Artificial Intelligence
A Modern Approach
Third Edition
Stuart Russell & Peter Norvig

Chapter 4
Beyond Classical Search

Local Search v.s. Systematic Search

- **Systematic search**
  - BFS, DFS, IDS, Best-First, A*
  - Keep some history of visited nodes
  - **Completeness**
    - **Always complete** for finite search spaces
    - **Some versions complete** for infinite spaces
  - Good for “building up” solutions incrementally
    - State：partial solution
    - Action：extend partial solution

- **Local Search**
  - Algorithms (meta heuristic)
    - Gradient descent (steepest decent), greedy local search, Simulated Annealing, Genetic Algorithm
  - Does not keep history of visited nodes
  - **Not complete**
  - May be able to argue terminating with “high probability”
  - Good for “fixing up” candidate solutions
    - State：complete candidate solution that may not satisfy all constraints
    - Action：make a small change in the candidate solution

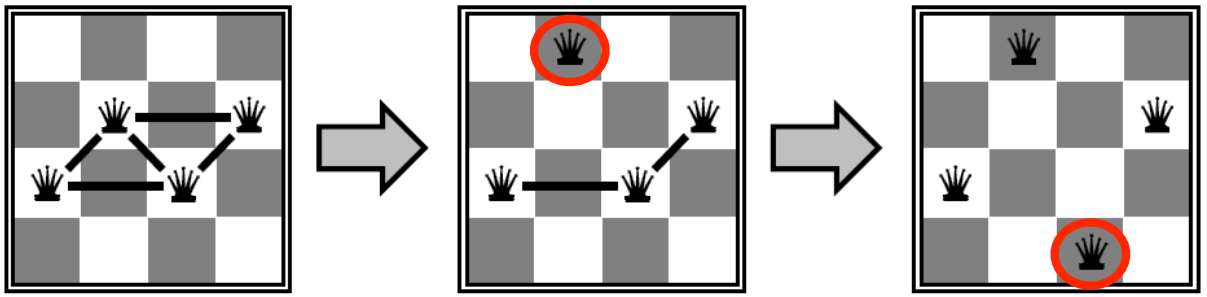
1. Local Search Algorithms and Optimization Problems

- In many optimization problems, the **state space** is the space of **all possible complete solutions**
- **Objective function**
  - How do you measure **cost** or **value** of a state?
  - Tells us **how "good"** a given state is, and we want to find the **solution (goal)** by **optimizing** (minimizing or maximizing) the value of this function

- If the **start state** may not be specified and the **path** to the goal doesn't matter
  - In such cases, we can use **local search algorithms** that keep a single **current** state (very memory efficient) and gradually try to **improve** it

Example: *n*-Queens Problem

- **Goal**: put n queens on an $n \times n$ board
  - No two queens on the same row, column, or diagonal
  - An NP-hard problem
- **Neighbor**: move 1 queen to another row or column
- **Search**: go from one neighbor to the next….
- **State space**: all possible n-queen configurations
- **Objective function**: number of **pairwise conflicts**
- What's a possible **local improvement strategy**?
  - Move one queen to another square to **reduce conflicts**



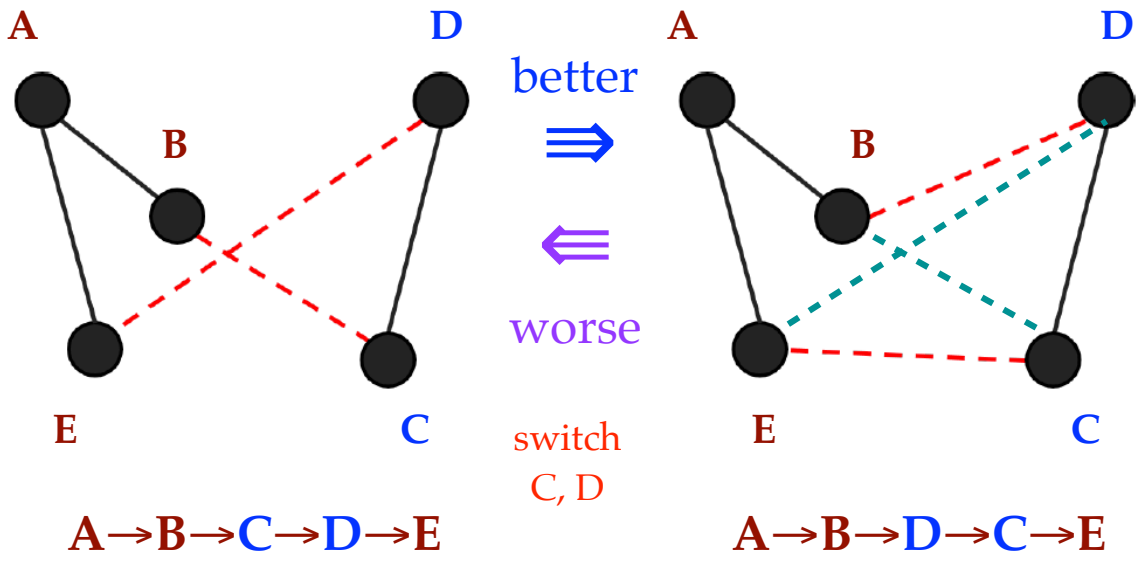
h: the no. of pairwise conflicts

Example: Traveling Salesman Problem

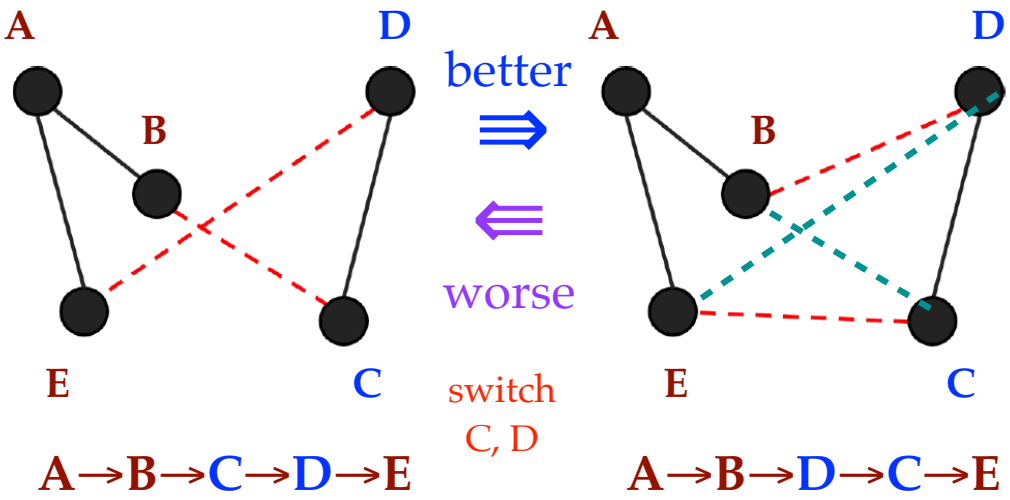
- Find the **shortest tour** connecting a given set of cities
- **State space**: all possible tours
- **Objective function**: length of tour
- TSP is an NP-hard problem

