Knowledge Representation & Reasoning COMP9016

Dr Ruairí O’Reilly
ruairi.oreilly@cit.ie

Informed Search Strategies & Beyond

Classical Search Continued….

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Beyond Classical Search Continued….

>> Local Search in the state space: ✓

- simulated annealing ✓
- genetic algorithms

>> Relaxing the assumptions of determinism and observability:

- agent cannot predict percepts
- Contingency?
- Tracking potential states

>> Online search

Local beam search

>> The **local beam search algorithm** keeps track of *k* states rather than just one. It begins with *k* randomly generated states. At each step, all the successors of all *k* states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the *k* best successors from the complete list and repeats.

>> Similar to **random-restart** in parallel instead of in sequence? Two algorithms are different:

> *LBS* useful information is passed among the parallel search threads.

>> In its simplest form, local beam search can suffer from a lack of diversity among the *k* states.

> A variant called **stochastic beam search**, analogous to stochastic hill climbing, helps.

> Chooses *k* successors at random, with the probability of choosing a given successor being an increasing function of its value.

Genetic algorithms

>> A genetic algorithm (or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states rather than by modifying a single state.

>> Like beam searches, GAs begin with a set of *k* randomly generated states, called the population. Each state, or individual, is represented as a string over a finite alphabet—most commonly, a string of 0s and 1s.

>> For Example, 8-queens = $8 \times \log_2 8$ = 24 bits

Production of the next generation of states

>> Fitness function (objective function)

>> Selection

>> Crossover

>> Mutation

>> Advantages?

> Schema

> Instances

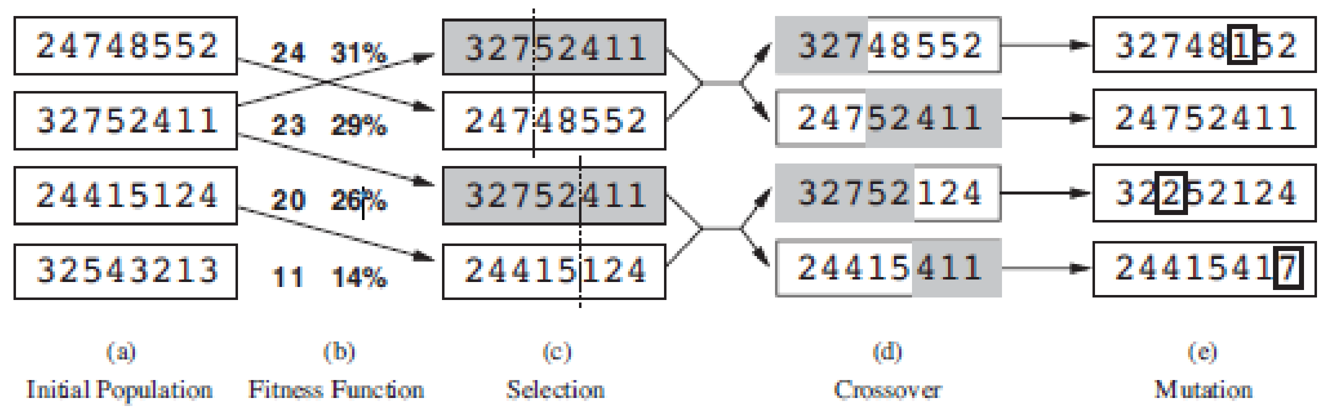


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

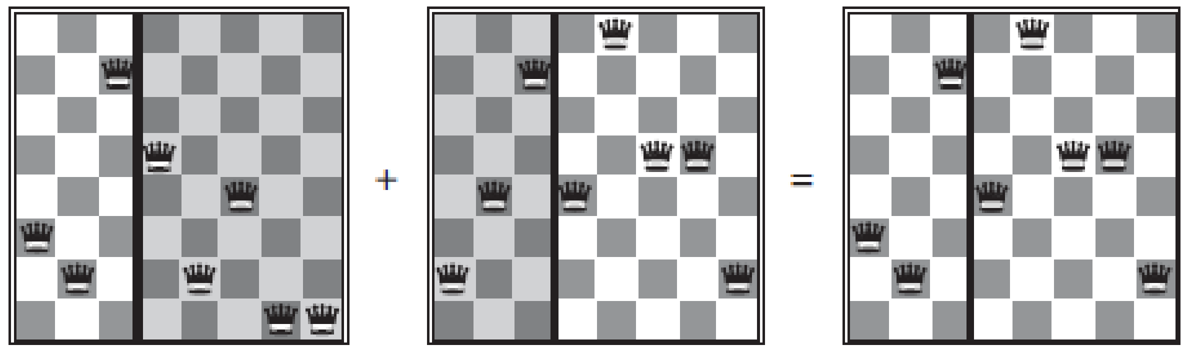
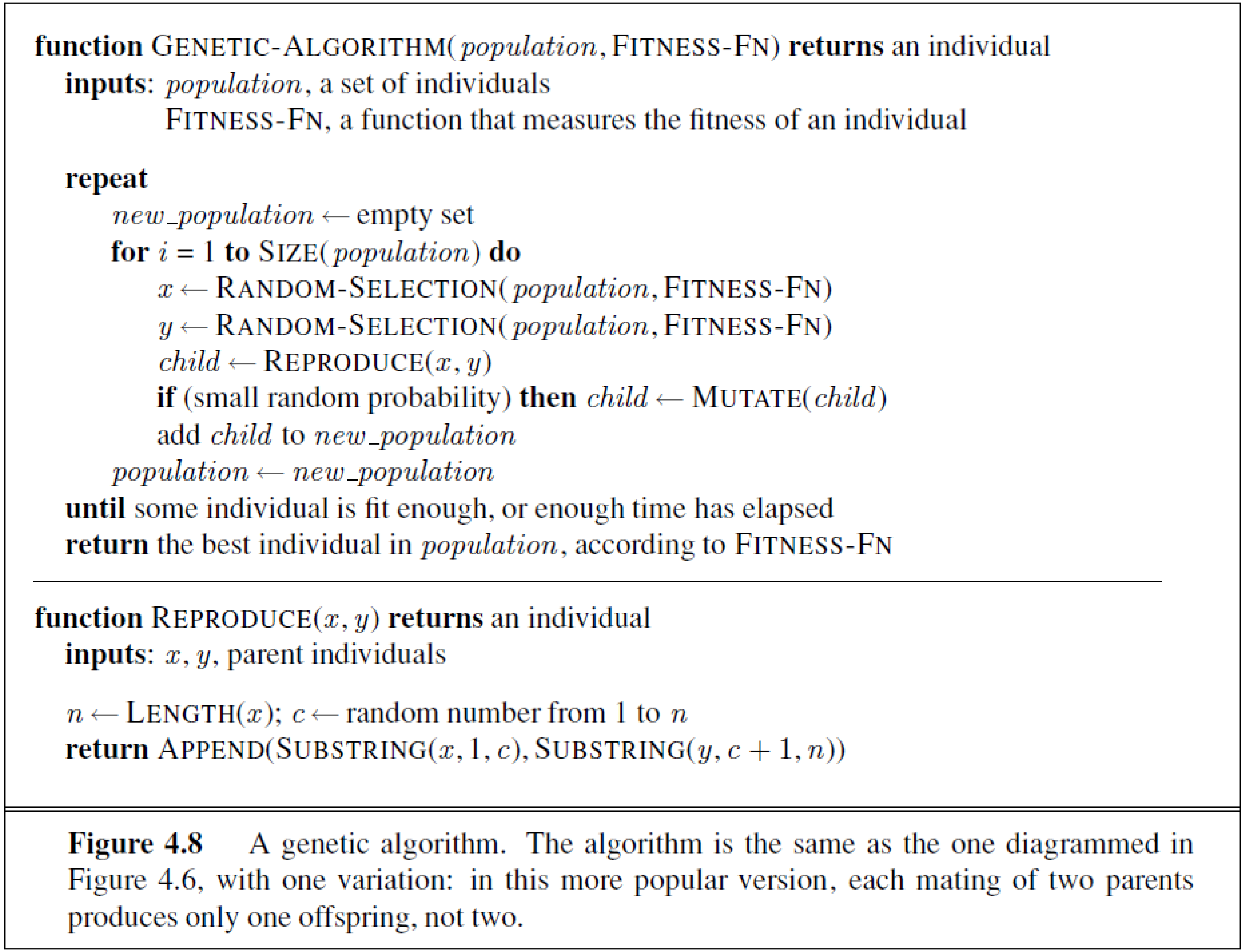


