Course: Introduction to AI

Prof. Ahmed Guessoum
The National Higher School of AI

Chapter 4

Beyond Classical Search

Outline

- Local Search Algorithms and Optimization Problems
  - Hill-climbing search
  - Simulated annealing
  - Local beam search
  - Genetic algorithms
- Searching with Nondeterministic Actions
  - The erratic vacuum world
  - AND–OR search trees
- Searching with Partial Observations
  - Searching with no observation
  - Searching with observations

Local Search Algorithms & Optimization Problems

- Search algorithms seen so far are designed to explore search spaces systematically.
  - keeping one or more paths in memory and
  - recording which alternatives have been explored at each point along the path.
- A path to the goal is a solution.
- In various applications, the path is not relevant, just the goal state. E.g. 8-Queens problem, integrated-circuit design, etc.
- ➔ we will consider a different class of algorithms; not care about path to goal.

Local Search

- **Local search** algorithms operate using a single **current node** and generally move only to neighbors of that node
- Two key advantages:
  1. they use very little memory;
  2. they often find reasonable solutions in <u>large or infinite (continuous) state spaces</u> for which systematic algorithms are unsuitable.
- Local search algorithms are useful for solving pure **optimization problems**: they find the best state according to an **objective function**.

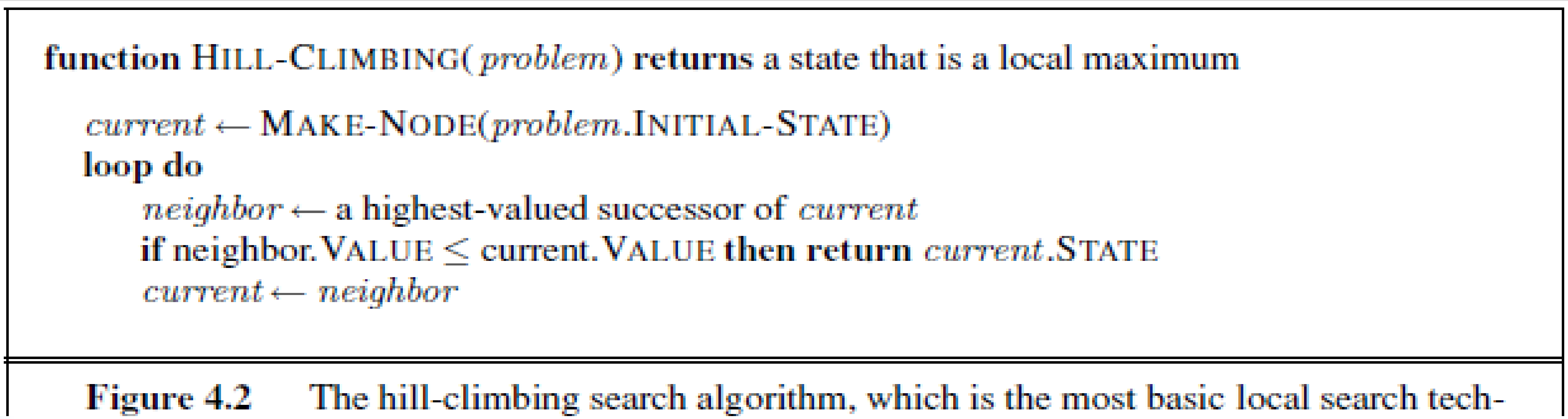
State-space landscape

- To understand local search, it is useful to consider the **state-space landscape.**
- A landscape has
  - A “location”, defined by the state; and
  - “elevation”, defined by the value of the heuristic cost function or objective function.
  - If cost function used, **aim** is **global min**
  - If objective function used, **aim** is **global max**
- A **complete** local search algorithm always finds a goal if one exists;
- An **optimal** algorithm always finds a global min/max.

Hill-climbing search

- The **hill-climbing** search algorithm (**steepest-ascent** version):
  - continually moves in the direction of increasing value (best successor)—i.e. uphill;
  - if more than one best successor, selects one randomly
  - terminates when it reaches a "peak" where no neighbour has a higher value.
- Hill-climbing does not maintain a search tree;
- The data structure for the current node need only record the state and the value of the objective function.
- Hill-climbing only considers the immediate neighbours for next move.

Hill-climbing search



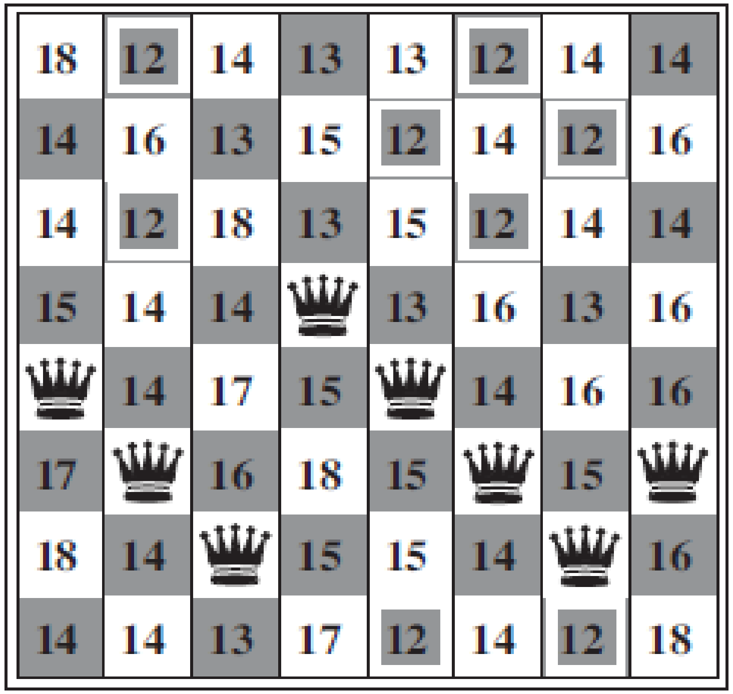
```
function HILL-CLIMBING(problem) returns a state that is a local maximum

  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
      neighbor ← a highest-valued successor of current
      if neighbor.VALUE ≤ current.VALUE then return current.STATE
      current ← neighbor
```

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search tech-

- Example: 8-Queens problem
  - <u>Successors of a state</u>: all states generated by moving a single queen to another square in the same column

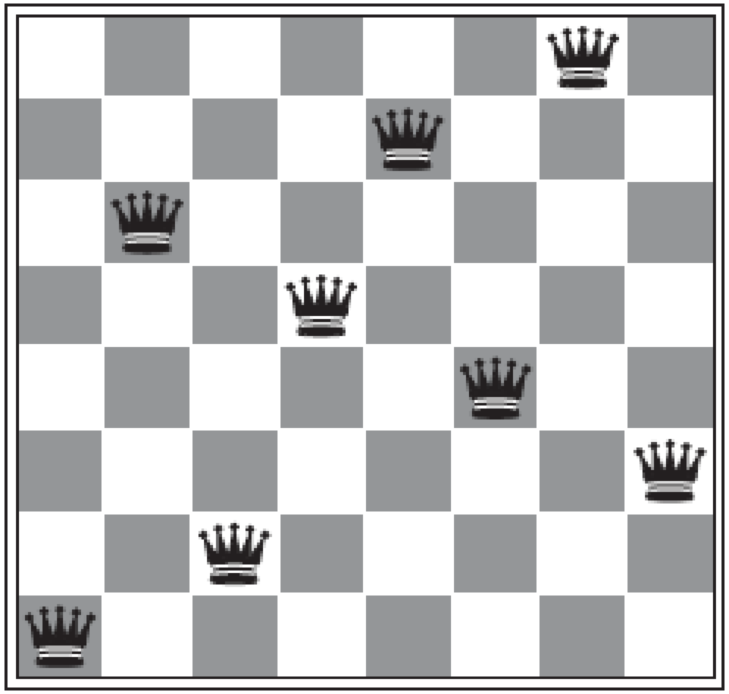
  ➔ 8×7=56 successors.
  - <u>Heuristic cost function *h*</u>: number of pairs of queens that are attacking each other, either directly or indirectly.
  - <u>Global minimum</u>: *h=0*

8-Queens Problem



18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18

(a)