- What's a possible **local improvement strategy**?
  - Start with any **complete tour** (could be random)
  - Solution is a **permutation**
  - Check if any **pairwise exchange** (e.g., operator：**swap-2**) shorten the tour
  - Stop criterion：no more better (shorten the tour) move

ABCDE $\xrightarrow{C_2^n \text{ valid moves}}$ C_2^n states, then apply **steep decent** to select the **min cost**



- We can check **swap-*k*** of course, but it takes $O(n^k)$ time



- Empirically, swap-2 or swap-3 using **steepest decent** makes a good TSP search even there are hundreds of cities

- Number of possible actions to take (resulting number of states)

original state

  - swap-2 : $C_2^n(2! - 1) \approx O(n^2)$
  - swap-3 : $C_3^n(3! - 1) \approx O(n^3)$

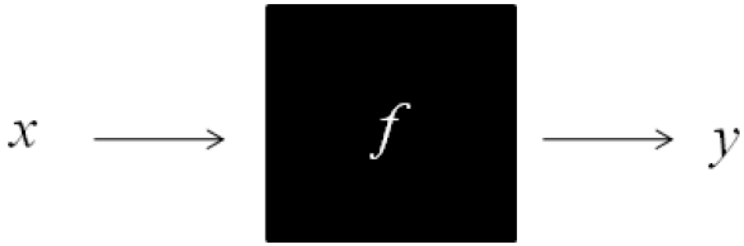
original state

$$P(n,r) = {}^nP_r = {}_nP_r = \frac{n!}{(n-r)!}$$

$$C(n,r) = {}^nC_r = {}_nC_r = \binom{n}{r} = \frac{n!}{r!(n-r)!}$$

Optimization Problems

$y = f(x)$



Equation solving

$x + 10 = 15 \Rightarrow x = 5$

Optimization

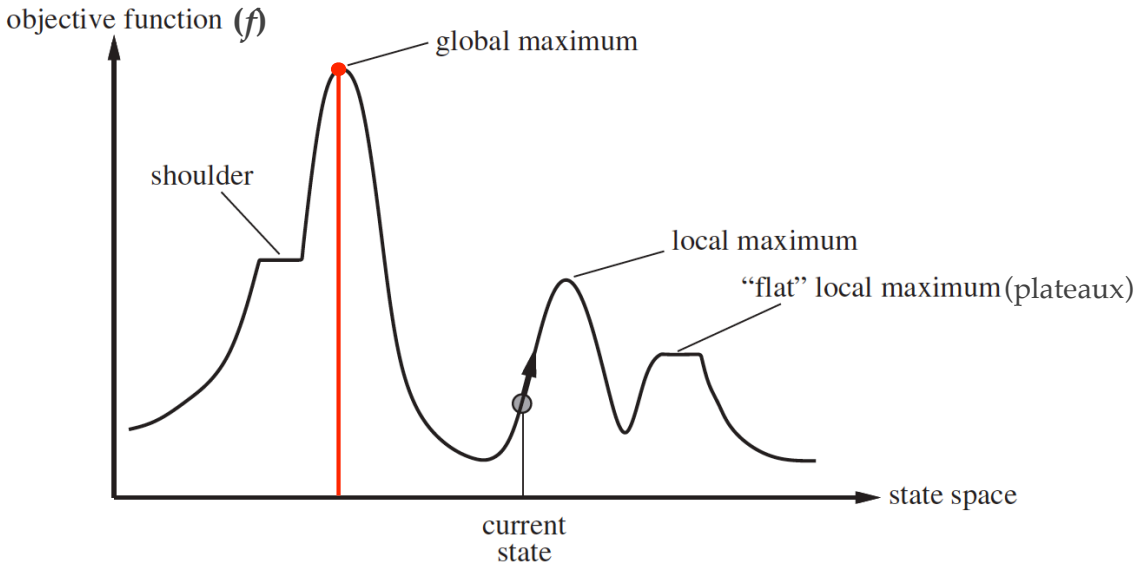
$x + 10 = 15 \Rightarrow x \rightarrow 5$

- Find x that yields **highest y** with an **unknown** f (**objective function**)
- **Local** vs. **global** optima

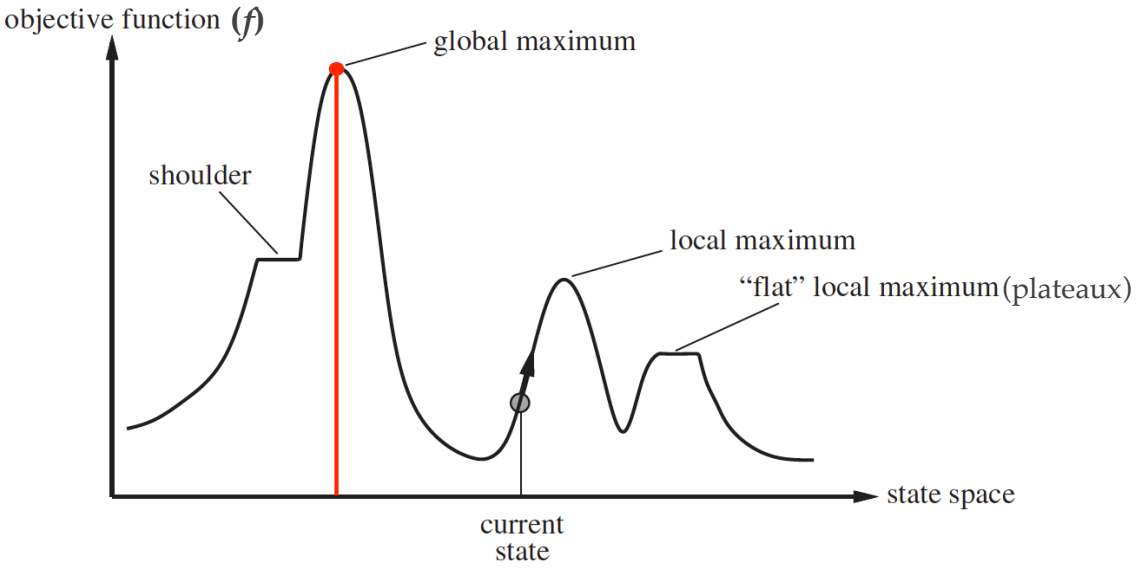
$f(x)$: may be simple function or complex system of equations