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.



```
function GENETIC-ALGORITHM($population$, FITNESS-FN) returns an individual
  inputs: $population$, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     $new\_population \leftarrow$ empty set
     for $i = 1$ to SIZE($population$) do
        $x \leftarrow$ RANDOM-SELECTION($population$, FITNESS-FN)
        $y \leftarrow$ RANDOM-SELECTION($population$, FITNESS-FN)
        $child \leftarrow$ REPRODUCE($x, y$)
        if (small random probability) then $child \leftarrow$ MUTATE($child$)
        add $child$ to $new\_population$
     $population \leftarrow new\_population$
  until some individual is fit enough, or enough time has elapsed
  return the best individual in $population$, according to FITNESS-FN
```

```
function REPRODUCE($x, y$) returns an individual
  inputs: $x, y$, parent individuals

  $n \leftarrow$ LENGTH($x$); $c \leftarrow$ random number from 1 to $n$
  return APPEND(SUBSTRING($x, 1, c$), SUBSTRING($y, c + 1, n$))
```

Figure 4.8 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.6, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

Local Search in continuous spaces

>> Discrete vs. continuous: The discrete/continuous distinction applies to the state of the environment, to the way time is handled, and to the percepts and actions of the agent.

>> For example, the chess environment has a finite number of distinct states (excluding the clock). Chess also has a discrete set of percepts and actions.

>> Taxi driving is a continuous-state and continuous-time problem: the speed and location of the taxi and of the other vehicles sweep through a range of continuous values and do so smoothly over time.

>>Taxi-driving actions are also continuous (steering angles, etc.). Input from digital cameras is discrete, strictly speaking, but is typically treated as representing continuously varying intensities and locations

An example

>> Suppose we want to place three new airports anywhere in Romania, such that the sum of squared distances from each city on the map to its nearest airport is minimized. The state space is then defined by the coordinates of the airports: ***(x1, y1), (x2, y2),*** and ***(x3, y3).*** This is a six-dimensional space; we also say that states are defined by six **variables**.

>> Moving around in this space corresponds to moving one or more of the airports on the map. The objective function ***f(x1, y1, x2, y2, x3, y3)*** is relatively easy to compute for any particular state once we compute the closest cities.

>> Let c_i be the set of cities whose closest airport (in the current state) is airport *i*. Then, in the neighborhood of the current state, where the c_is remain constant, we have

$$f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in c_i} (x_i - x_c)^2 + (y_i - y_c)^2$$

Discretization

>> One way to avoid continuous problems is simply to discretize the neighborhood of each state. For example, we can move only one airport at a time in either the x or y direction by a fixed amount $\pm\delta$. With 6 variables, this gives 12 possible successors for each state. We can then apply any of the local search algorithms described previously. We could also apply stochastic hill climbing and simulated annealing directly, without discretizing the space.

>> These algorithms choose successors randomly, which can be done by generating random vectors of length δ.

Searching with nondeterministic actions

>> Assumption to date with our search example - the environment is **fully observable** and **deterministic** and that the agent knows what the effects of each action are.

>> When the environment is either **partially observable** or **nondeterministic** (or both), percepts become useful.

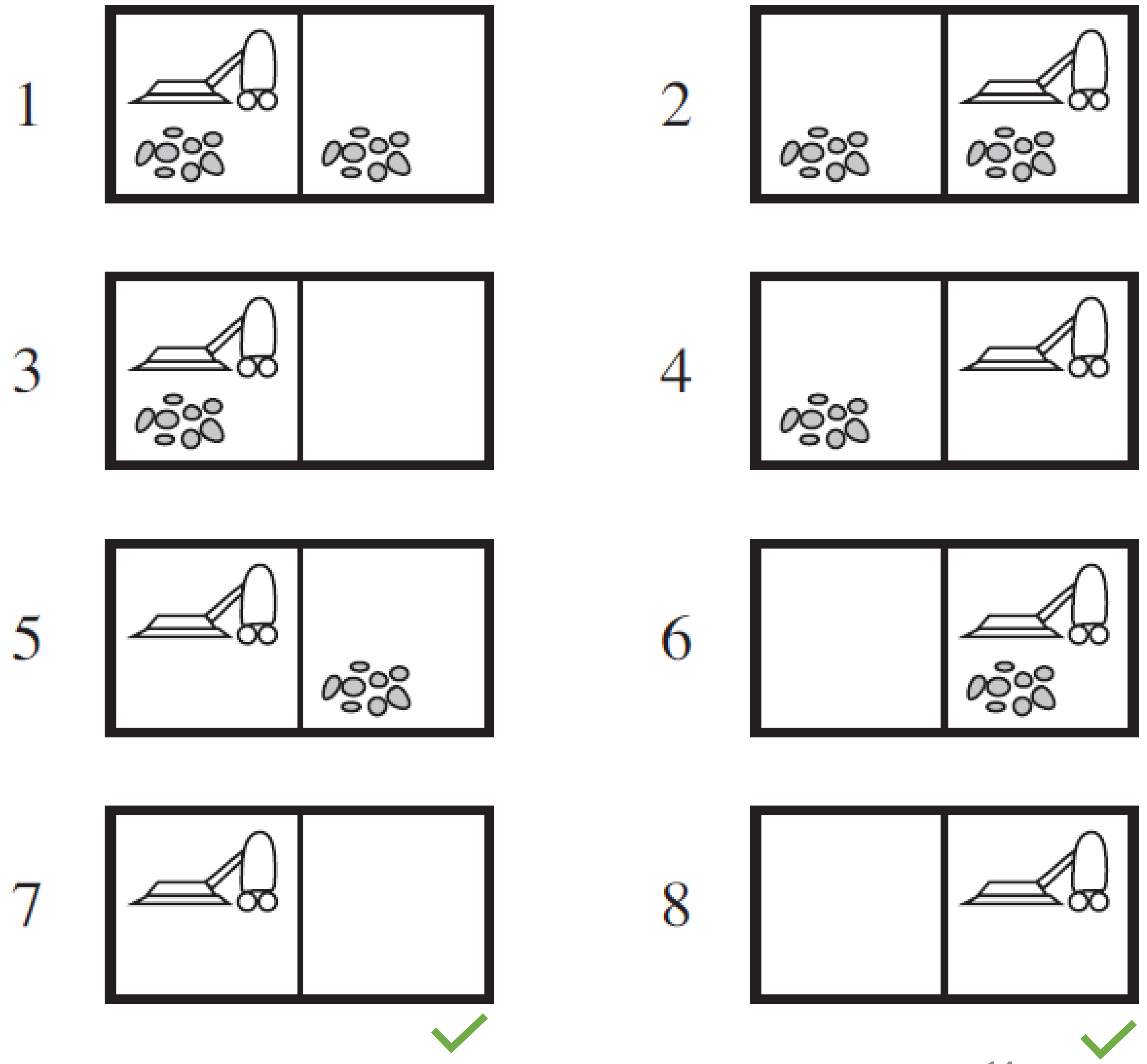
>> In both cases the solution to a problem is not a sequence but a **contingency plan** (also known as a **strategy**) that specifies what to do depending on what percepts are received.

The erratic vacuum world

>> Recall that the state space has eight states.

>> There are three actions—Left, Right, and Suck—and the goal is to clean up all the dirt (states 7 and 8).

>> If the environment is observable, deterministic and completely known, then the problem is trivially solvable by any of the uninformed or informed algorithms and the solution is an action sequence.



Introducing nondeterminsm

>> Suck action works as follows:

- When applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square, too.
- When applied to a clean square the action sometimes deposits dirt on the carpet.