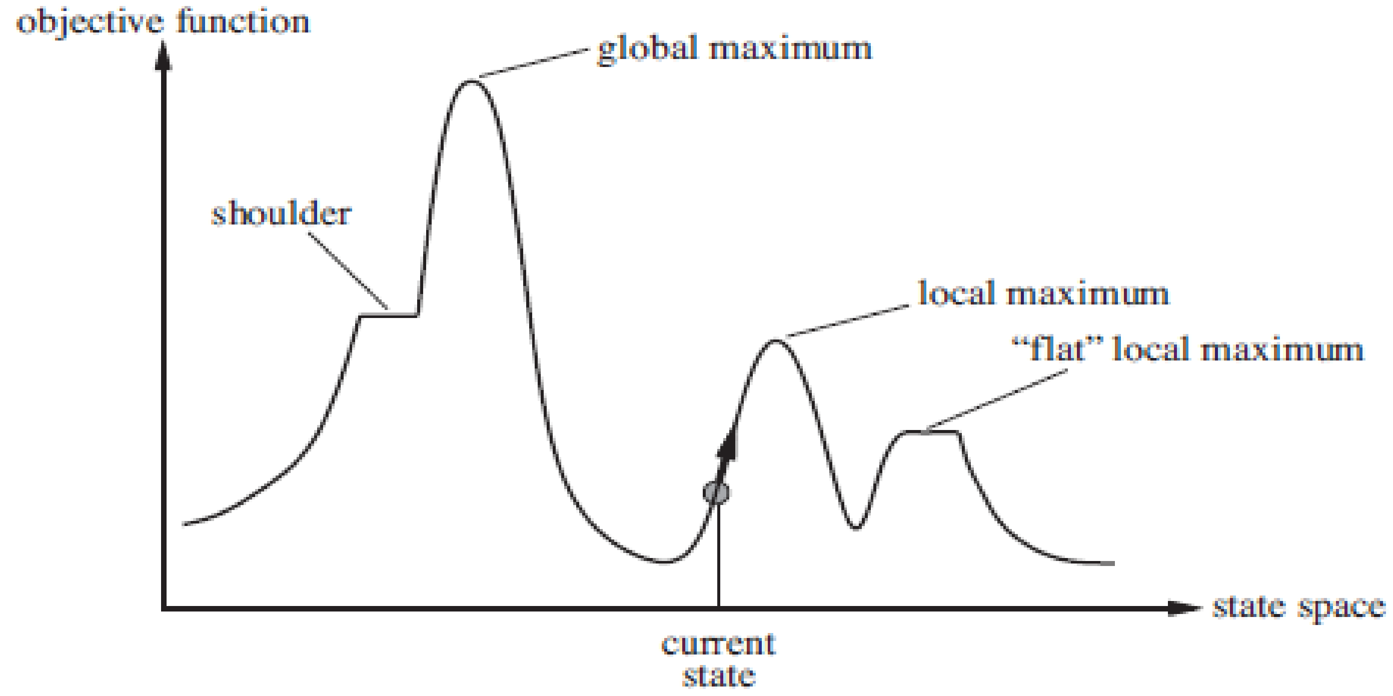


(b)

Figure 4.3 (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.

Hill-Climbing

- HC is sometimes called **greedy local search** because it looks for immediate improvement, without thinking ahead about where to go next.
- HC often makes rapid progress toward a solution because it is usually quite easy to improve a bad state. E.g. in Figure, HC goes from h=17 to h=1 in 5 moves.
- HC often gets stuck because of **Local maxima, Ridges**, or **Plateaux**.
- In each case, the algorithm reaches a point at which no progress is being made.



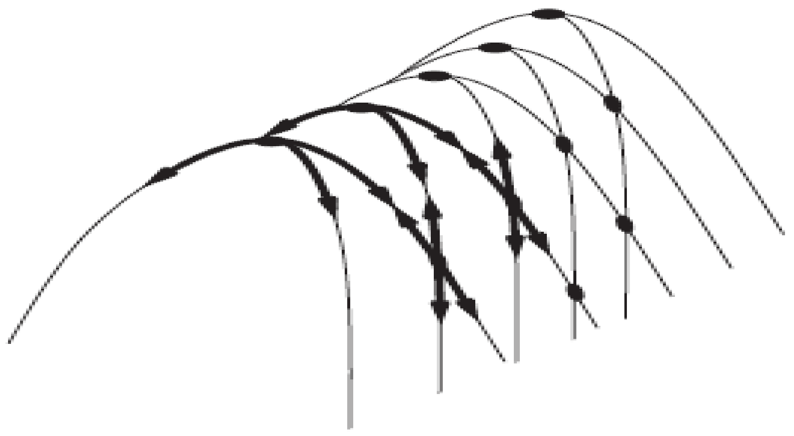


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

Hill-Climbing

- E.g. from a randomly generated 8-queens state, steepest-ascent HC
  - gets stuck 86% of the time, solving only 14% of problem instances.
  - It works quickly,
    - taking just 4 steps on average when it succeeds
    - and 3 steps when it gets stuck
  - Not bad for a state space with $8^8 \approx$ 17 million states
- HC will get stuck at a plateau (best successor with same value as current state).
- HC Is **incomplete** (because of local maxima!)

Hill-Climbing

- **Possibility**: to allow a **sideways move**, in case the plateau is in fact a shoulder.
- BUT **risk** of falling in an infinite loop in case of a real plateau.
- **Solution?** limit on the number of consecutive sideways moves allowed. E.g. 100 for 8-Queens problem.
  - Raises the percentage of problem instances solved by hill climbing from 14% to 94%.
  - BUT: the algorithm averages ≈ 21 steps for each successful instance and 64 for each failure.

Variants of HC

- Many variants of HC have been invented.
    - **Stochastic HC:** chooses at random from among the uphill moves;
        - usually converges more slowly than steepest ascent
        - in some state landscapes, it finds better solutions.
    - **First-choice HC:** implements stochastic HC by generating successors randomly until one is generated that is better than the current state.
        - a good strategy when a state has many (e.g., thousands of) successors.
    - **Random-Restart HC**: conducts a series of HC searches from randomly generated initial states, until a goal is found.
        - If each HC search has probability *p* of success, then the expected number of restarts required is *1/p*.

Variants of HC

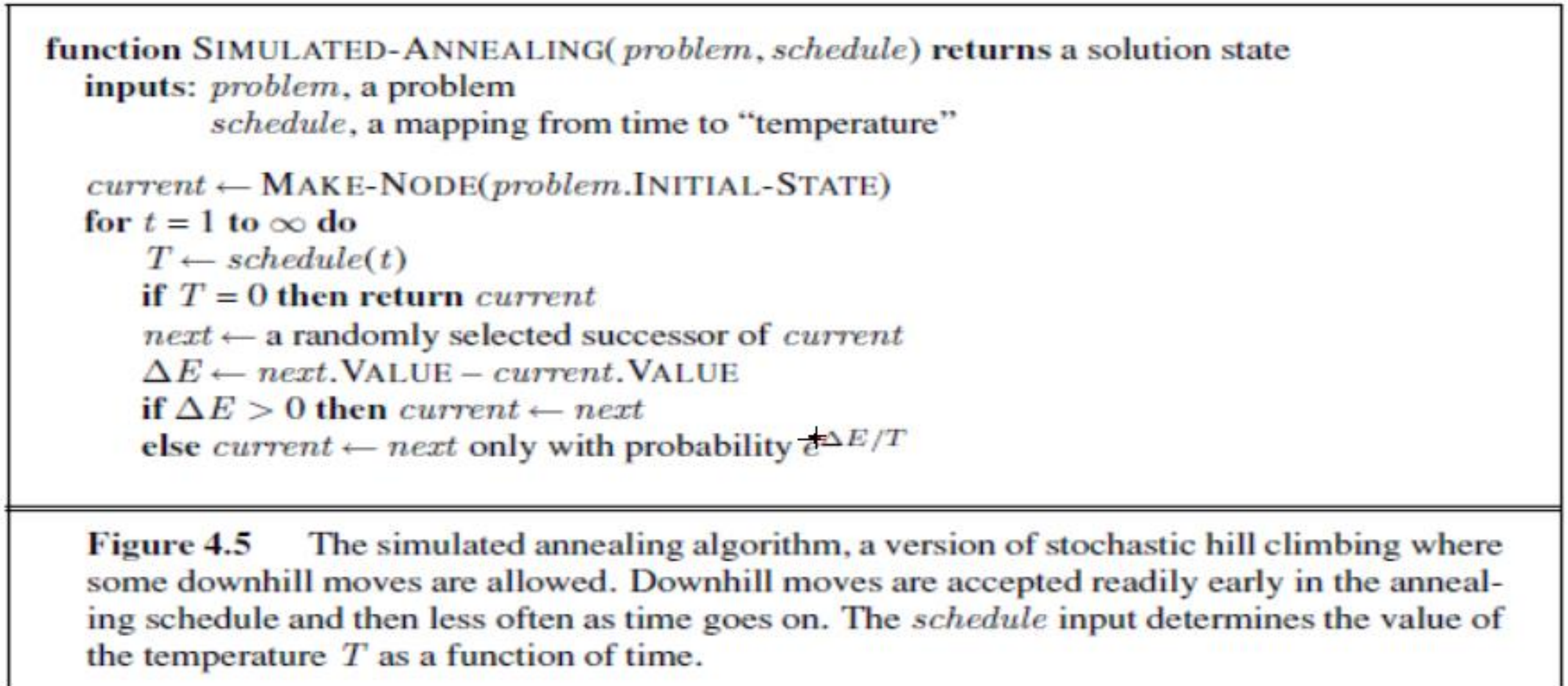
- For 8-queens instances:
  - With no sideways moves allowed,
    - p ≈0.14 (≈7 iterations) to find a goal (6 fails; 1 success).
    - Expected number of steps is the cost of one successful iteration plus *(1−p)/p* times the cost of failure, i.e. ≈22 steps.
  - With sideways moves allowed, average of 1/0.94 ≈ 1.06 iterations needed and ≈ 25 steps
- **Random-Restart HC** is very effective! Even for 3 million queens, it can find solutions in under a minute.
- Success of HC depends a lot on the shape of the state-space landscape: if there are few local maxima and plateaux, RRHC will find a good solution very quickly.
- Many real problems are NP-hard, with an exponential number of local maxima!