Glass box: $f(x)$ is known

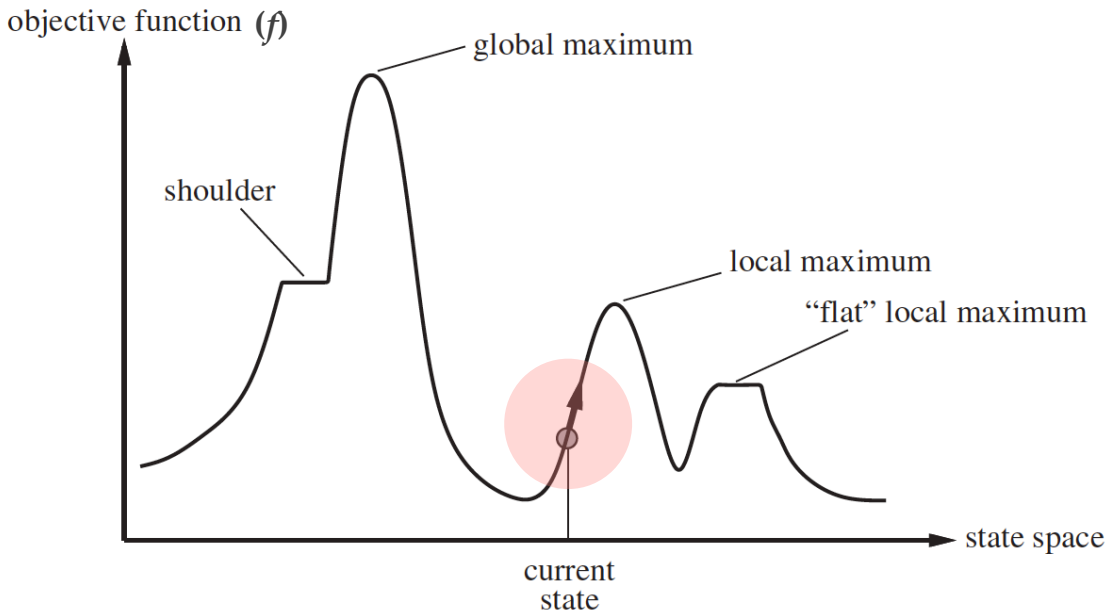
Black box: $f(x)$ is unknown



- To understand local search, we consider the **state-space landscape**
  - **Location**: defined by the **state**
  - **Elevation**: defined by the **value** of the **heuristic cost function** or **objective function (*f*)**
    - If elevation corresponds to **cost**
      - **Aim**: find the **lowest valley** — a **global minimum**
    - If elevation corresponds to an **objective function**
      - **Aim**: find the **highest peak** — a **global maximum**



- A **complete** local search algorithm: always finds a goal if one exists
- An **optimal** algorithm: always finds a **global minimum/ maximum**
- **Hill-climbing search** modifies the current state and tries to improve it, as shown by the **arrow**

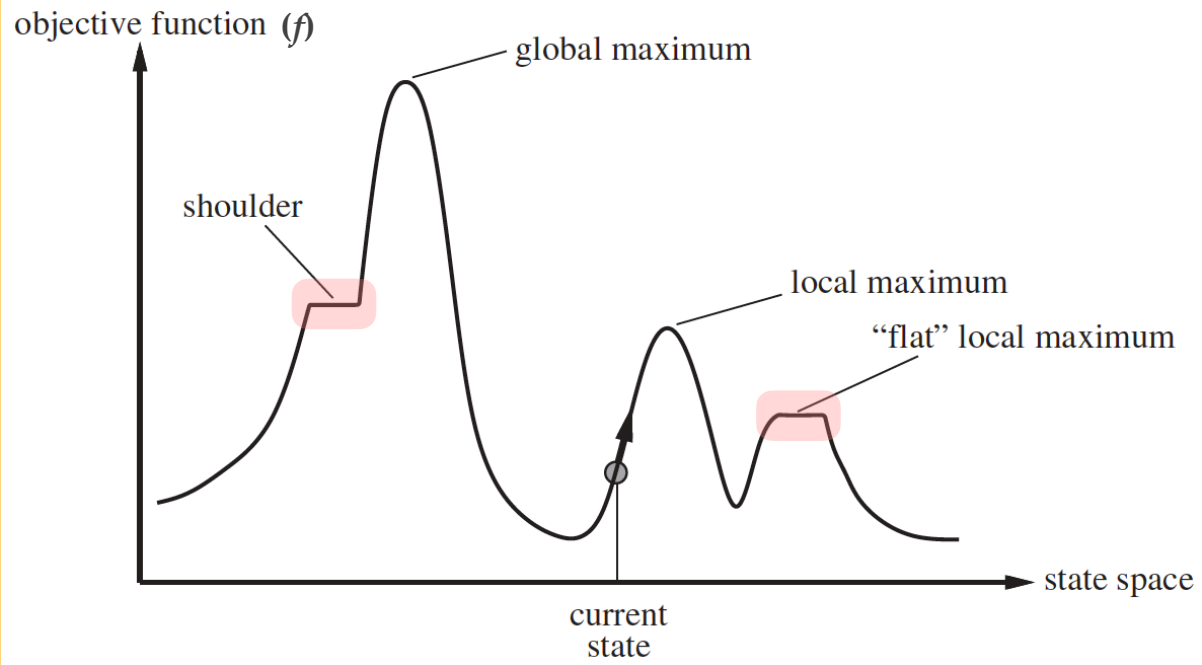


Hill Climbing Search

- A **loop** that continually moves in the direction of **increasing value**, i.e., **uphill**
- It **terminates** when it reaches a **peak** where no neighbor has a higher value
- The algorithm does **not** maintain a **search tree**
  - The **data structure** for the current node needs only record the **state** and the **value** of the **objective function**
- Hill climbing does **not look ahead** beyond the immediate neighbors of the current state
- "Like climbing mount Everest in thick fog with amnesia"

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

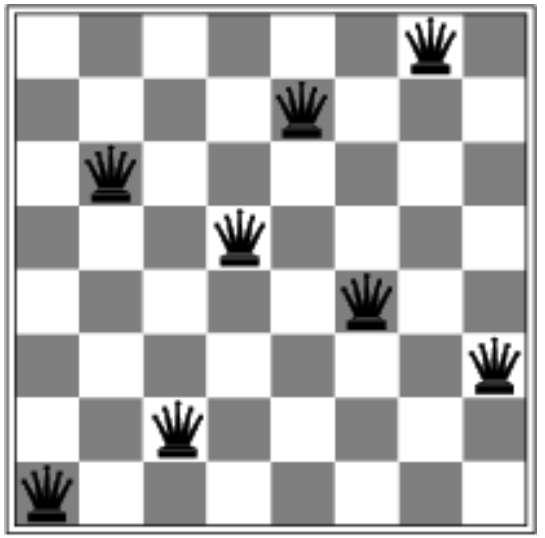
current ← MAKE-NODE(*problem*.INITIAL-STATE)
loop do
neighbor ← a highest-valued successor of *current*
if neighbor.VALUE ≤ current.VALUE **then return** *current*.STATE
current ← *neighbor*