>> To provide a precise formulation of this problem, we need to generalize the notion of a transition model. Instead of defining the transition model by a **RESULT** function that returns a single state, we use a **RESULTS** function that returns a set of possible outcome states.

>> We also need to generalize the notion of a **solution** to the problem. For example, if we start in state 1, there is no single sequence of actions that solves the problem. Instead, we need a contingency plan such as the following:

*[Suck, **if** State =5 **then** [Right, Suck] else [ ]]*

Solutions for nondeterministic problems can contain nested if–then–else statements; this means that they are trees rather than sequences

AND–OR search trees

>> The next question is how to find contingent solutions to nondeterministic problems. We begin by constructing **search trees**, but here the trees have a different character.

>> In a **deterministic environment**, the only branching is introduced by the agent's own choices in each state. We call these nodes **OR** nodes. In the vacuum world, for example, at an **OR** node the agent chooses *Left or Right or Suck.*

>> In a **nondeterministic environment**, branching is also introduced by the ***environment's*** choice of outcome for each action. We call these nodes **AND nodes**. For example, the Suck action in state 1 leads to a state in the set {5, 7}, so the agent would need to find a plan for state 5 and for state 7.

AND–OR search trees

>> A solution for an AND–OR search problem is a subtree that:

> Has a goal node at every leaf.

> Specifies one action at each of its OR nodes.

> Includes every outcome branch at each of its AND nodes.

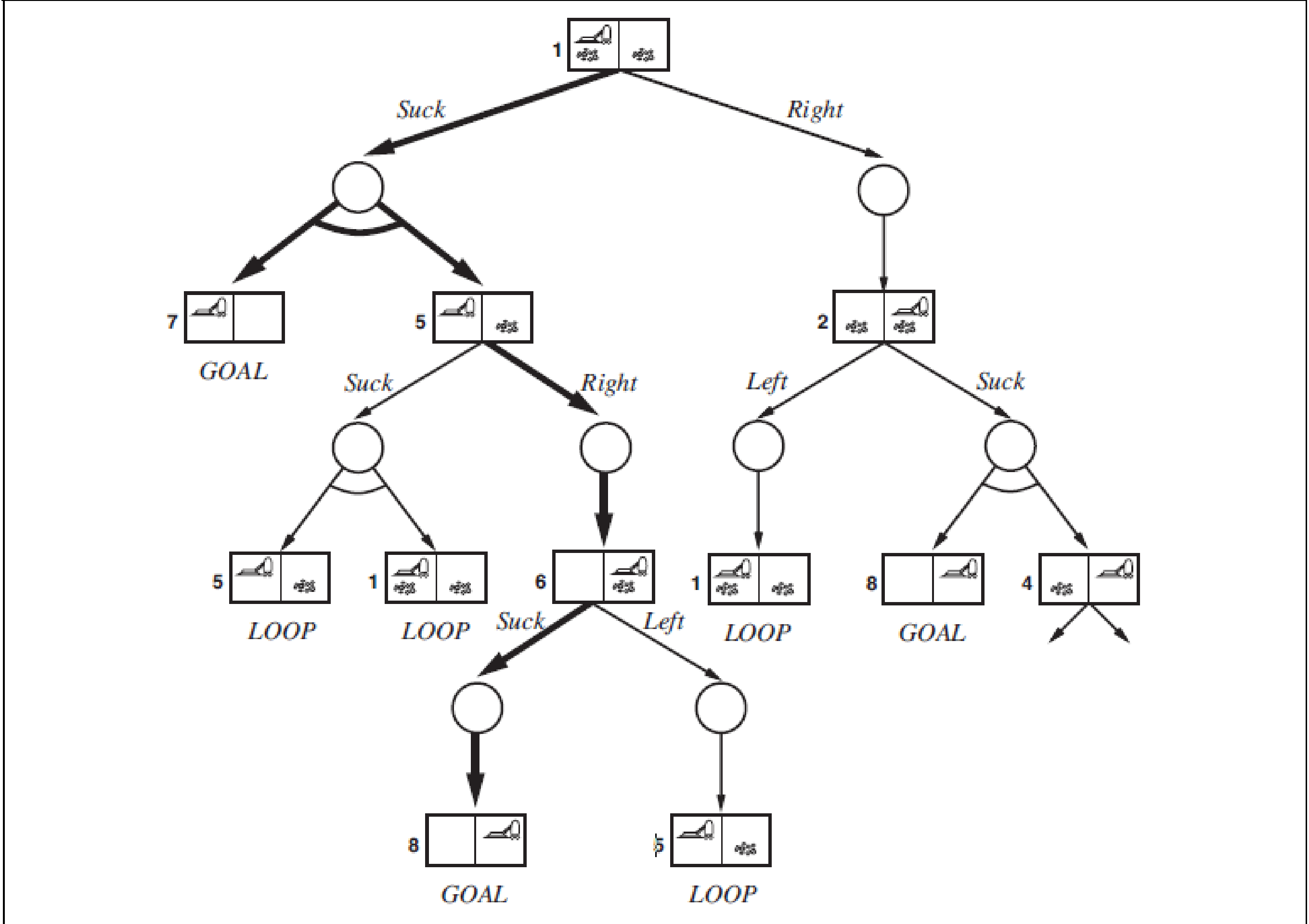
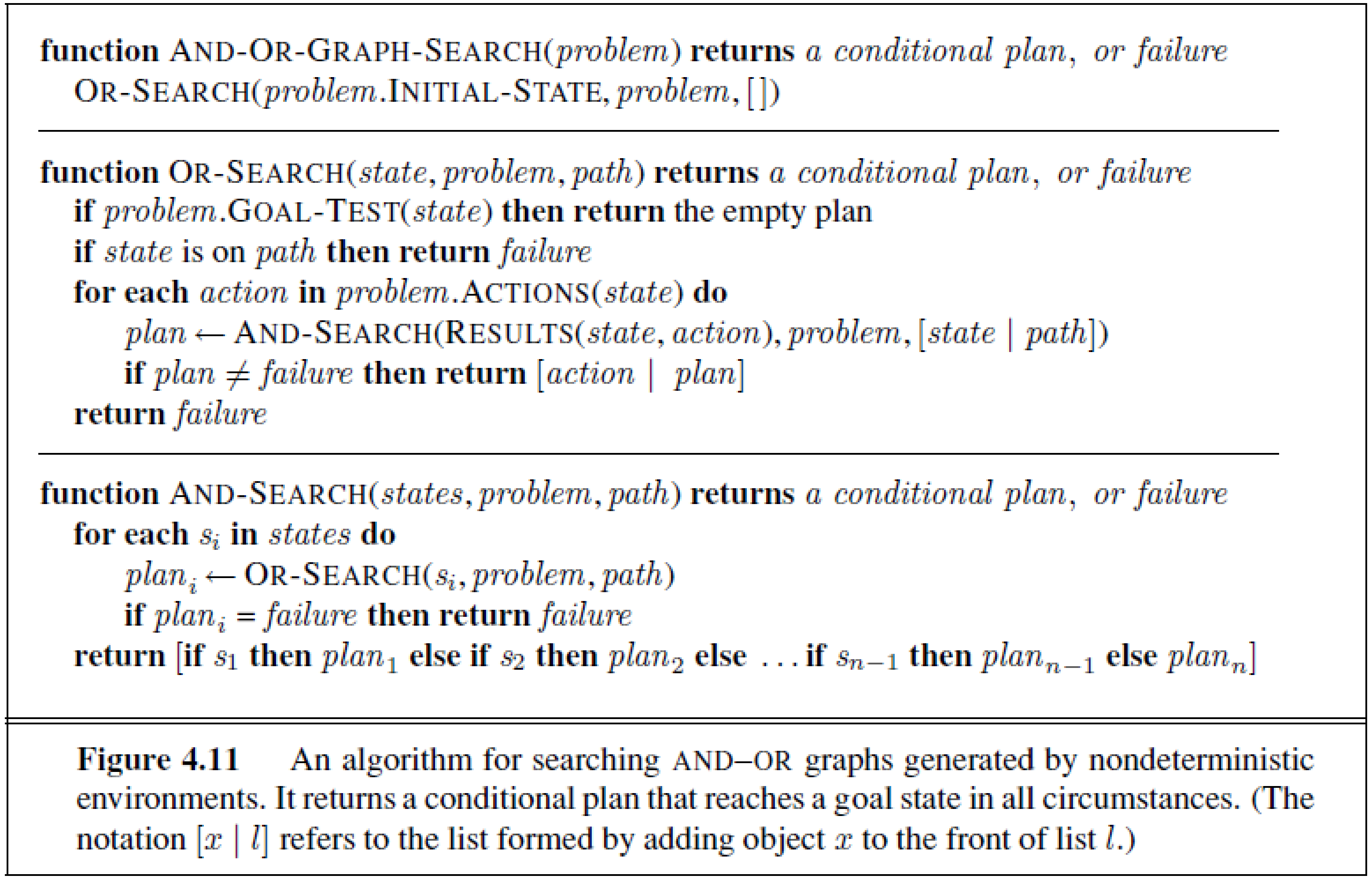