Simulated annealing

- HC that does not go downhill is incomplete.
- Purely random walk (successor chosen uniformly at random from the set of successors) is complete but extremely inefficient
- **Simulated annealing:** combines HC with a random walk so as to yield both efficiency and completeness.
- Consider minimizing the cost (instead of "climbing a hill").
- Imagine trying to get a ball down into the deepest crevice in a bumpy surface.
  - Letting the ball roll down, it will stop at a local minimum;
  - Shaking the surface, the ball can be bounced out of the local minimum.
  - **Trick:** shake surface just hard enough to bounce the ball out of local minima but not hard enough to dislodge it from the global minimum.

Simulated annealing

- **Annealing:** process in metallurgy used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low energy crystalline state.
- **Simulated-annealing solution**: start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"

   current ← MAKE-NODE(problem.INITIAL-STATE)
   for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← next.VALUE − current.VALUE
      if ΔE > 0 then current ← next
      else current ← next only with probability $e^{\Delta E/T}$
```

Figure 4.5 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The *schedule* input determines the value of the temperature *T* as a function of time.

- Innermost loop of the **simulated-annealing algorithm** is quite similar to HC:
  - However, instead of picking the *best* move, it picks a *random* move.
  - If the move improves the situation, it is always accepted.
  - Otherwise, the algorithm accepts the move with some probability less than 1.
  - The probability decreases exponentially with the "badness" of the move—the amount ΔE by which the evaluation is worsened.
  - The probability also decreases as the "temperature" T goes down: "bad" moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases.
- If the schedule lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

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.
- Is this equivalent to running *k* random restarts in parallel?
- **NO!** In a random-restart search, each search process runs independently of the others.
- *In a local beam search, useful information is passed among the parallel search threads.*

Local beam search

- Local beam search can suffer from a lack of diversity among the *k* states, which can quickly become concentrated in a small region of the state space.
- **Stochastic beam search (SBS)**:
  - Analogous to stochastic Hill Climbing.
  - Instead of choosing the best *k* from the pool of candidate successors, SBS chooses *k* successors at random, with the probability of choosing a given successor being an increasing function of the value of this successor.

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.
- E.g. an 8-Queens problem state must specify the positions of 8 queens in each column:
  - Using bits as $8\times \log_2 8 = 24$ bits, or
  - Using 8 digits, each in the range from 1 to 8.

Genetic algorithms

- (In GA terminology) the **fitness function** (objective function) is a function that should return higher values for better states.
- E.g. a fitness function for 8-Queens: “the number of *non-attacking* pairs of queens”, which has a value of 28 for a solution.
- From a given **population** (generation), individuals or pairs thereof are selected (with some probabilities) for operations to be applied on them:
  - Crossover
  - Mutation
  - Selection
- The resulting individuals *with highest fitness* become the new population (next generation)

Genetic Operations

- **Crossover**: For each pair to be mated:
  - a **crossover** point is chosen randomly from the positions in the string. (In next example, after 3rd digit in 1st pair and after 5th digit in 2nd pair.)
  - the offsprings themselves are generated by <u>crossing</u> over parent strings at the crossover point. (The parents are taken from the population with some probabilities).
- Each location is subject to random **mutation** with a small independent probability.
- A **selection** operation is applied with some probability to select

Genetic Algorithm on 8-Queens

Here, the probability of being chosen for reproducing is taken as directly proportional to the fitness score. (One individual is selected twice; one not at all.)

The percentages are shown next to the raw fitness scores.

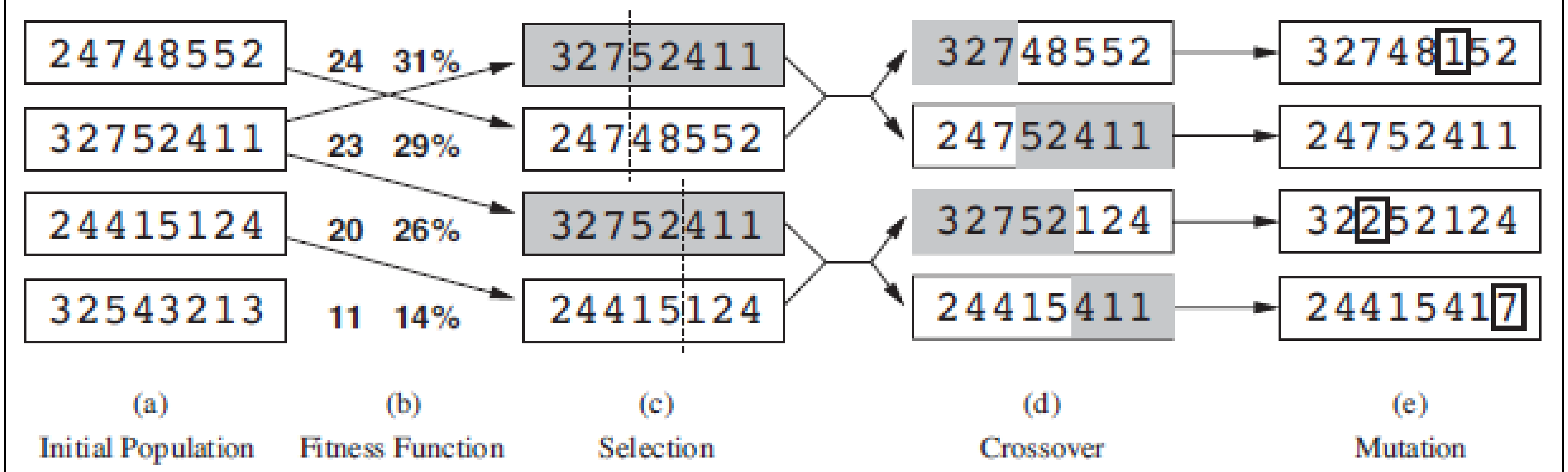


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Crossover on 8-Queens states

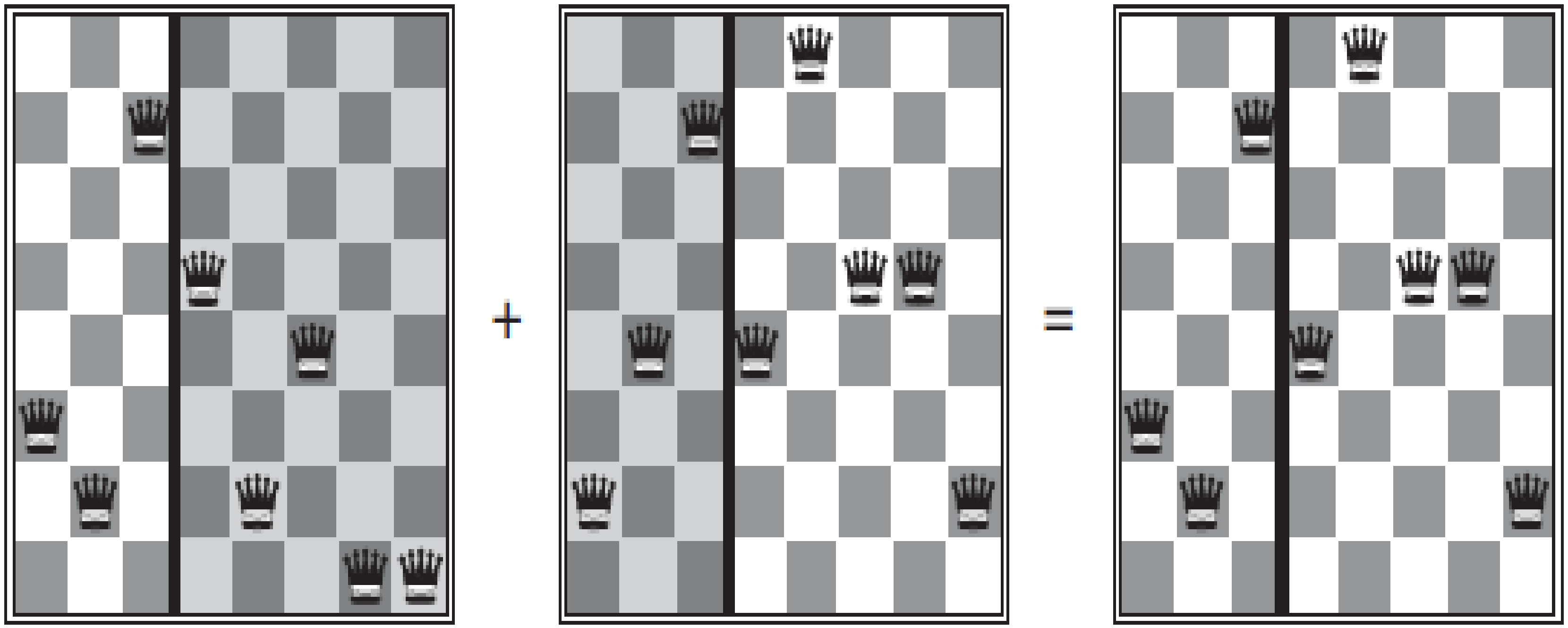
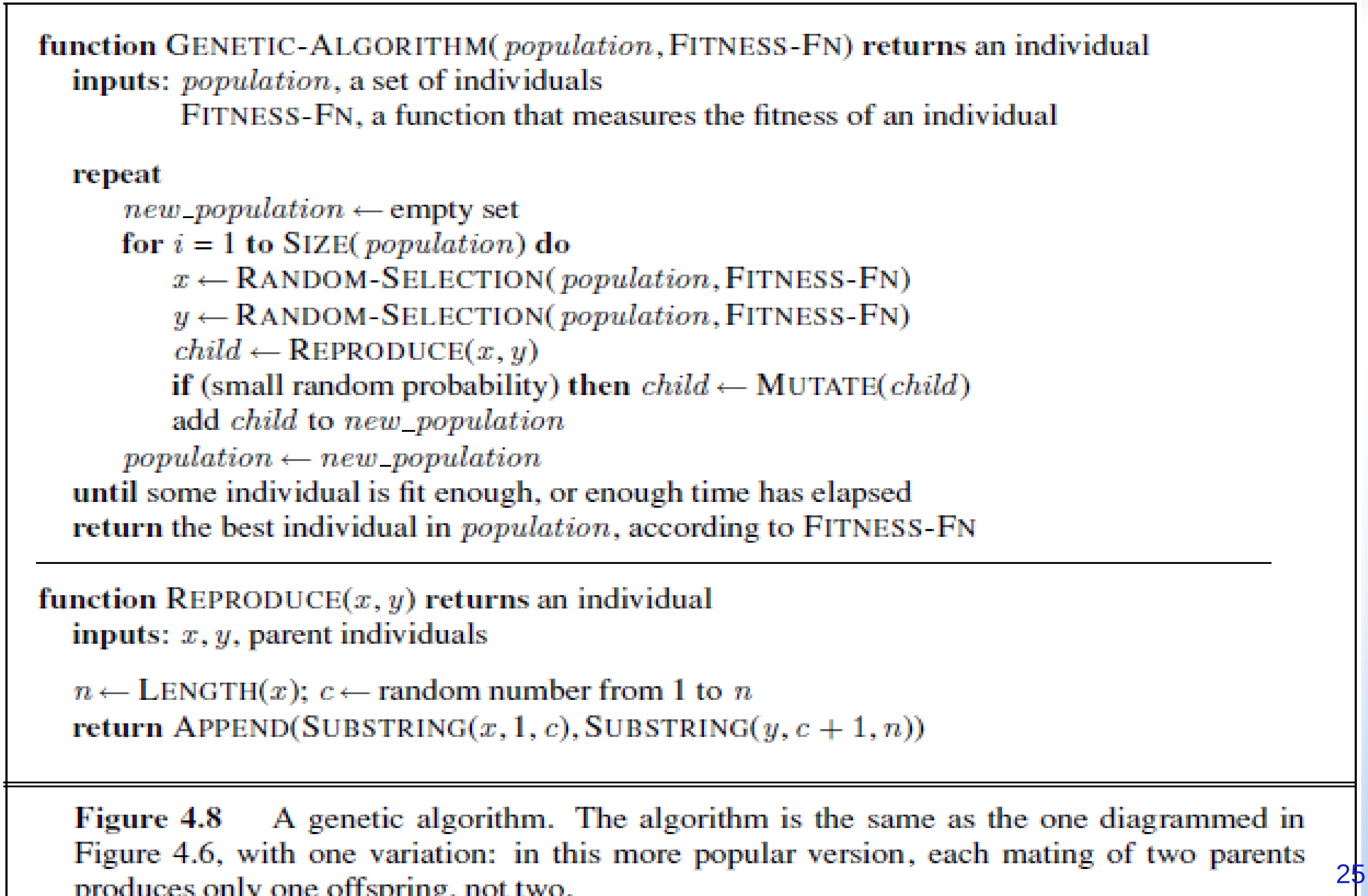


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

Genetic Algorithm