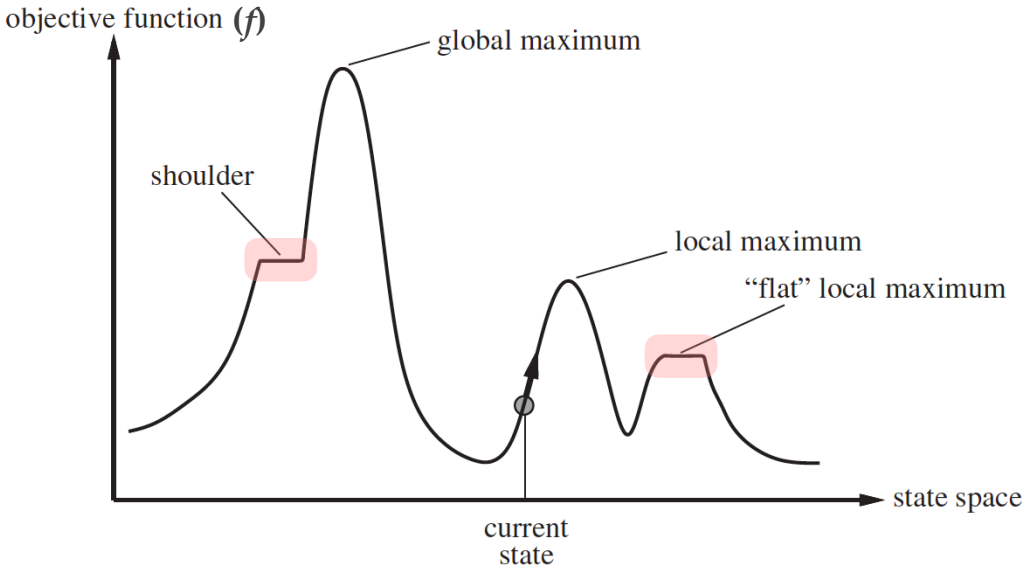
Set limit for ≤ ? for stoping criterion

- The hill climbing search algorithm (**steepest-ascent** version)
  - At each step the current node is **replaced** by the **best neighbor**; i.e., the neighbor with the **highest value**
  - If a **heuristic cost estimate, *h*,** is used, we would find the neighbor with the **highest *h***
- **Variants**
  - Choose first better successor
  - Randomly choose among better successors

- Is it **complete**/**optimal**?
  - No, can get **stuck** in **local optima**
  - Example: local optimum for the 8-queens problem using steepest decent
- How to **escape** local maxima?
  - **Random restart** hill-climbing
- What about "shoulders"?
  (not optimal)
- What about "plateaux"?
  (local optimal)

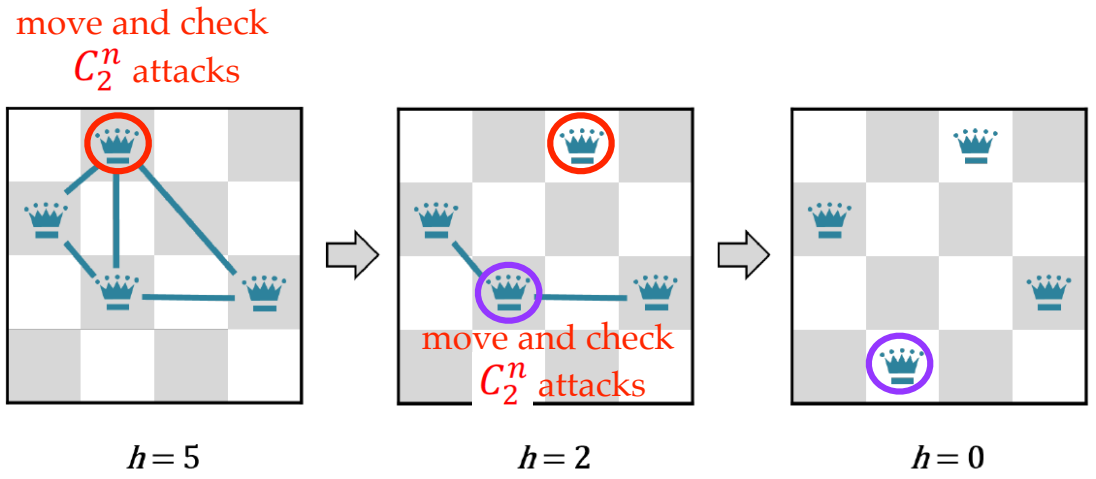


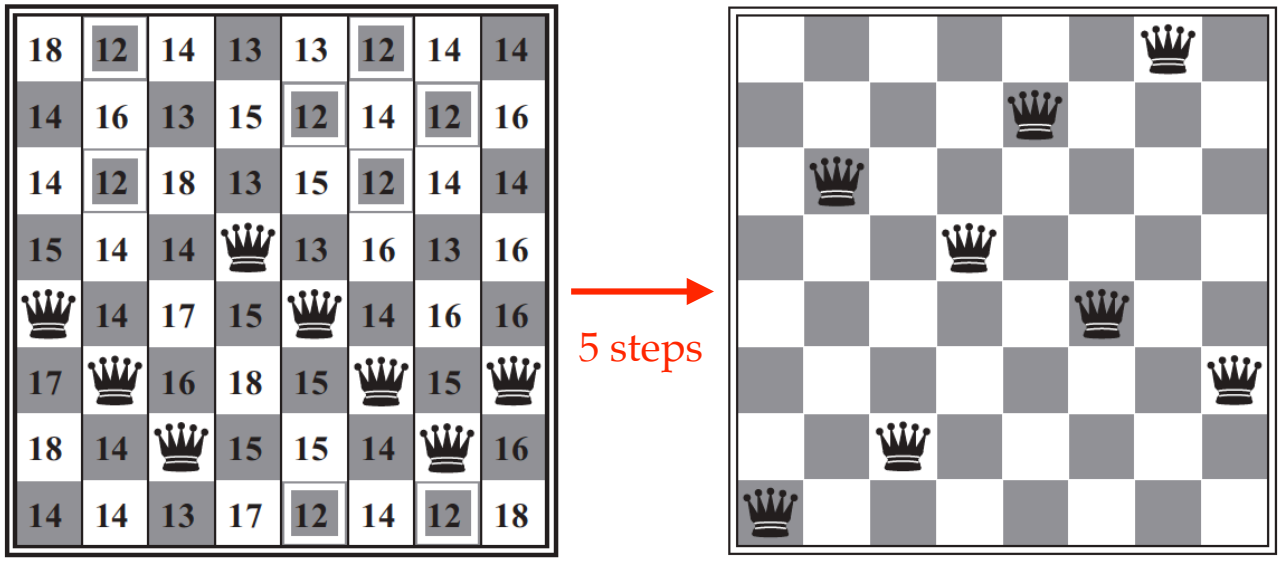
h=1



Hill-Climbing Search: 8-Queens Problem

- The **successors** of a state: all possible states generated by **moving a single queen to another square**
- The **heuristic cost function, h**: the number of pairs of queens that are attacking each other, either directly or indirectly
- The **global minimum** of this function: **zero**, which occurs only at **perfect solutions**
- Move a queen to **reduce conflicts**
- Almost always solve n-queen puzzle (optimal solution) **immediately** even for $n \cong 10^6$ but not for $n \cong 10^{12}$

