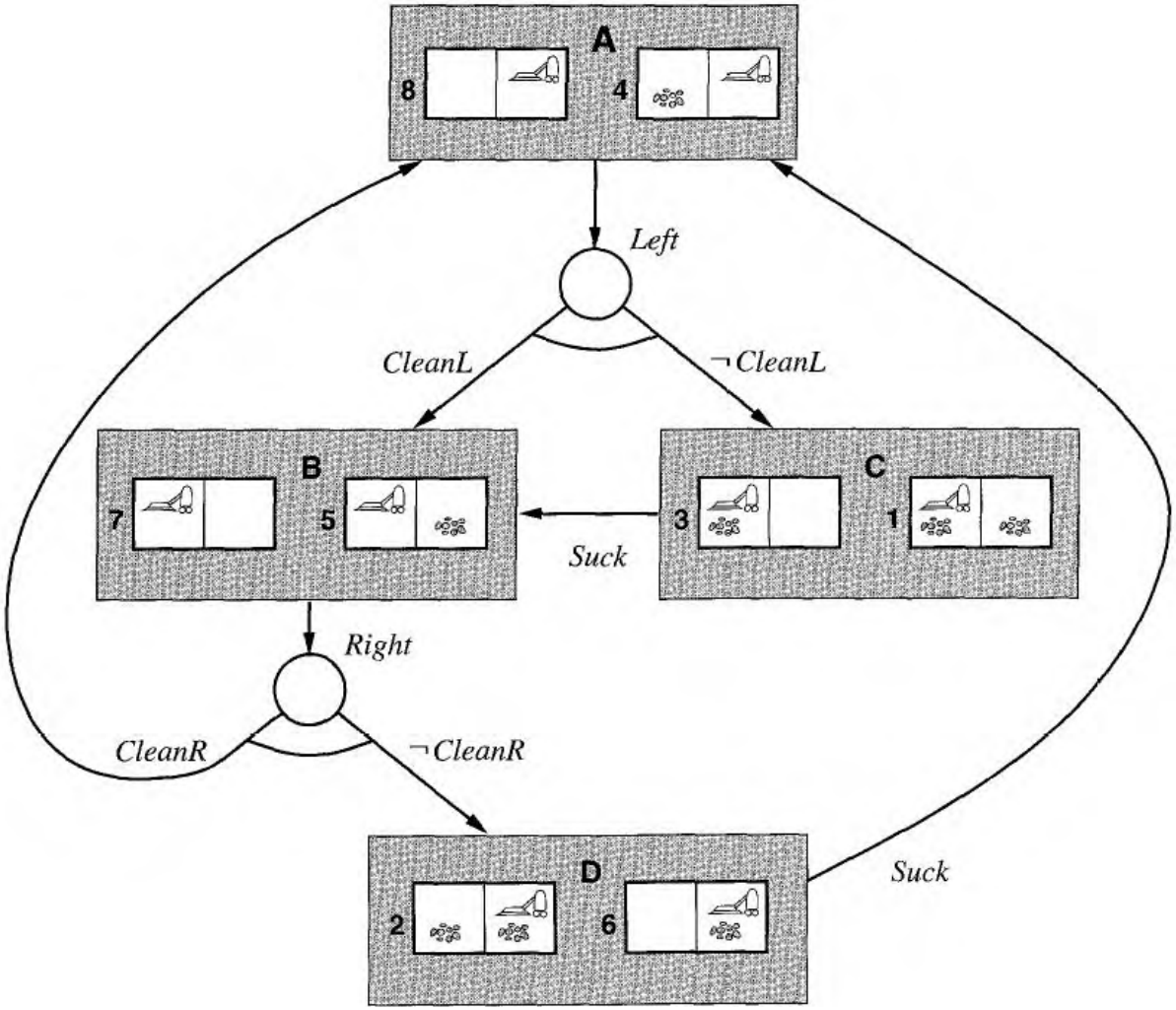
The preceding section dealt with fully observable environments, which have the advantage that condit.ional tests can ask any question at all and be sure of getting an answer. In the real world, partial observability is much more common

The simplest way to model this situation is to say that the initial state belongs to a **state set;** the state set is a way of describing the agent's initial **belief state.**

There are three basic choices for belief states:

1. Sets of full state descriptions. For example, the initial belief state in Figure 12..12 is

{ *(AtR A CleanR A CleanL), (AtR A CleanR A* ¬*CleanL)* } .



2. Logical sentences that capture exactly the set of possible worlds in the belief state. For example, the initial state can be written as

AtR A CleanR

3. **Knowledge propositions** describing the agent's kaowledge. Forthe initial state, we have

$$K(AtR) \wedge K(CleanR)$$

There are two choices here. We can have **automatic sensing,** which means that at every time step the agent gets all the available percepts

Alternatively, we can insist on **active sensing,** which means that percepts are obtained only SENSORY ACTIONS by executing specific **sensory actions** such as CheckDirt and ChleckLocation

To find out whether the square is dirty, the agent can *CheckDirt:*

$$Action(CheckDirt, \text{EFFECT:}\textbf{when}\ AtL \wedge CleanL: K(CleanL) \wedge$$
$$\textbf{when}\ AtL \wedge \neg CleanL: K(\neg CleanL) \wedge$$
$$\textbf{when}\ AtR \wedge CleanR: K(CleanR) \wedge$$
$$\textbf{when}\ AtR \wedge \neg CleanR: K(\neg CleanR)).$$