```
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 ← empty set
     for $i$ = 1 to SIZE(population) do
        $x$ ← RANDOM-SELECTION(population, FITNESS-FN)
        $y$ ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE($x, y$)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
     population ← 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

  $n$ ← LENGTH($x$); $c$ ← 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.

On Representation…

- **Schema**: a substring in which some of the positions can be left unspecified.
  - E.g., 246***** describes all 8-queens states in which the first three queens are in positions 2, 4, and 6, respectively.
  - Strings that match the schema (such as 24613578) are called **instances** of the schema.
- It was shown that <u>if the average fitness of the instances of a schema is above the mean</u>, then the number of instances of the schema within the population will grow over time.
- Genetic algorithms work best when schemata correspond to meaningful components of a solution.
- For example, if the string is a representation of an antenna, then the schemata may represent components of the antenna, such as reflectors and deflectors.
- This suggests that a successful use of genetic algorithms requires careful engineering of the representation.

Outline

- Local Search Algorithms and Optimization Problems
  - Hill-climbing search
  - Simulated annealing
  - Local beam search
  - Genetic algorithms
- Searching with Nondeterministic Actions
  - The erratic vacuum world
  - AND–OR search trees
- Searching with Partial Observations
  - Searching with no observation
  - Searching with observations

Searching With Nondeterministic Actions

- In Chapter 3, we assumed that the environment
  - is fully observable and deterministic and
  - that the agent knows what the effects of each action are.

  ➔ the agent can calculate exactly which state results from any sequence of actions & always knows which state it is in.
- When the environment is either partially observable or nondeterministic (or both), percepts become useful.
  - In a partially observable environment, every percept helps narrow down the set of possible states the agent might be in;
  - When the environment is nondeterministic, percepts tell the agent which of the possible outcomes of its actions has actually occurred.

The erratic vacuum world

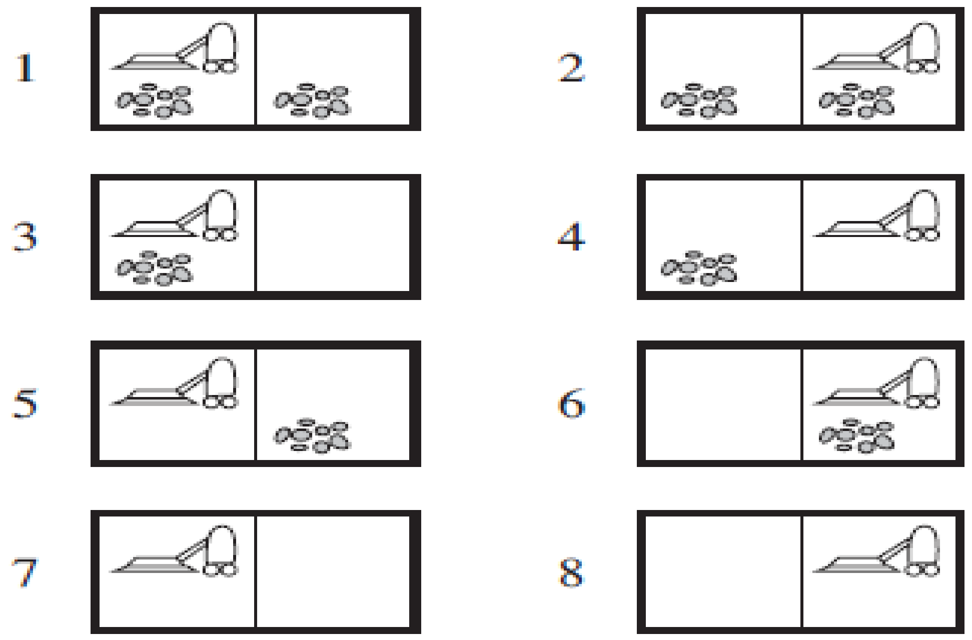


Figure 4.9 The eight possible states of the vacuum world; states 7 and 8 are goal states.

- If the environment is observable, deterministic, and completely known, then the problem is trivially solvable by any of the algorithms in Chapter 3.
- The solution is an action sequence. E.g. from State 1, action sequence [*Suck*,*Right*,*Suck*] will reach goal state 8.

Non-determinism

- Suppose that we introduce nondeterminism in the form of an erratic vacuum cleaner in which, the *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 formalize the problem, we need to generalize the notion of a **transition model.**
- Here we use a function that returns a ***set* of possible outcome states**. E.g. in the erratic vacuum world, the *Suck* action in state 1 leads to a state in the set {5, 7}

Erratic vacuum cleaner world

- We also need to generalize the notion of a **solution** to the problem.
  - E.g. 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** [ ]] .
- So, solutions for nondeterministic problems can contain nested **if–then–else** statements.

AND–OR search trees

- Search trees:
  - In a deterministic environment: only branching is introduced by the agent's own choices in each state. These nodes are called **OR nodes**. E.g. vacuum world, 3 branches: *Left, Right* ***or*** *Suck.*
  - In a non-deterministic environment: because any of alternative nodes can occur (non-deterministically), ***all*** must be considered. These nodes are called **AND nodes**. E.g.:
    - From sate 1, either *Suck* **or** Right.
    - *Suck* action in state 1 leads to a state in the set {5, 7}, so a plan is needed for state 5 ***and*** for state 7.
    - These two kinds of nodes alternate, leading to an **AND–OR tree**

And-Or search

- A solution for an AND–OR search problem is a subtree that:
  - (1) has a *goal* or a *Loop* node at every leaf,
  - (2) specifies one action at each of its OR nodes, and
  - (3) includes every outcome branch at each of its AND nodes.