Figure 4.10 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

AND–OR search trees

>> A recursive, depth-first algorithm for AND–OR graph search.

>> Dealing with cycles?



```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(problem.INITIAL-STATE, problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if problem.GOAL-TEST(state) then return the empty plan
  if state is on path then return failure
  for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
      if plan ≠ failure then return [action | plan]
  return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
  for each $s_i$ in states do
      $plan_i$ ← OR-SEARCH($s_i$, problem, path)
      if $plan_i$ = failure then return failure
  return [if $s_1$ then $plan_1$ else if $s_2$ then $plan_2$ else ... if $s_{n-1}$ then $plan_{n-1}$ else $plan_n$]
```

Figure 4.11 An algorithm for searching AND–OR graphs generated by nondeterministic environments. It returns a conditional plan that reaches a goal state in all circumstances. (The notation $[x \mid l]$ refers to the list formed by adding object x to the front of list l.)

Slippery vacuum world

>> Movement actions sometimes fail, leaving the agent in the same location.

>> Moving “*Right*” in state 1

>> Thus, our cyclic solution is

[Suck, L1 : Right , if State =5 then L1 else Suck]

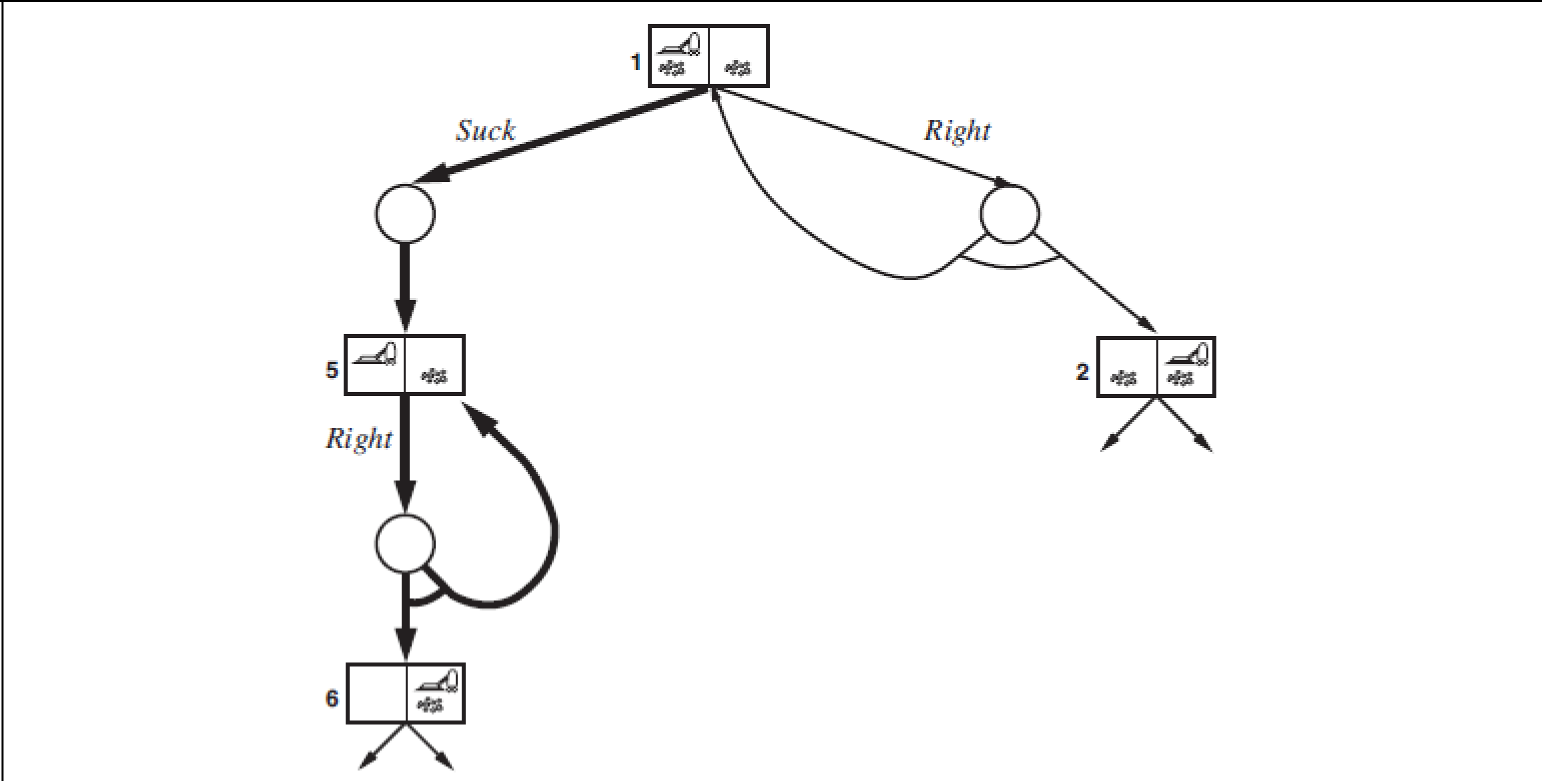


Figure 4.12 Part of the search graph for the slippery vacuum world, where we have shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably.

Searching with partial observations

>> Partial observability, where the agent’s percepts do not suffice to pin down the exact state.

>> If the agent is in one of several possible states, then an action may lead to one of several possible outcomes —*even if the environment is deterministic.*

>> The key concept required for solving partially observable problems is the **belief state**, representing the agent’s current belief about the possible physical states it might be in, given the sequence of actions and percepts up to that point.

>> The simplest scenario for studying belief states, which is when the agent has no sensors at all; then we add in partial sensing as well as nondeterministic actions.

Searching with no observation

>> When the agent’s percepts provide no information at all, we have what is called a **sensorless** problem or sometimes a **conformant** problem.

>> A sensorless vacuum world.

Initial state could be any element of the set {1, 2, 3, 4, 5, 6, 7, 8}. What happens if action Right?

states {2, 4, 6, 8}—the agent now has more information

[Right,Suck] will always end up in one of the states {4, 8}.

Finally, the sequence [Right,Suck,Left,Suck] is guaranteed to reach the goal state 7 no matter what the start state.

>> We say that the agent can **coerce** the world into state 7

Searching with no observation

>> To solve sensorless problems, we search in the space of belief states rather than physical states.

>> Suppose the underlying physical problem *P* is defined by $ACTIONS_P$, $RESULT_P$, $GOAL{-}TEST_P$, $STEP{-}COST_P$. Then we can define the corresponding sensorless problem as follows:

Belief states: The entire belief-state space contains every possible set of physical states. If *P* has *N* states, then the sensorless problem has up to *2^N* states, although many may be unreachable from the initial state.

Initial state: Typically the set of all states in *P*, although in some cases the agent will have more knowledge than this.