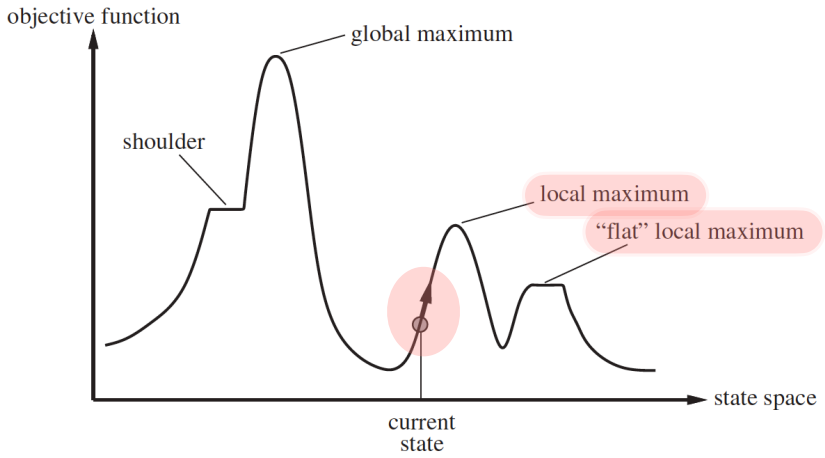


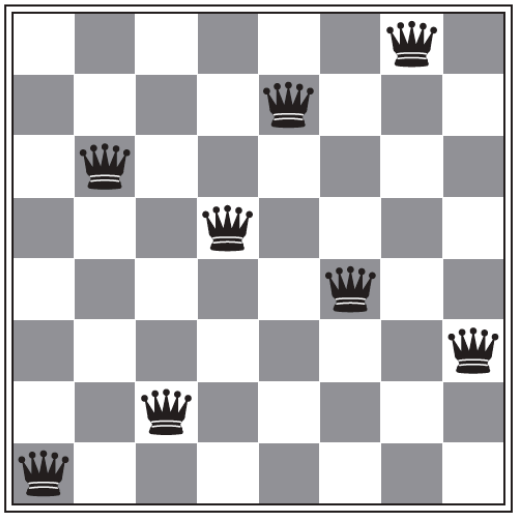


(a) $h = 17$ (b) $h = 1$

- **(a)** h = number of pairs of queens that attack each other, either directly or indirectly (h = 17 for (a))
- Each number indicates h if we move a queen in its **column** to that square
- The values of all its successors, with the **best successors** having h = 12
- Hill-climbing algorithms typically **choose randomly** among the set of **best successors** if there is **more than one**
- From the state in (a), it takes just **five** steps to reach the state **(b)**, which has h= 1 and is very nearly a solution

- Hill climbing often **gets stuck** for the following reasons **(difficulties of ALL local search algorithms)**
  - **Local maxima**
    - A **peak** that is **higher** than each of its neighboring states but **lower** than the global maximum
    - Hill-climbing algorithms that reach the **vicinity** of a **local maximum** will be drawn **upward** toward the peak but will then be **stuck** with **nowhere else to go**
    - For example, the state in figure (b) is a **local maximum** ($h = 1$); **every move** of a single queen makes the situation **worse (higher h)**

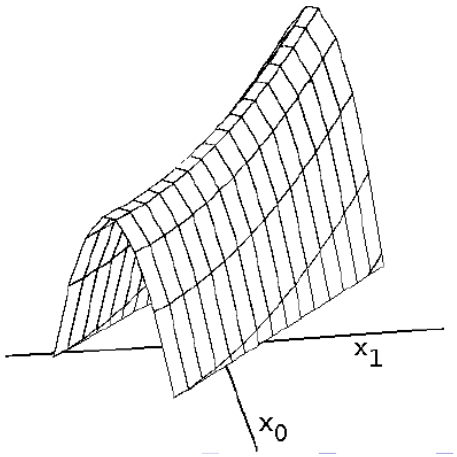


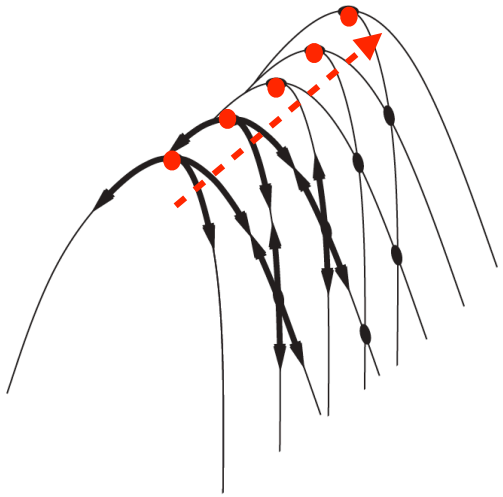


h = 1, local maximum

(b)

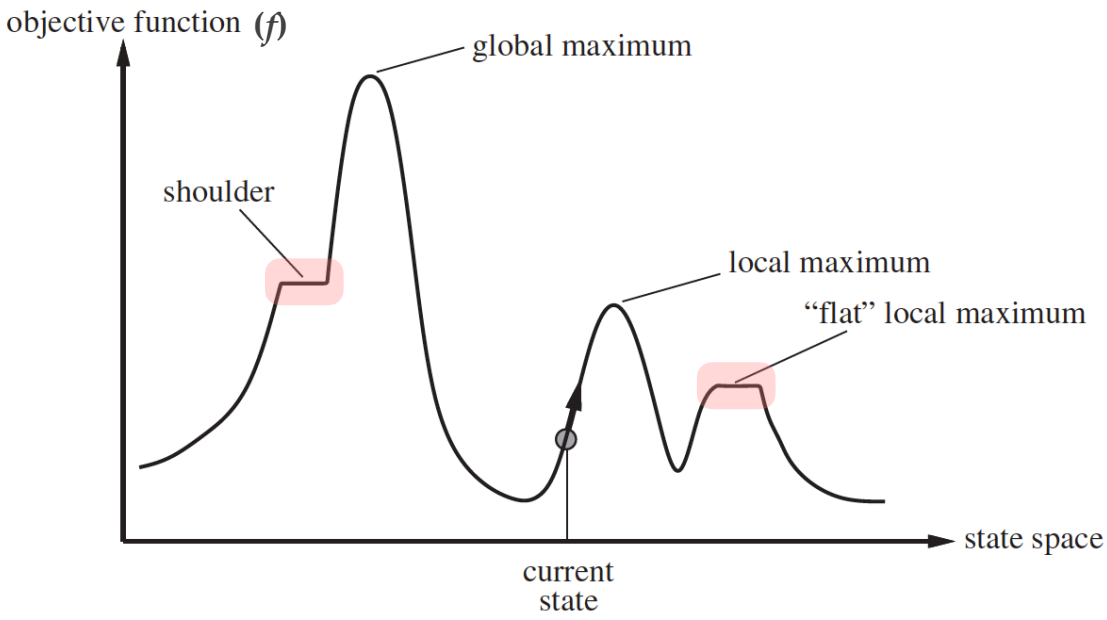
- **Ridges**
  - Fold a piece of paper and hold it tilted up at an **unfavorable angle** to every possible search space step
  - **Every neighbor appears to be downhill; but the ridge leads uphill**
  - From each **local maximum**, all the available actions point downhill
  - The **grid** of states (red 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**, so it is very difficult for greedy algorithms to navigate





Plateaux

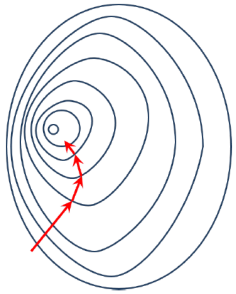
- A plateau is a **flat area** of the state-space landscape
- It can be a **flat local maximum**, from which no uphill exit exists, or a **shoulder**, from which progress is possible
- A hill-climbing search might **get lost** on the plateau