And-Or tree for the Erratic Vacuum world

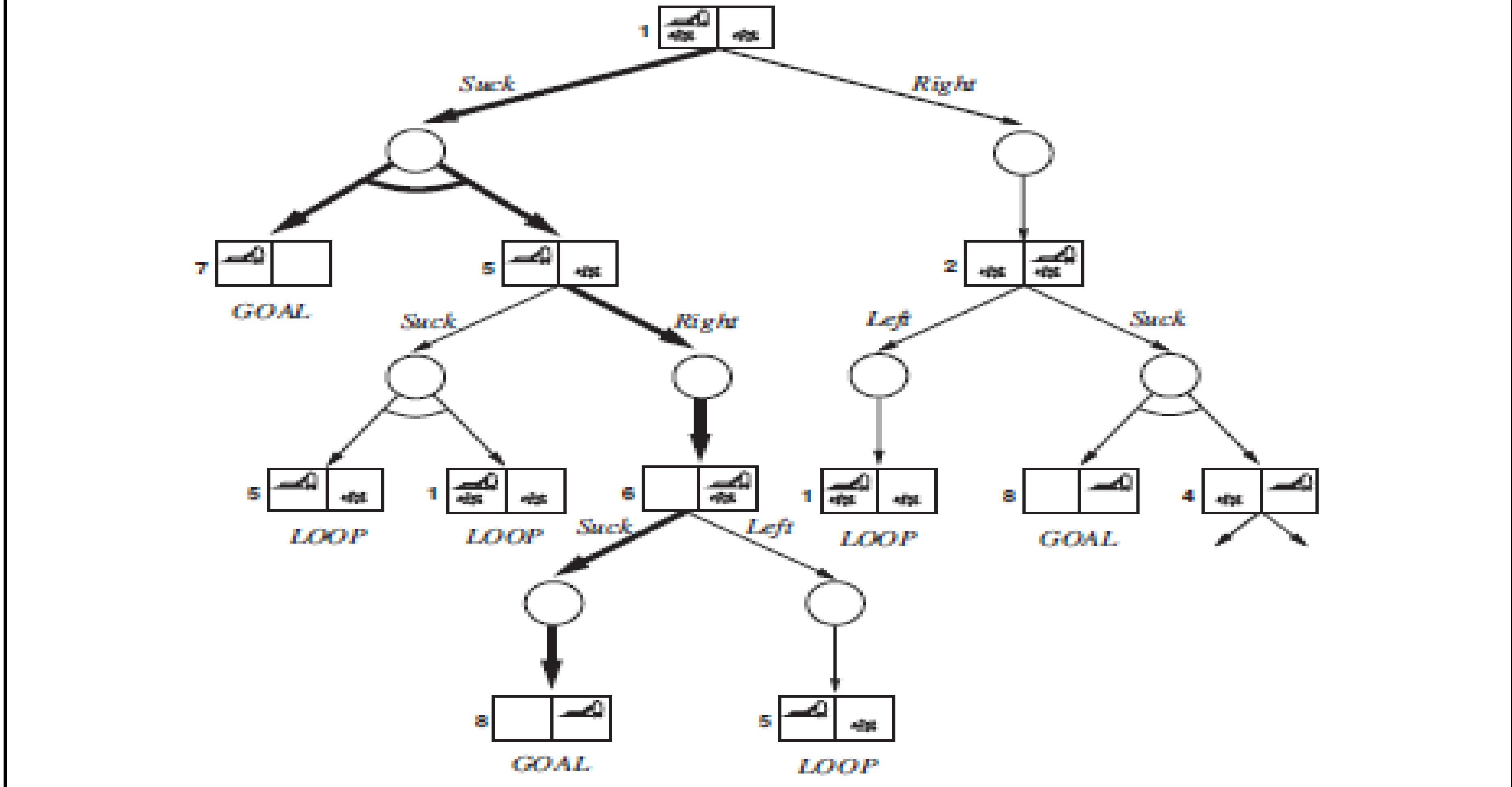
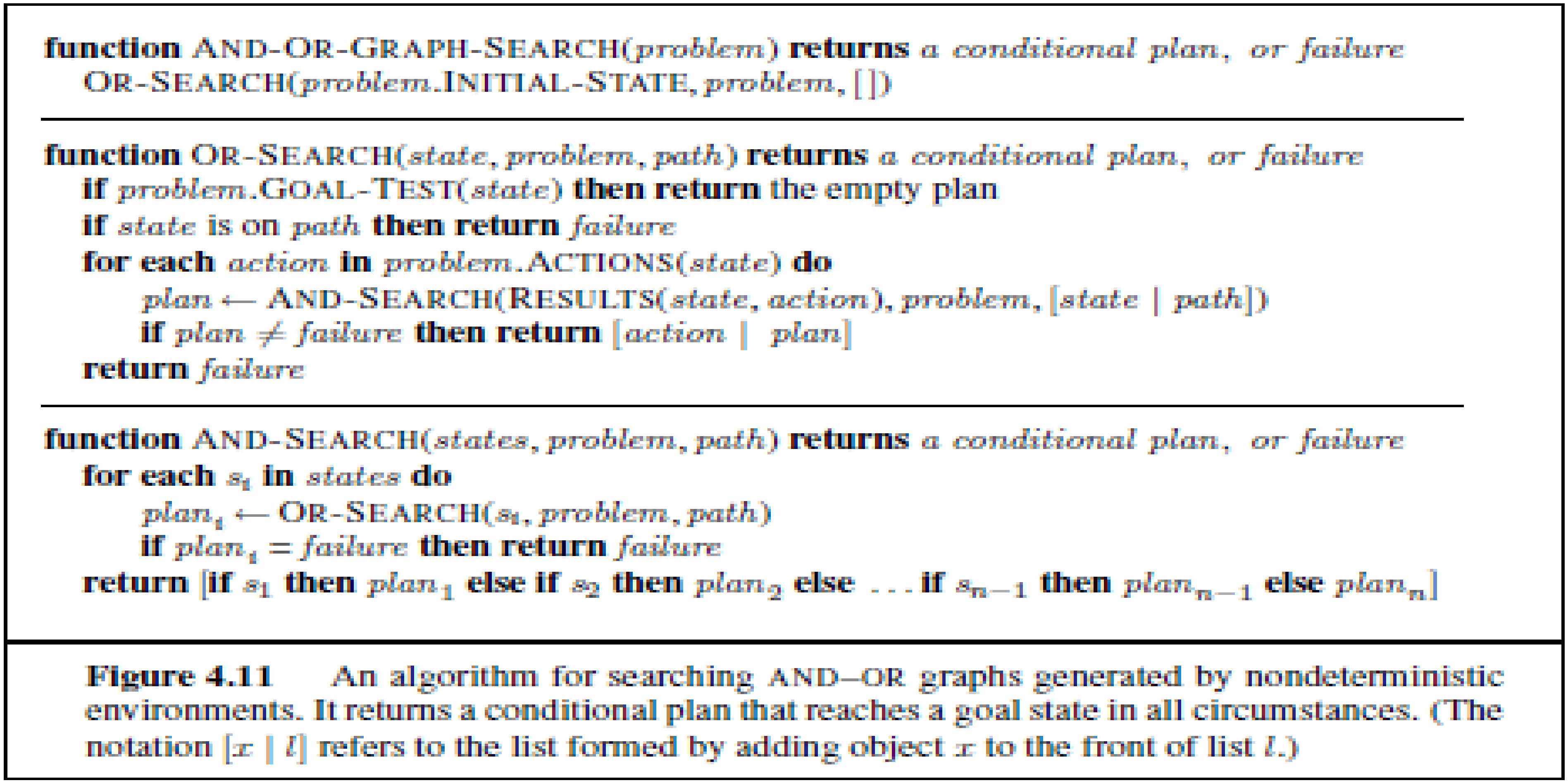


Figure 4.10 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

And-Or Graph Search

- One key aspect of the AND–OR graph search algorithm is the way in which it deals with cycles, which often arise in nondeterministic problems.
  - If the current state is identical to a state on the path from the root, then it returns with failure.
  - This doesn’t mean that there is *no* solution from the current state;
  - it simply means that if there *is* a noncyclic solution, it must be reachable from the earlier occurrence of the current state, so the new one can be discarded.
  - This way, we ensure that the algorithm terminates in every finite state space, because every path must reach a goal, a dead end, or a repeated state.