Actions: This is slightly tricky. Suppose the agent is in belief state *b={s1, s2}*, but $ACTIONS_P(s1)$!= $ACTIONS_P(s2)$; then the agent is unsure of which actions are legal. If we assume that illegal actions have no effect on the environment, then it is safe to take the union of all the actions in any of the physical states in the current belief state b: $ACTIONS(b) = \bigcup_{s \in b} ACTIONS_P(s)$

Searching with no observation – Transition model

The agent doesn't know which state in the belief state is the right one; so as far as it knows, it might get to any of the states resulting from applying the action to one of the physical states in the belief state. For deterministic actions, the set of states that might be reached is

$$b' = RESULT(b, a) = \{s' : s' = RESULT_P(s, a) \text{ and } s \in b\}$$

With deterministic actions, b' is never larger than b. With nondeterminism, we have

$$b' = RESULT(b, a) = \{s' : s' \in RESULT_P(s, a) \text{ and } s \in b\}$$
$$= \bigcup_{s \in b} RESULT_P(s, a)$$

which may be larger than *b*, as shown in Figure 4.13. The process of generating the new belief state after the action is called the **prediction** step; the notation b' = $PREDICT_P(b,a)$ will come in handy.

Searching with no observation – Transition model

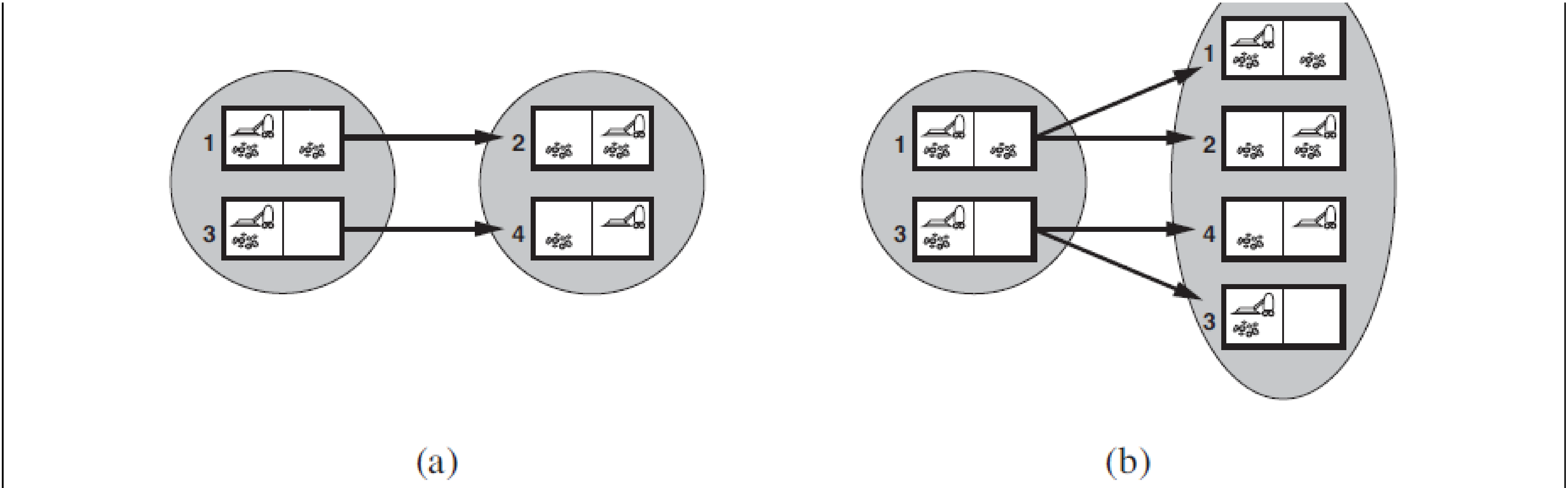


Figure 4.13 (a) Predicting the next belief state for the sensorless vacuum world with a deterministic action, *Right*. (b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

Searching with no observation

- Goal test: The agent wants a plan that is sure to work, which means that a belief state satisfies the goal only if *all* the physical states in it satisfy $GOAL{-}TEST_P$. The agent may *accidentally* achieve the goal earlier, but it won’t *know* that it has done so.

- Path cost: This is also tricky. If the same action can have different costs in different states, then the cost of taking an action in a given belief state could be one of several values. For now we assume that the cost of an action is the same in all states and so can be transferred directly from the underlying physical problem.

>> The reachable belief-state space for the deterministic, sensorless vacuum world. There are only 12 reachable belief states out of 2^8 =256 possible belief states.

>> In “ordinary” graph search, newly generated states are tested to see if they are identical to existing states.

>> This works for belief states, too:

> Sequence [Suck,Left,Suck] reaches the same belief state as [Right,Left,Suck]

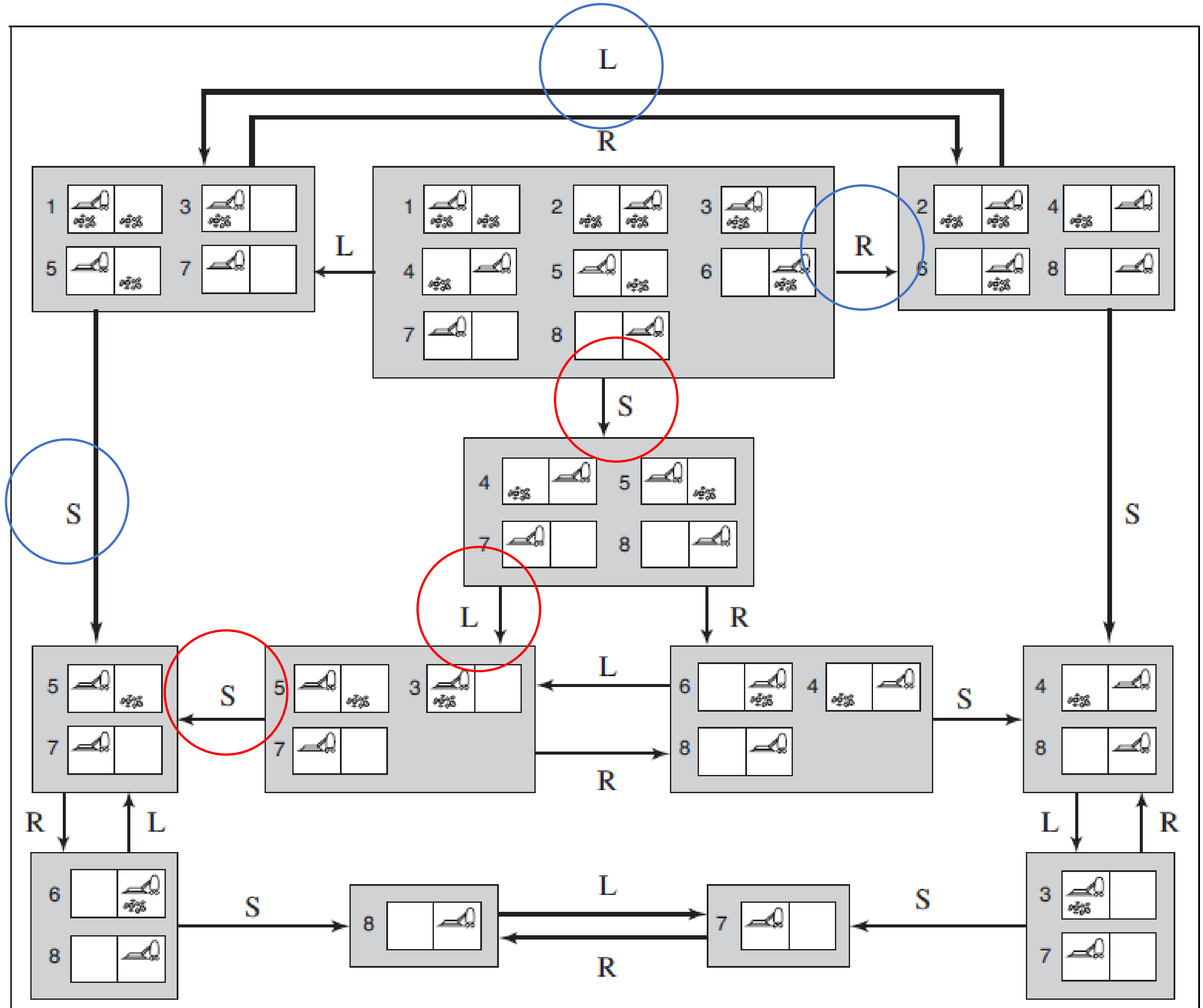


Figure 4.14 The reachable portion of the belief-state space for the deterministic, sensorless vacuum world. Each shaded box corresponds to a single belief state. At any given point, the agent is in a particular belief state but does not know which physical state it is in. The initial belief state (complete ignorance) is the top center box. Actions are represented by labeled links. Self-loops are omitted for clarity.