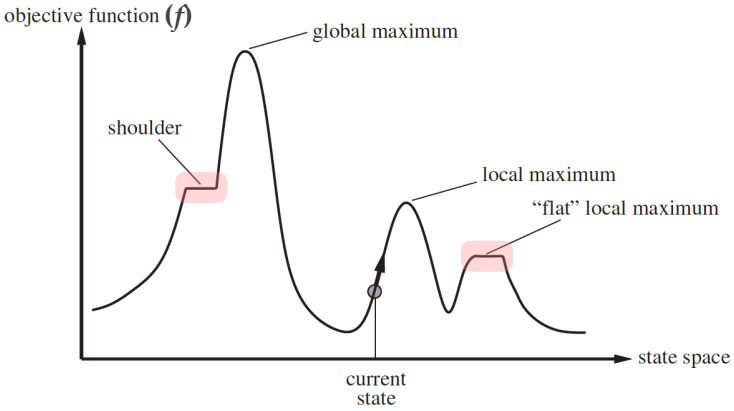
Steepest Descent (Ascent)

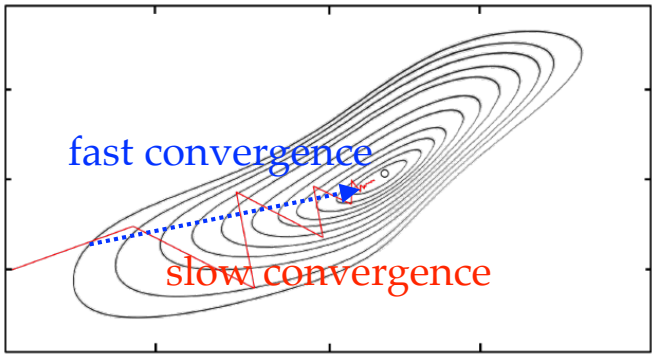
- Steepest descent is a.k.a. **gradient descent**

$s \xrightarrow{a \in \text{Actions}(s)} s'$, (successor or neighbor, evaluate $f(s')$ and find $max(f)$ or $min(f)$

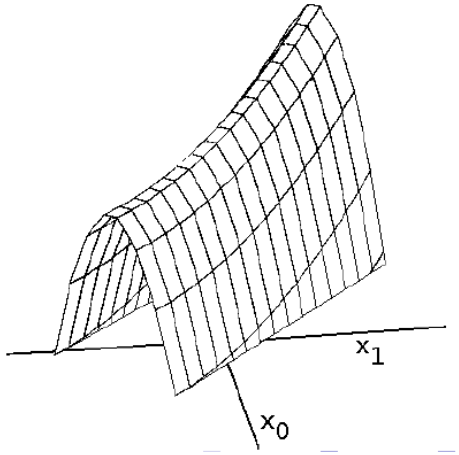


- Allowing **sideway moves** is usually good, but need to put a **limit** to **prevent infinite loop**
- Find only the **nearest** local optimum
- No guarantee **optimal** solution
- **Performance**: may suffer from **slow convergence** due to **zig-zagging** behavior



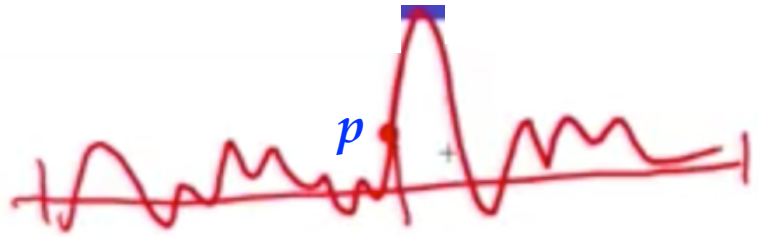


- Ridges and plateau are difficult too



- Random **restart** helps
  - Prob. p to succeed (restart on the hill close to the global optimal) (e.g. $p = 0.1$)
  - $1/p$ restarts needed to succeed (e.g. $1/p = 10$)
  - **Cost = 1 × cost-of-success + $(1 - p)/p$ × cost-of-failure**
    (1 time) (9 times)

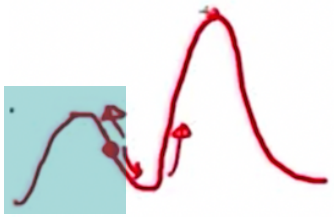
Failure probability: $(1-p)+(1-p)^2+(1-p)^3+\ldots = (1-p)/p$



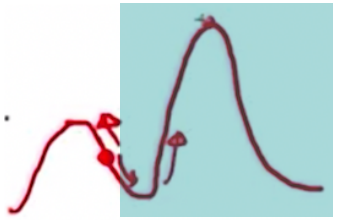
Simulated Annealing (SA)

- **Hill-climbing**
  - Never makes "downhill" moves toward states with lower value (or higher cost) — is **incomplete**, because it can get **stuck** on a **local maximum**
- **Purely random walk**
  - Moving to a successor chosen uniformly at random from the set of successors — is **complete** but extremely **inefficient**

- **Simulated annealing**
  - Combine hill climbing with a random walk in some way that yields both **efficiency** and **completeness**

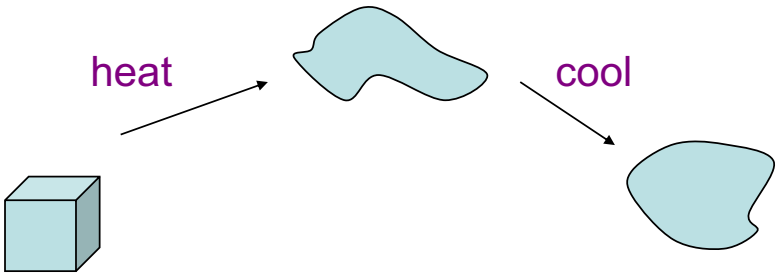


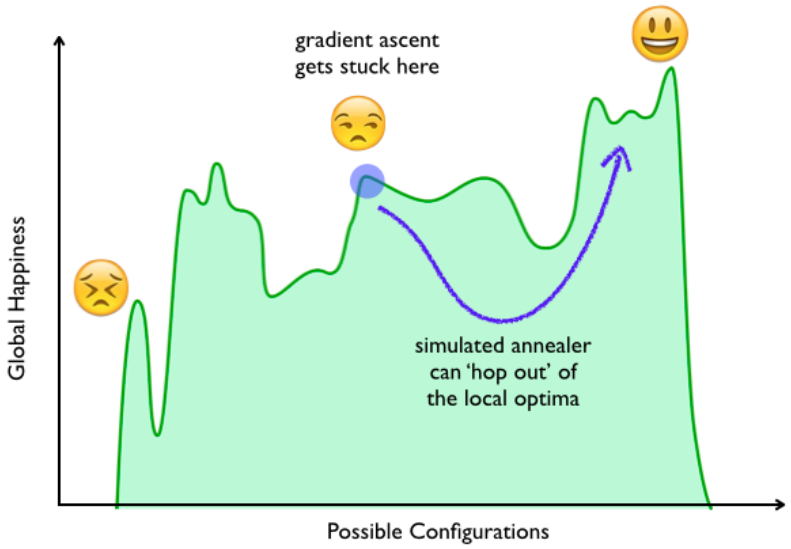
  - **Motivation**
    - Improve the problem of **steepest decent** that gives only the **nearest local optimal**
    - SA expects to use **random walk** to explore extensively (with some probability of going "downhill" and then going another way of "uphill") in the beginning, and then use **steepest decent** afterward