And-Or Graph Search



```
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.)

- AND–OR graphs can also be explored by breadth-first or best-first methods.
- The concept of a heuristic function must be modified to estimate the cost of a contingent solution rather than a sequence.
- The notion of admissibility carries over and there is an analog of the A∗ algorithm for finding optimal solutions.

Slippery vacuum cleaner world

- The slippery vacuum world is identical to the ordinary (non-erratic) vacuum world except that movement actions sometimes fail, leaving the agent in the same location.
- Example: moving *Right* in state 1 leads to the state set {1, 2}.
- There are no acyclic solutions from state 1, and AND-OR-GRAPH-SEARCH would return with failure.
- There is, however, a **cyclic solution**, which is to keep trying Right until it works. This solution can be expressed by adding a **label** to denote some portion of the plan.
- The cyclic solution is [*Suck*, L1 : Right , **if** State =5 **then** L1 **else** Suck] (or better "**while** State =5 **do** Right .")

The slippery vacuum world

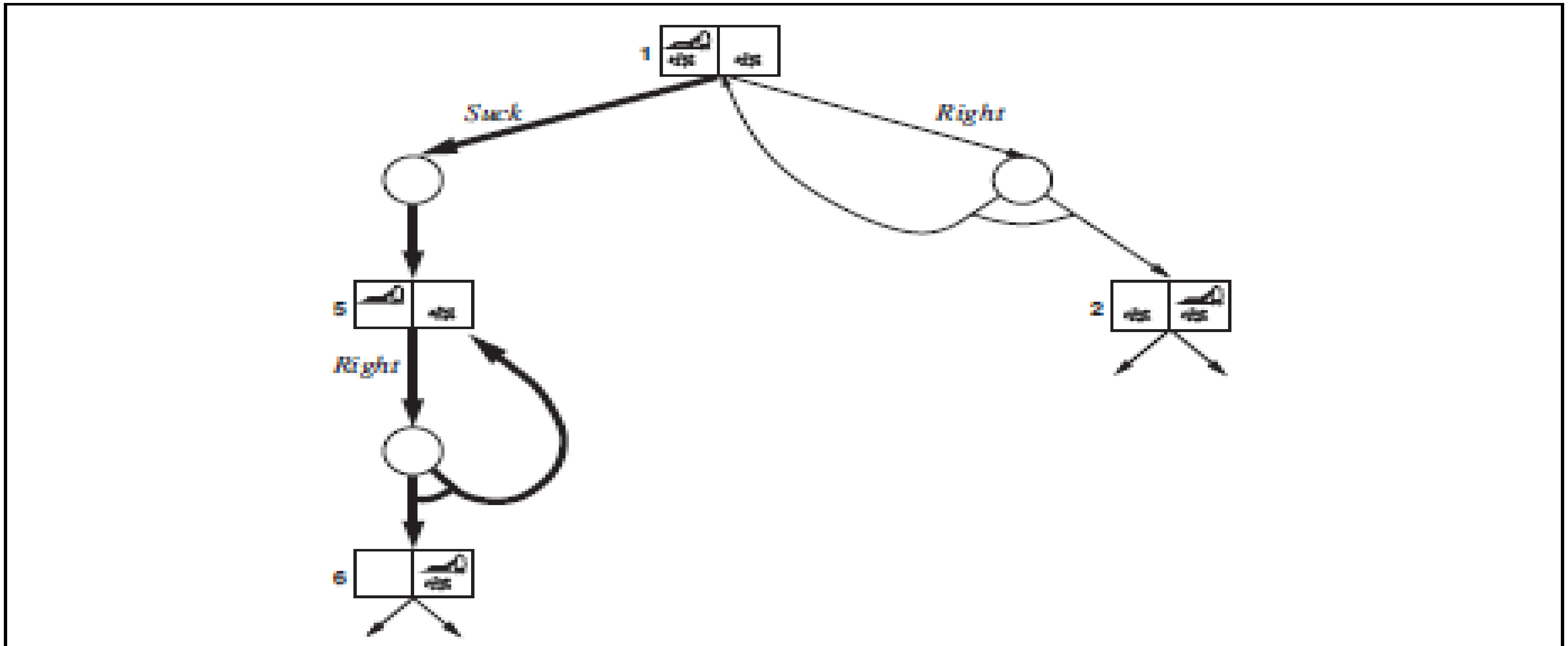


Figure 4.12 Part of the search graph for the slippery vacuum world, where we have shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably.

Cyclic solution:

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

(or, better, "while State =5 do Right .")

Outline

- Local Search Algorithms and Optimization Problems
  - Hill-climbing search
  - Simulated annealing
  - Local beam search
  - Genetic algorithms
- Searching with Nondeterministic Actions
  - The erratic vacuum world
  - AND–OR search trees
- Searching with Partial Observations
  - Searching with no observation
  - Searching with observations

Searching with Partial Observations

- The key concept required for solving partially observable problems is the ***belief state***.
- A **Belief state** represents 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.
- **Searching with no observation**: when the agent’s percepts provide *no information at all*, we have what is called a **sensorless** (also **conformant**) problem.
- Sensorless agents can be useful in some applications/settings.

Searching with no observation

- Examples of sensorless agents:
    - Manufacturing systems: ingenious methods have been developed for orienting parts correctly from an unknown initial position by using a sequence of actions with no sensing at all.
    - Medical doctors: often prescribe a broad spectrum of antibiotics rather than using the contingent plan of doing an expensive blood test. Waiting for the results to prescribe a more specific antibiotic and perhaps hospitalization may allow the infection to progress too far.

Sensorless version of the vacuum world

- Suppose the agent knows the geography of its world, but doesn't know its location or the distribution of dirt.
  - So initial state could be any element of the set {1, 2, 3, 4, 5, 6, 7, 8}.
  - If action is *Right* then it will be in one of the states {2, 4, 6, 8}—the agent now has more information!
  - The action sequence [*Right*,*Suck*] will always lead to one of the states {4, 8}.
  - The sequence [*Right*,*Suck*,*Left*,*Suck*] always takes to the goal state 7 whatever the start state.
  - We say that the agent can **coerce** the world into state 7.

Sensorless version of the vacuum world

- To solve sensorless problems, search in the space of belief states rather than physical states.
- Notice that in the belief-state space, the problem is *fully observable* because the agent always knows its own belief state.
- Since no percepts at all, there are no contingencies to plan for, *even if the environment is nondeterminstic.*

Construction of the belief-state search problem

- Suppose the underlying physical problem P is defined by ACTIONS_P, RESULT_P, GOAL-TEST_P, and STEP-COST_P.