Searching with observations

>> For a general partially observable problem, we have to specify how the environment generates percepts for the agent. E.g. Vac-world agent has position and local dirt sensor but no sensor capable of detecting dirt in other squares.

>> *PERCEPT(s)* function returns the percept received in a given state.

> For example, in the local-sensing vacuum world, the *PERCEPT* in state 1 is *[A, Dirty]*.

> Fully observable problems are a special case in which *PERCEPT(s)=s* for every state *s*, while sensorless problems are a special case in which PERCEPT(s)=null.

>> When observations are partial, it will usually be the case that several states could have produced any given percept. For example, the percept *[A, Dirty]* is produced by state 3 as well as by state 1. Hence, given this as the initial percept, the initial belief state for the local-sensing vacuum world will be {1, 3}.

Searching with observations

>> The ACTIONS, STEP-COST, and GOAL-TEST are constructed from the underlying physical problem just as for sensorless problems, but the transition model is a bit more complicated.

>> We can think of transitions from one belief state to the next for a particular action as occurring in three stages:

• The **prediction** stage is the same as for sensorless problems given the action *a* in belief state *b*, the predicted belief state is $\hat{b}$=*PREDICT(b, a)*
• The **observation prediction** stage determines the set of percepts *o* that could be observed in the predicted belief state: *POSSIBLE-PERCEPTS*($\hat{b}$) = {*o* : *o*=*PERCEPT(s)* and *s* $\in \hat{b}$} .
• The **update** stage determines, for each possible percept, the belief state that would result from the percept. The new belief state b_o is just the set of states in $\hat{b}$ that could have produced the percept: b_o = *UPDATE*($\hat{b}$, *o*) = {*s* : *o*= *PERCEPT(s)* and *s* $\in \hat{b}$} .

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Putting these three stages together, we obtain the possible belief states resulting from a given action and the subsequent possible percepts:
in the predicted belief state

RESULTS(b, a) = {b_o : b_o =
UPDATE(PREDICT(b, a), o) and
o ∈ POSSIBLE-PERCEPTS(PREDICT(b, a))}

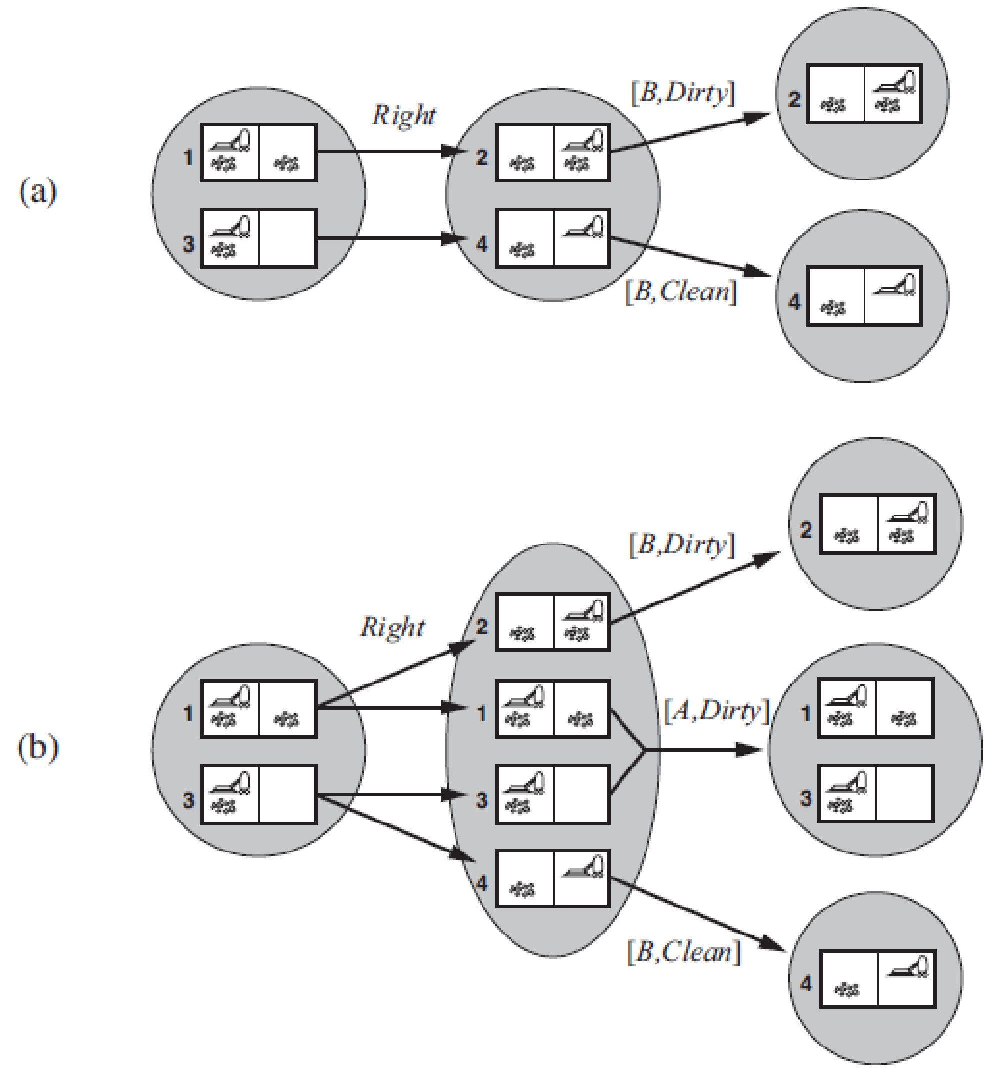


Figure 4.15 Two example of transitions in local-sensing vacuum worlds. (a) In the deterministic world, *Right* is applied in the initial belief state, resulting in a new belief state with two possible physical states; for those states, the possible percepts are $[B, \mathit{Dirty}]$ and $[B, \mathit{Clean}]$, leading to two belief states, each of which is a singleton. (b) In the slippery world, *Right* is applied in the initial belief state, giving a new belief state with four physical states; for those states, the possible percepts are $[A, \mathit{Dirty}]$, $[B, \mathit{Dirty}]$, and $[B, \mathit{Clean}]$, leading to three belief states as shown.

Solving partially observable problems

>> We have shown how to derive the *RESULTS* function for a nondeterministic belief-state problem from an underlying physical problem and the *PERCEPT* function. Given such a formulation, the *AND–OR* search algorithm can be applied directly to derive a solution.

> [A, Dirty].