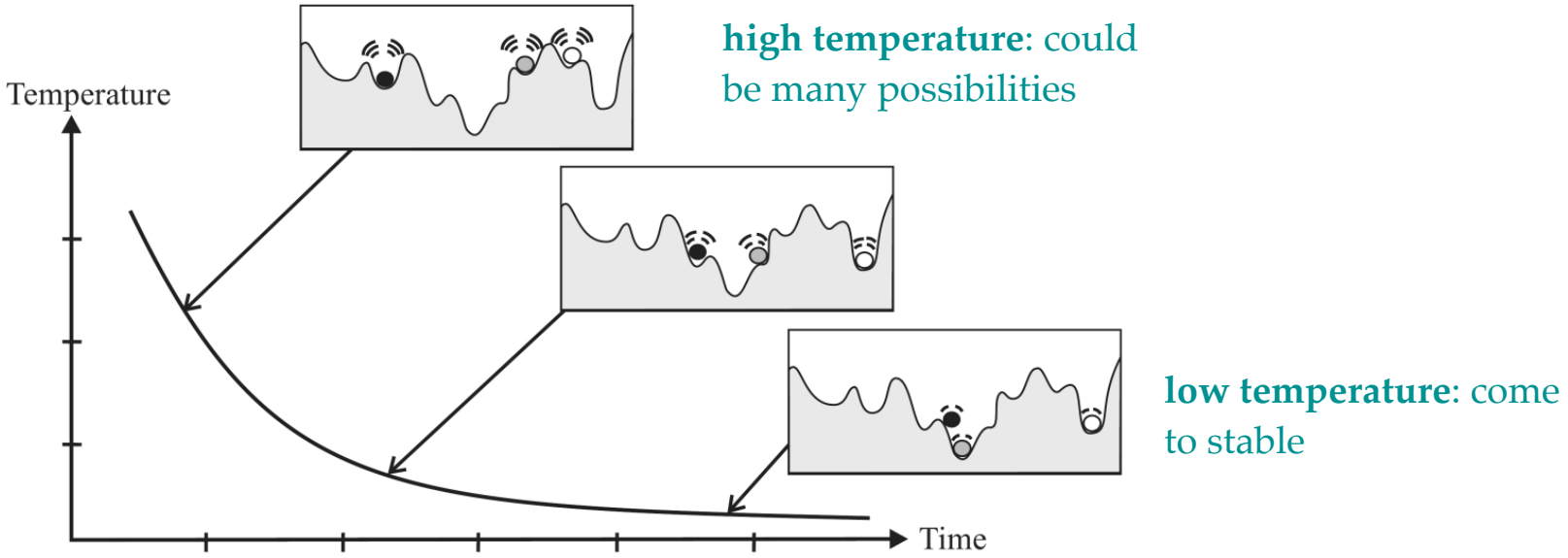
- In metallurgy, annealing is the process used to temper or harden metals and glass by **heating them to a high temperature** and then **gradually cooling them**, thus allowing the material to reach a **low energy crystalline state**

- The simulated-annealing solution is to **start** by **shaking hard** (i.e., at a high temperature) and then **gradually reduce the intensity** of the shaking (i.e., lower the temperature)





- **Idea**: escape local maxima by allowing some "bad" moves but gradually **decrease their frequency**
- Simulated annealing introduces a temperature parameter, *T*, which **cools down** as time goes by



[$f(x)$]

- If the **new state** has a **better** value ($\triangle E > 0$), SA **accepts** the new state [$\triangle E = next.\text{VALUE} - current.\text{VALUE}$]
- If the **new state** has a **worse** value ($\triangle E < 0$), SA **accepts** the new state with a **probability** $e^{\triangle E/T}$

  [$\triangle E$ is negative: (a) $T \rightarrow 0$ then $e^{-\infty} \rightarrow 0$, (b) $T \rightarrow \infty$ then $e^{0} \rightarrow 1$]

  - When T is high, it is easier to accept a state that is worse than the current state.
  - When T is low, less willing to accept a state that is worse than the current state

- **Proved**: If temperature (T) **decreases slowly enough**, then SA search will find a **global optimum** with **probability** approaching **1**

  $T(t)$: T (temperature) is a **decreasing** function of t (time) [inversely proportional]

- However
  - This usually takes a VERY, VERY long time, impractically long
  - Note: any **finite search space**, **RANDOM GUESSING** also will find a **global optimum** with probability approaching **1**
  - So, ultimately this is a **very weak claim**
- SA is widely used in "very big" problems
  - VLSI layout, national airline scheduling, large logistics ops, etc.

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**
$T \leftarrow schedule(t)$ [T is a **decreasing** function of t]
if $T = 0$ **then return** *current*
next ← a randomly selected successor of *current*
$\Delta E \leftarrow next.\text{VALUE} - current.\text{VALUE}$
if $\Delta E > 0$ **then** *current* ← *next*
else *current* ← *next* only with probability $e^{\Delta E/T}$

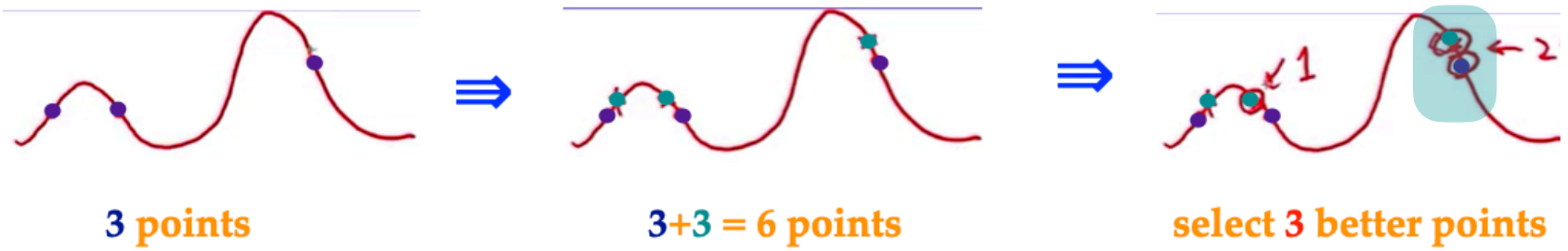
Local Beam Search

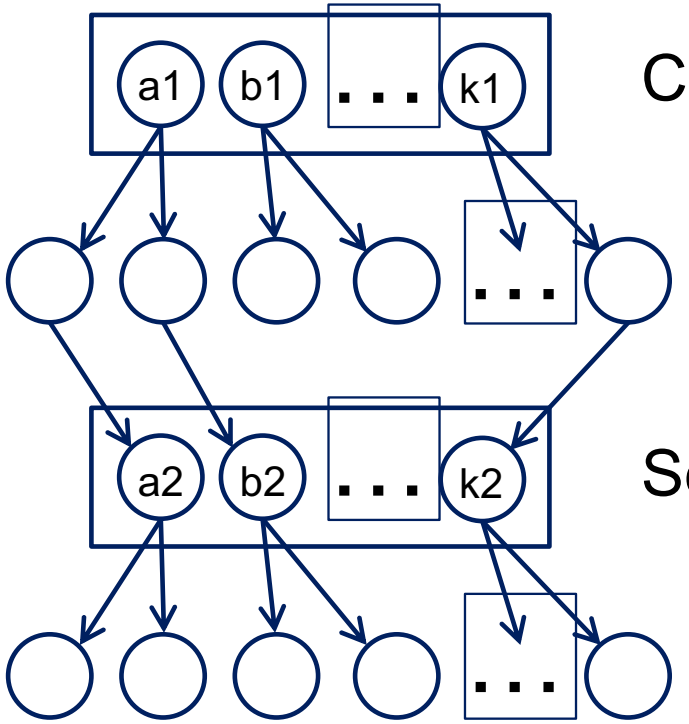
- Start with *k* randomly generated states
- At each iteration, all the successors of all *k* **states** are generated
- Keep best *k* states instead of just one