- Then the corresponding sensorless problem is defined as follows:
  - **Belief states**: contains every possible set of physical states.
    - If P has N states, then the sensorless problem has up to 2^N states,
    - Note that many states may be unreachable from the initial state.
  - **Initial state**:
    - Typically, the set of all states in P.
    - In some cases, the agent will have more knowledge than this.

Construction of the belief-state search problem

- **Actions**: This is slightly tricky. Suppose the agent is in belief state $b=\{s_1,s_2\}$, but $\mathrm{ACTIONS}_P(s1) \neq \mathrm{ACTIONS}_P(s2)$; 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:

$$\mathrm{ACTIONS}(b) = \bigcup_{s \in b} \mathrm{ACTIONS}_P(s)$$

On the other hand, if an illegal action is potentially dangerous, it is safer to **allow only the *intersection***, i.e. the set of actions legal in *all* the states.

For the vacuum world, every state has the same legal actions, so both methods give the same result.

Construction of the belief-state search problem

- **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

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

  With deterministic actions, b' is never larger than b.

  With nondeterminism, we have

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

  which may be larger than b (see Figure 4.13).

The process of generating the new belief state after the action is called the **prediction** step; the notation $b' = \text{PREDICT}_P(b,a)$ will come in handy.

Construction of the belief-state search problem

- **Goal test**: The agent wants a plan that is sure to work, which means that <u>a belief state satisfies the goal only if *all* the physical states in it satisfy GOAL-TEST$_P$</u>. 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. (This gives rise to a new class of problems, explored in Exercise 4.9.)
  - 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.

Predicting the next belief-state

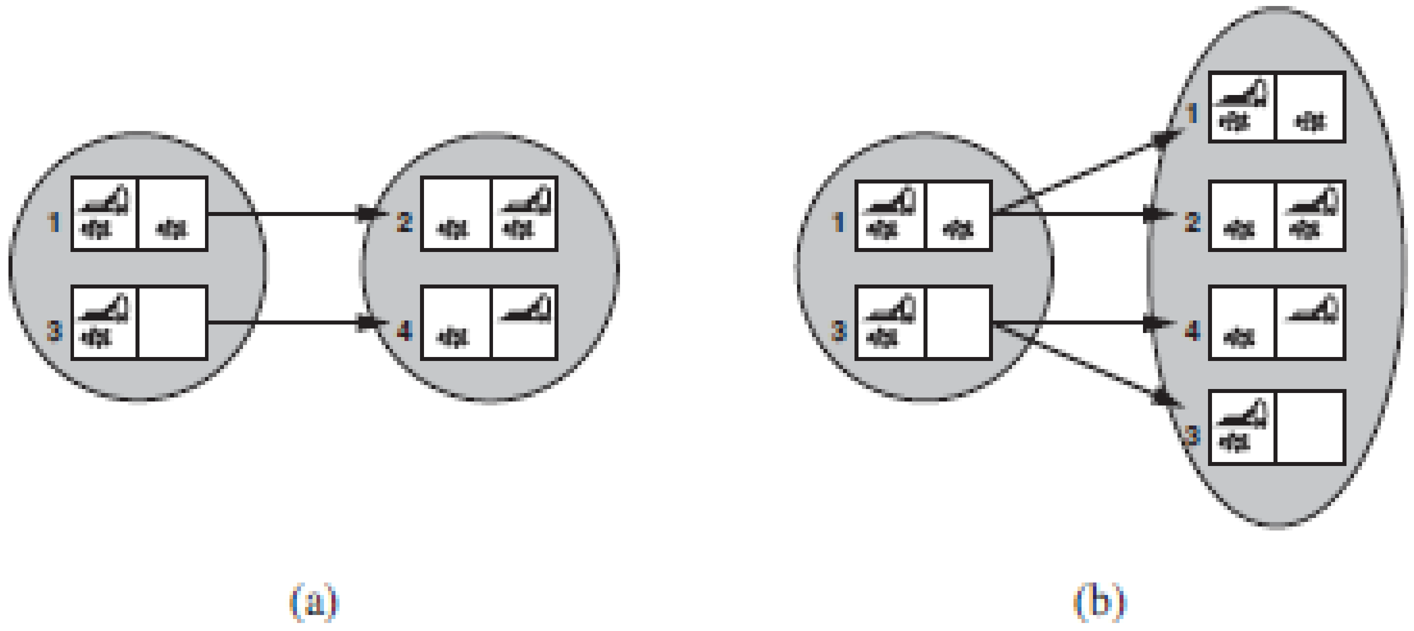


Figure 4.13 (a) Predicting the next belief state for the sensorless vacuum world with a deterministic action, *Right*. (b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

Reachable belief-state space for the deterministic, sensorless vacuum world

There are only 12 reachable belief states out of 2^8 (i.e. 256) possible belief states.

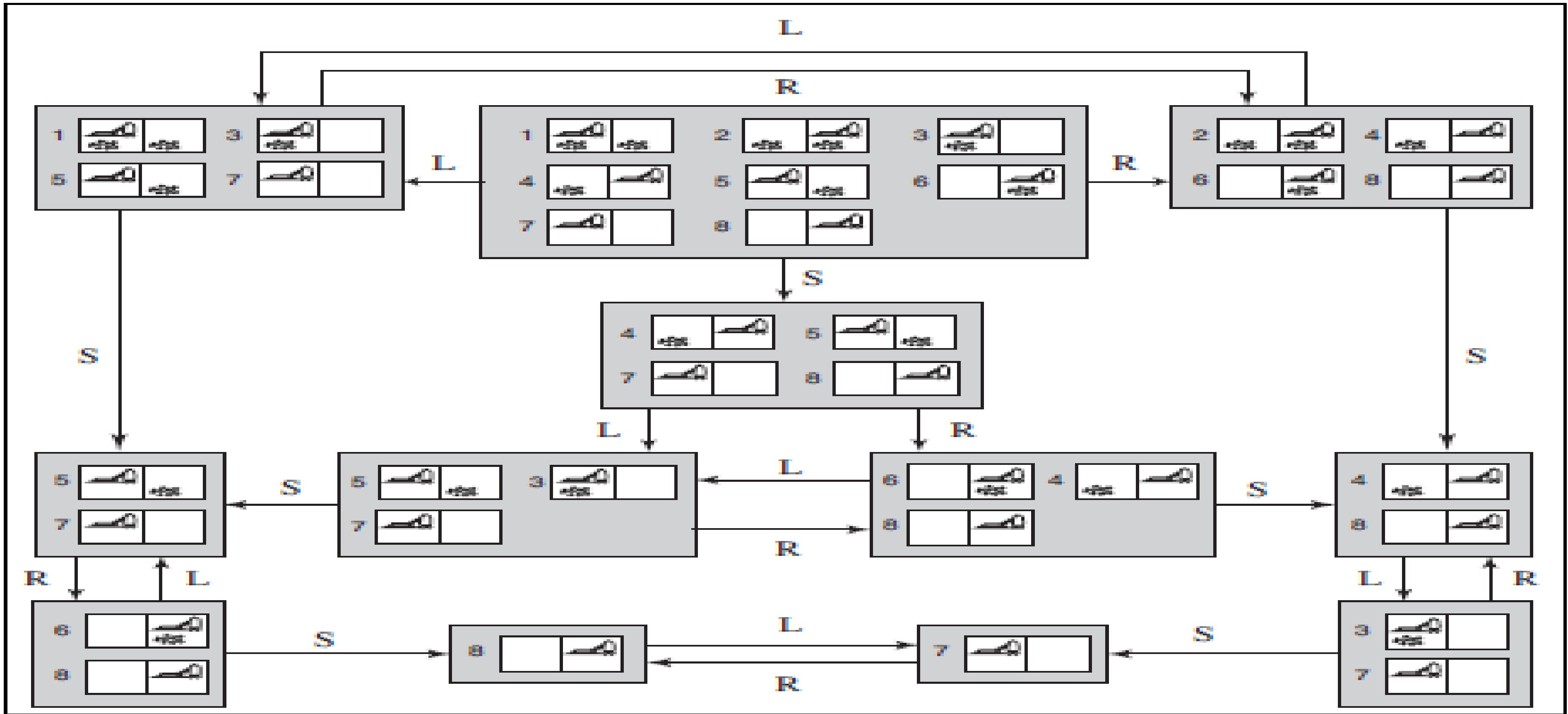


Figure 4.14 The reachable portion of the belief-state space for the deterministic, sensorless vacuum world. Each shaded box corresponds to a single belief state. At any given point, the agent is in a particular belief state but does not know which physical state it is in. The initial belief state (complete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

Search in belief-state spaces

- The preceding definitions enable the automatic construction of the belief-state problem formulation from the definition of the underlying physical problem.
- Once the problem is formulated, the studied search algorithms of Chapter 3 can be applied.
- As in "ordinary" graph search, newly generated states are tested to see if they are "identical" (equivalent) to existing states.
    - Example: The action sequence [*Suck*,*Left*,*Suck*] starting at the initial state reaches the same belief state as [*Right*,*Left*,*Suck*], namely, {5, 7}.

Search in belief-state spaces

- Now, consider the belief state reached by [*Left*], namely, {1, 3, 5, 7}. Obviously, this is not identical to {5, 7}, but it is a *superset* of it.
- It is easy to prove (Exercise 4.8) that if an action sequence is a solution for a belief state b, it is also a solution for any subset of b.
    - So, we can discard a path reaching {1, 3, 5, 7} if {5, 7} has already been generated.
- Conversely, if {1, 3, 5, 7} has already been generated and found to be solvable, then any *subset*, such as {5, 7}, is guaranteed to be solvable.