> *[Suck, Right, **if** Bstate ={6} **then** Suck **else** [ ]] .*

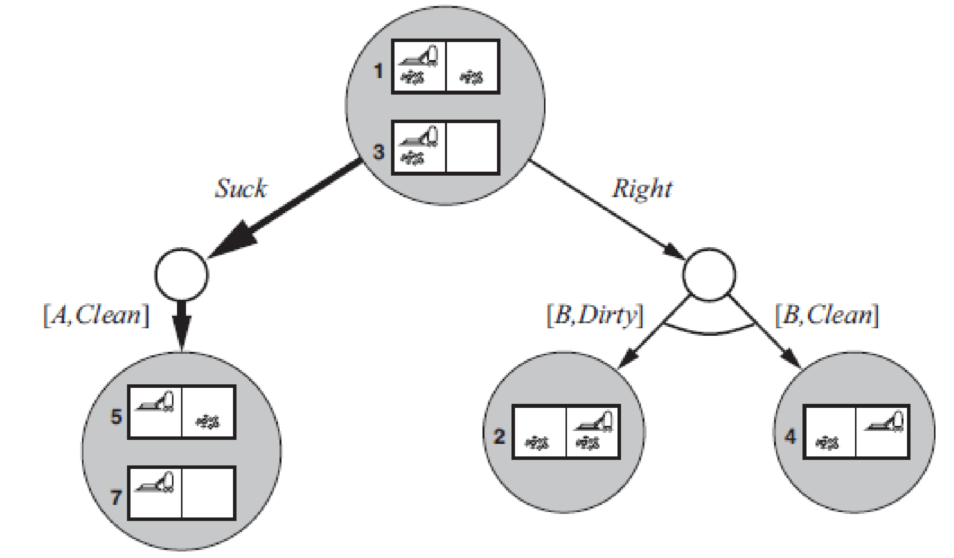


Figure 4.16 The first level of the AND–OR search tree for a problem in the local-sensing vacuum world; *Suck* is the first step of the solution.

An agent for partially observable environments

>> Similar to the simple problem-solving agent: the agent formulates a problem, calls a search algorithm (such as AND-OR-GRAPH-SEARCH) to solve it, and executes the solution.

>> Two main differences:

> The solution to a problem will be a conditional plan rather than a sequence.

> The agent will need to maintain its belief state as it performs actions and receives percepts.

>> Given an initial belief state *b*, an action *a*, and a percept *o*, the new belief state is:

$b' = UPDATE(PREDICT(b, a), o)$

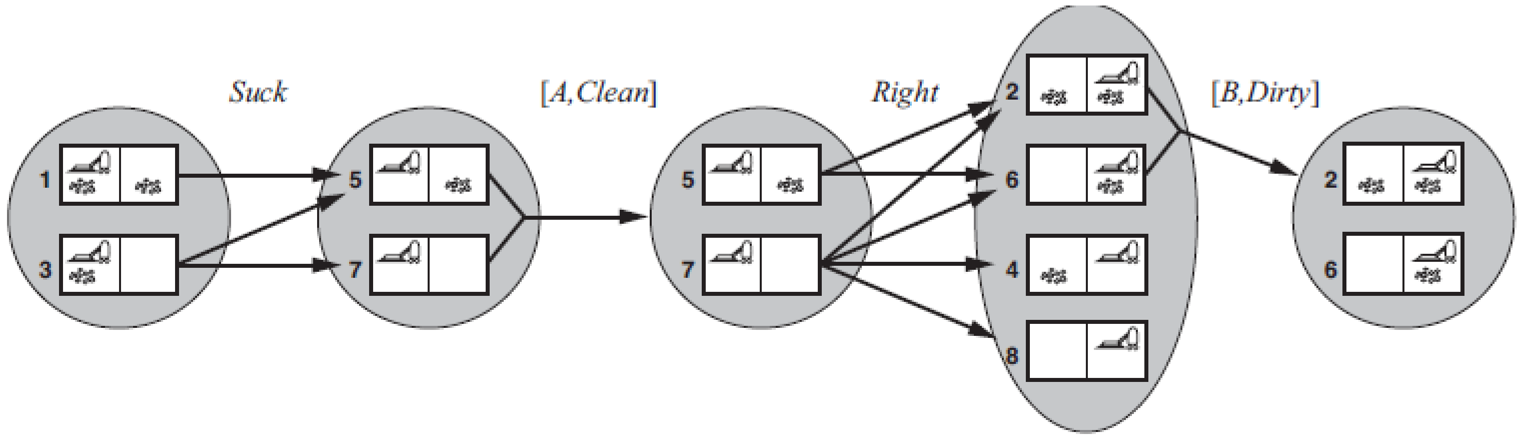


Figure 4.17 Two prediction–update cycles of belief-state maintenance in the kindergarten vacuum world with local sensing.

An agent for partially observable environments

>> In partially observable environments—which include the vast majority of real-world environments—maintaining one’s belief state is a core function of any intelligent system.

>> This function goes under various names, including **monitoring**, **filtering** and **state estimation**. The equation on the previous slide is called a **recursive** state estimator because it computes the new belief state from the previous one rather than by examining the entire percept sequence.

>> Most work on this problem has been done for stochastic, continuous-state environments with the tools of probability theory, as we will discuss in probalistic reasoning over time. Here we will show an example in a discrete environment with deterministic sensors and nondeterministic actions

Discrete environment with deterministic sensors and nondeterministic actions

>> The example concerns a robot with the task of localization.

>> Our robot is placed in the maze-like environment; the robot is equipped with four sonar sensors that tell whether there is an obstacle in each of the four compass directions

>> Assume that the sensors give perfectly correct data, and that the robot has a correct map of the environment. Unfortunately the robot's navigational system is broken, so when it executes a Move action, it moves randomly to one of the adjacent squares. The robot's task is to determine its current location

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Our robot

>> Initial belief state *b* – set of all locations.