- This extra level of pruning may dramatically improve the efficiency of sensorless problem solving.

Search in belief-state spaces

- Even with such pruning, sensorless problem-solving as described is seldom feasible in practice.
- The difficulty is not really the vastness of the belief-state space—even though it is exponentially larger than the underlying physical state space;
  - In most cases the branching factor and solution length in the belief-state space and physical state space are not so different.
- The real difficulty lies with the size of each belief state.
  - For example, the initial belief state for the 10×10 vacuum world contains 100×2^{100} (around 10^{32}) physical states—far too many if we use an explicit representation of the list of states.

Search in belief-state spaces

Two possible solutions to internal representation of belief states:

1. Represent the belief state by some more compact description.
   - In English, we could say the agent knows “Nothing” in the initial state; after moving *Left*, we could say, “Not in the rightmost column,” and so on. (Future chapter)
2. Avoid the standard search algorithms; look *inside* the belief states and develop **incremental belief-state search** algorithms that build up the solution one physical state at a time.
   - Do not seek an action sequence for belief state {1,2,3,4,5,6,7,8}; first find a solution that works for state 1; then we check if it works for state 2; if not, go back and find a different solution for state 1, and so on.
   - This approach tends to be very effective!

Searching with observations

- For a general partially observable problem, we have to specify how the environment generates percepts for the agent and which percepts it receives.
- The formal problem specification includes a PERCEPT(s) function that returns the percept received in a given state. (If sensing is nondeterministic, then a PERCEPTS function returns a set of possible percepts.)
  - E.g. 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*

The vacuum world (Reminder)

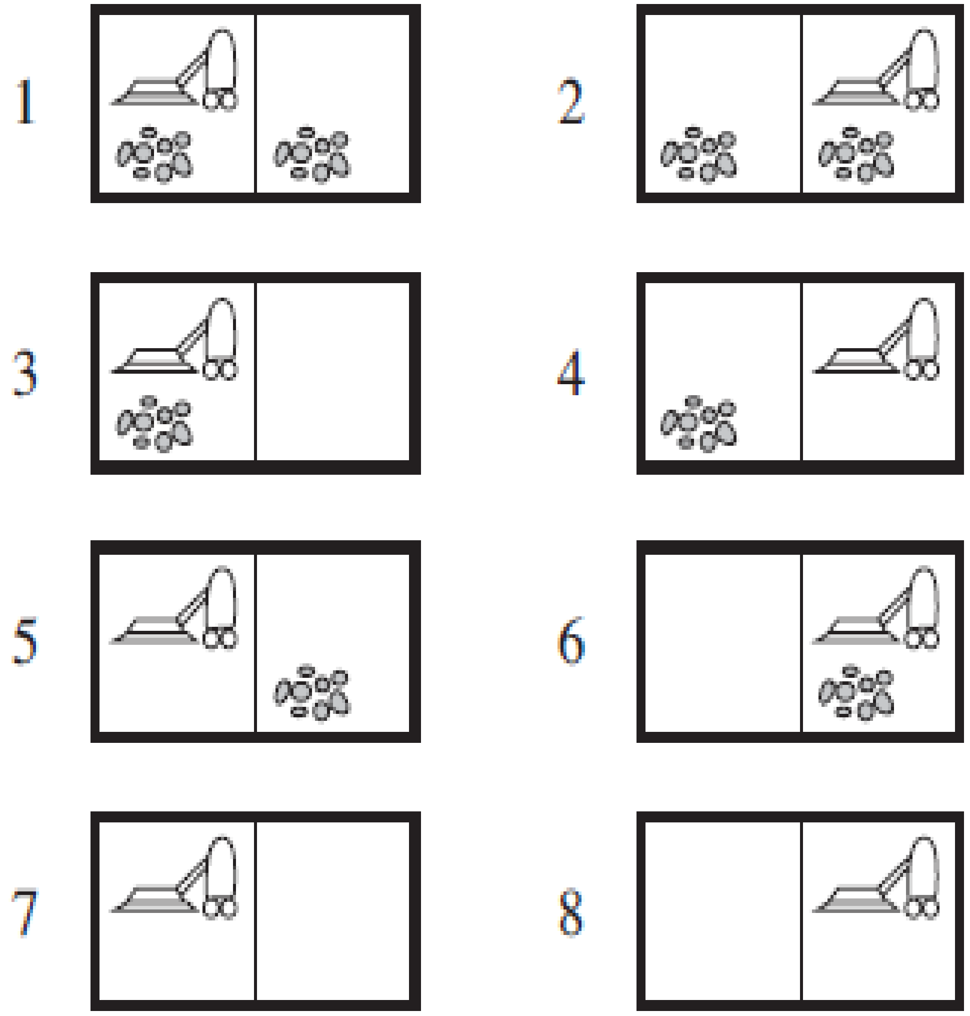


Figure 4.9 The eight possible states of the vacuum world; states 7 and 8 are goal states.

Searching with observations

- When observations are partial, usually several states could have produced any given percept.
  - E.g., percept [A, Dirty] is produced by state 3 as well as by state 1.

  ➔ given this as the initial percept, the initial belief state for the local-sensing vacuum world will be {1, 3}.
- The ACTIONS, STEP-COST, and GOAL-TEST are constructed from the underlying physical problem just as for sensorless problems.
- The transition model is more complicated.

Transition model for partially observed problem

- 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:

$$POSSIBLE\_PERCEPTS(\hat{b}) = \{O\colon\ O = PERCEPT(S)\ and\ S \in \hat{b}\}$$

  - 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:

$$b_o = UPDATE(\hat{b}, O) = \{S : O = PERCEPT(S) and\ S \in \hat{b}\}$$

Transition model for partially observed problem

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

$$RESULTS(b, a) = \{b_o : \ b_o = UPDATE(PREDICT(b, a), o) \ and \\ o \ \in POSSIBLE\_PERCEPTS(PREDICT(b, a))\}$$

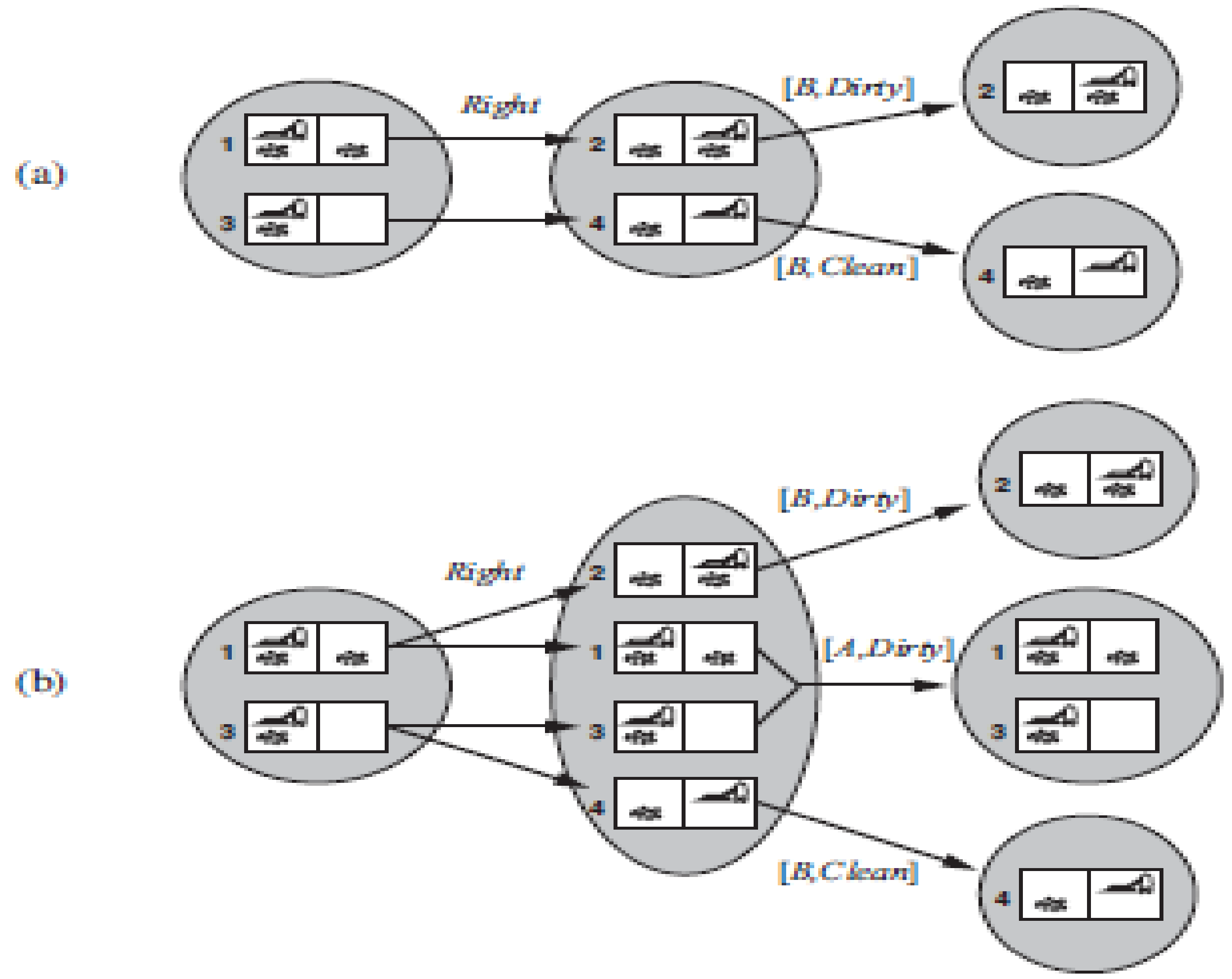
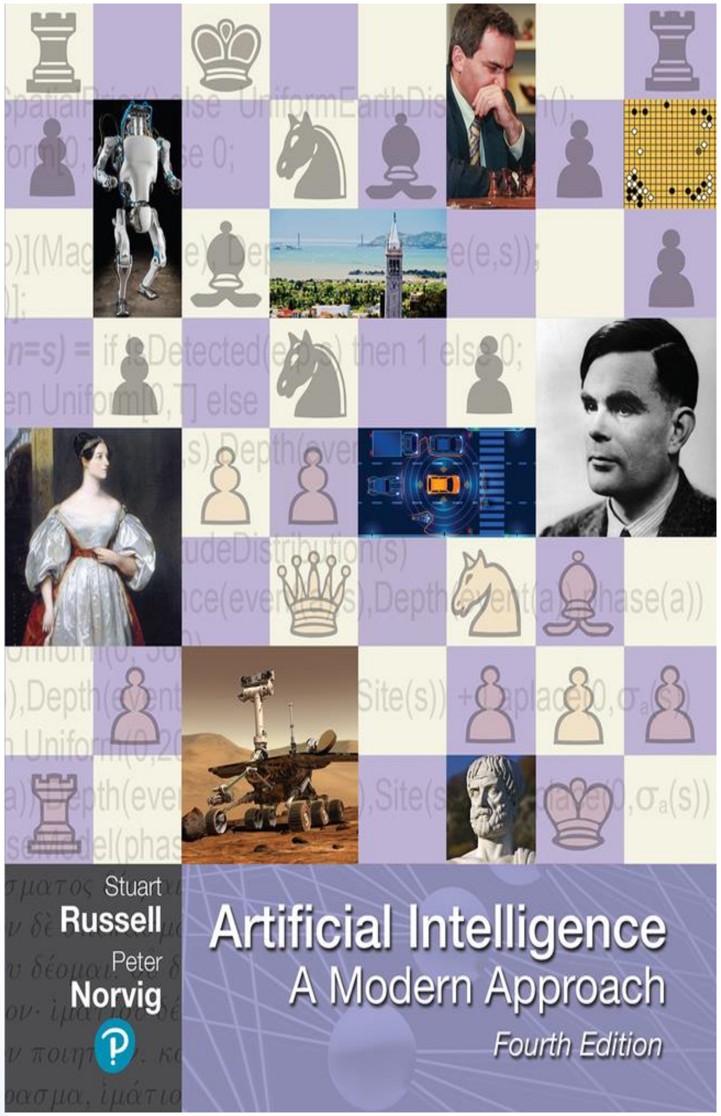


Figure 4.15 Two example of transitions in local-sensing vacuum worlds. (a) In the deterministic world, *Right* is applied in the initial belief state, resulting in a new belief state with two possible physical states; for those states, the possible percepts are [B, *Dirty*] and [B, *Clean*], leading 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, *Dirty*], [B, *Dirty*], and [B, *Clean*], leading to three belief states as shown.

Slides based on the textbook



- Russel, S. and Norvig, P. (2020) Artificial Intelligence,

A Modern Approach (4th Edition), Pearson Education Limited.