>> *PERCEPT – NSW*

> b_o *=UPDATE(b)*

>> *EXECUTE Move* action

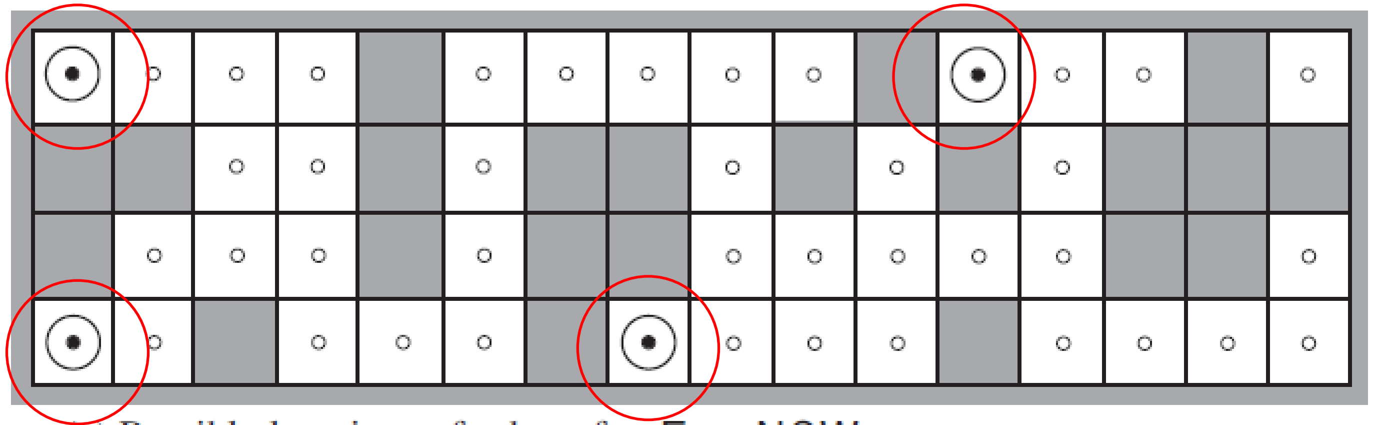
> New belief state

b_a *=PREDICT(*b_o*, Move)*

>> *PERCEPT – NS*

>> *UPDATE(*b_a*, NS)*

>> *UPDATE(PREDICT(UPDATE(b,NSW),Move),NS) .*


(a) Possible locations of robot after $E_1 = NSW$

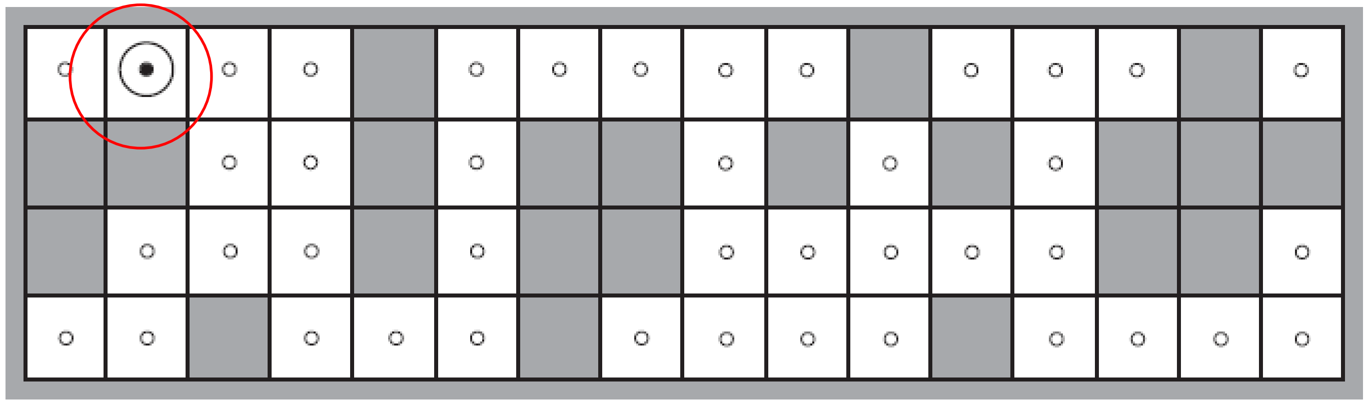

(b) Possible locations of robot After $E_1 = NSW, E_2 = NS$

Figure 4.18 Possible positions of the robot, $\odot$, (a) after one observation $E_1 = NSW$ and (b) after a second observation $E_2 = NS$. When sensors are noiseless and the transition model is accurate, there are no other possible locations for the robot consistent with this sequence of two observations.

ONLINE SEARCH AGENTS AND UNKNOWN ENVIRONMENTS

>> So far we have concentrated on agents that use **offline search** algorithms. They compute a complete solution before setting foot in the real world and then execute the solution.

>> In contrast, an **online search** agent **interleaves** computation and action: first it takes an action, then it observes the environment and computes the next action. Online search is a good idea in dynamic or semi dynamic domains—domains where there is a penalty for sitting around and computing too long.

>> Online search is a necessary idea for unknown environments, where the agent does not know what states exist or what its actions do. In this state of ignorance, the agent faces an **exploration problem** and must use its actions as experiments in order to learn enough to make deliberation worthwhile.

Online search problems

>> An online search problem must be solved by an agent executing actions, rather than by pure computation. We assume a deterministic and fully observable environment, but we stipulate that the agent knows only the following:

• *ACTIONS(s)*, which returns a list of actions allowed in state s;
• The step-cost function *c(s, a, s')* —note that this cannot be used until the agent knows that s is the outcome; and
• *GOAL-TEST(s)*.

>> Note in particular that the agent cannot determine *RESULT(s, a)* except by actually being in *s* and doing *a*.
E.g. Maze example

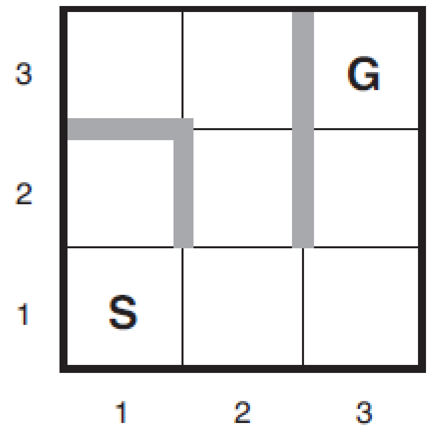


Figure 4.19 A simple maze problem. The agent starts at S and must reach G but knows nothing of the environment.

Online search problems

>> Finally, the agent might have access to an admissible heuristic function h(s) that estimates the distance from the current state to a goal state. For example, in Figure 4.19, the agent might know the location of the goal and be able to use theManhattan-distance heuristic.

>> Typically the agent’s objective is to reach a goal state while minimizing cost.

> *Total path cost as compared to the path cost – **competitive ratio***

> The best achievable competitive ratio is infinite in some cases?

> **irreversibility** and **dead-ends**

Online search problems

>> *no algorithm can avoid dead ends in all state spaces*

>> An **adversary argument**

>> Assumption – **safely explorable**

>> Describing performance?

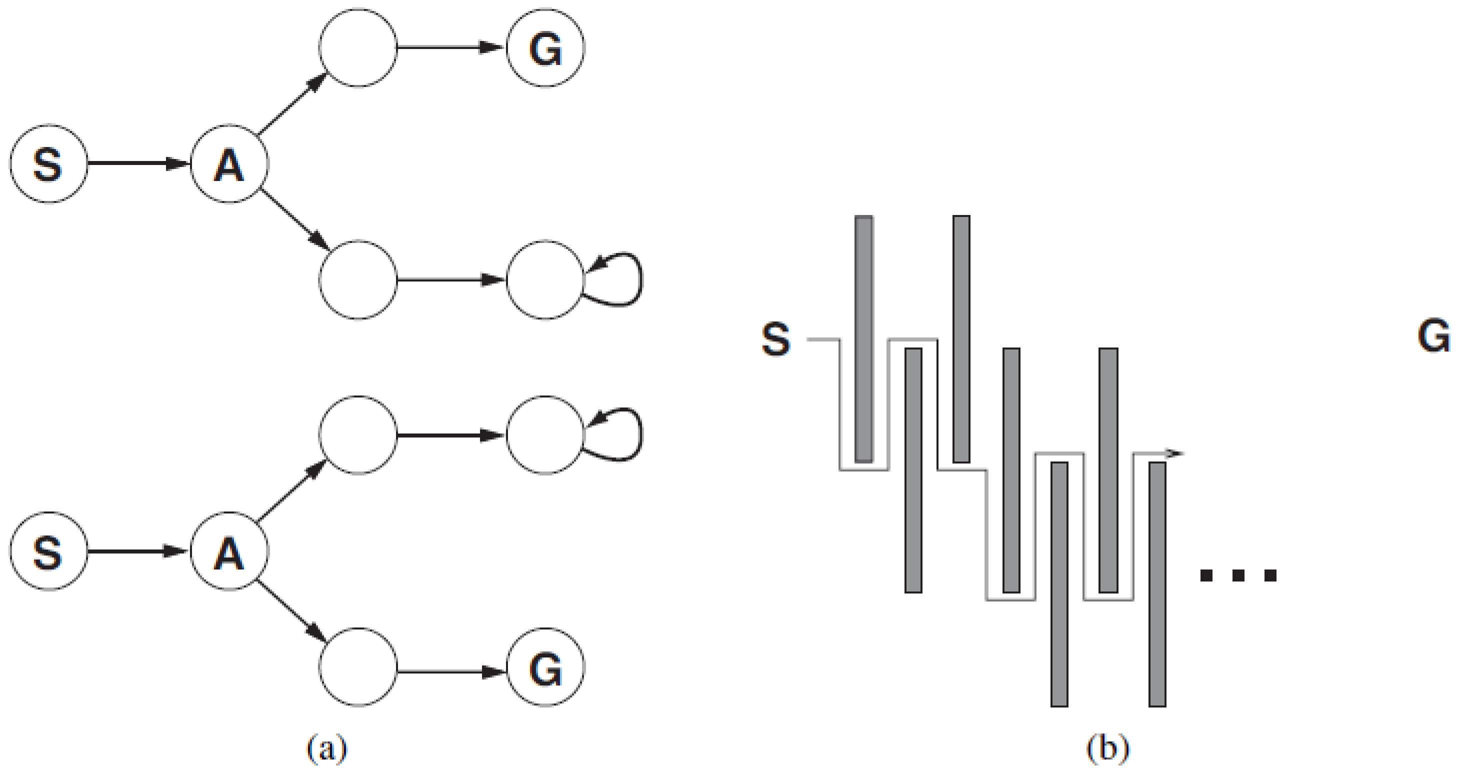


Figure 4.20 (a) Two state spaces that might lead an online search agent into a dead end. Any given agent will fail in at least one of these spaces. (b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent makes, the adversary blocks that route with another long, thin wall, so that the path followed is much longer than the best possible path.

TO BE COMPLETED….

References

[1] Russell, S. and Norvig, P., 2002. Artificial intelligence: a modern approach Beyond Classical Search, Chapter 4.

Summary