3 points **3+3 = 6 points** **select 3 better points**

- If any one is a **goal state**, **stop**; else select the *k* **best successors** from the complete list and repeat

- Concentrates search effort in areas believed to be **fruitful**
  - May **lose diversity** as search progresses, resulting in wasted effort (clustering effect may be wrong)





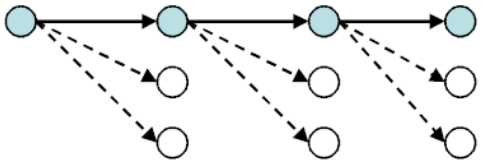
Create *k* random initial stats

Generate their children

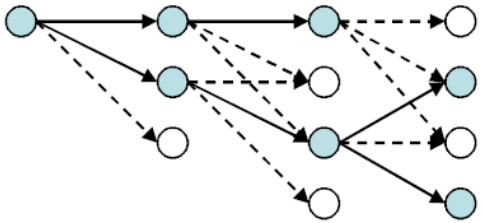
Select the *k* best children

Repeat indefinitely…

- Is this the same as running *k* greedy searches in parallel? (No, beam search have interact among states, but greedy search does not)

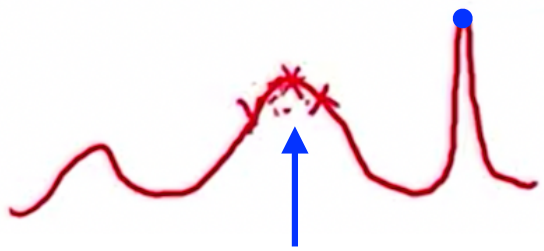


Greedy search



Beam search

- **Stochastic beam search** randomly chooses ***k*** successors with a **probability** proportional to their **goodness** [$f(x)$].
  - Neighbors selection is not limited to higher hills (not 100%, but with higher probability), there is some probability to select from lower hills (with lower probability)
- **Variant**
  - If select 10 points in total, the best 5 points are unconditionally selected, and the remaining 5 points are selected by stochastic

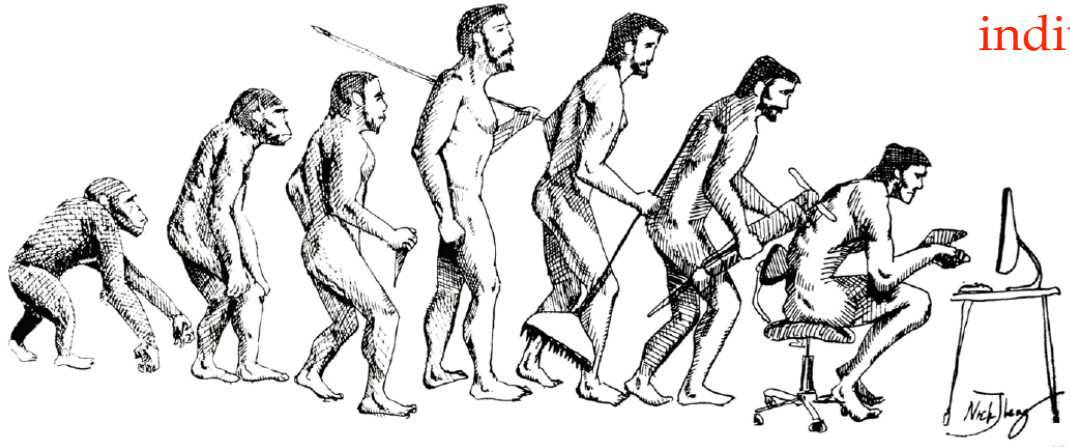


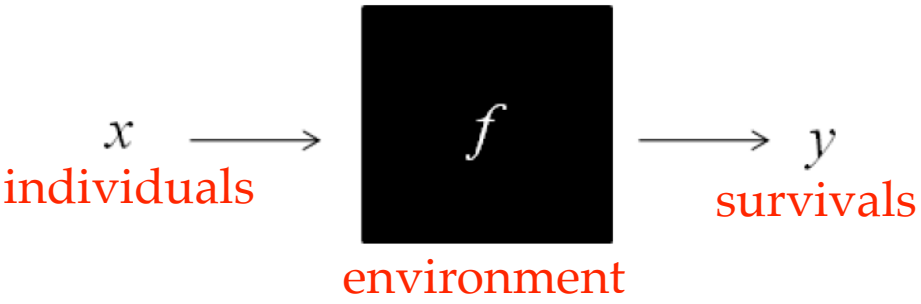
← e.g. This global optimum may have 0.1 (= 1-0.9) probability not being selected

Clustering effect. Seems good, still wrong

Darwinian Evolution

- **Evolution**
  - The change in populations of organisms over generations



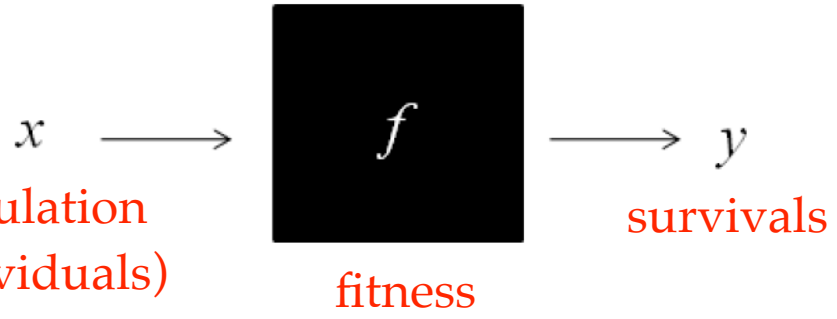


- **Darwin's idea**
  - **Natural selection [自然選擇]**
    - The process whereby organisms **better adapted** to their environment tend to survive and produce more offspring
  - **Struggle to survive [物競天擇]** (mandatory condition)
    - The **competition in nature** among organisms of a population to maintain themselves in a given environment and to survive to reproduce others of their kind
  - **Survival of the fittest [適者生存]**
    - A natural process resulting in the evolution of organisms **best adapted** to the environment
  - **Genetic variation [遺傳變異]**
    - **Variations of genomes** between members of species, or between groups of species thriving in different parts of the world as a result of **genetic mutation**

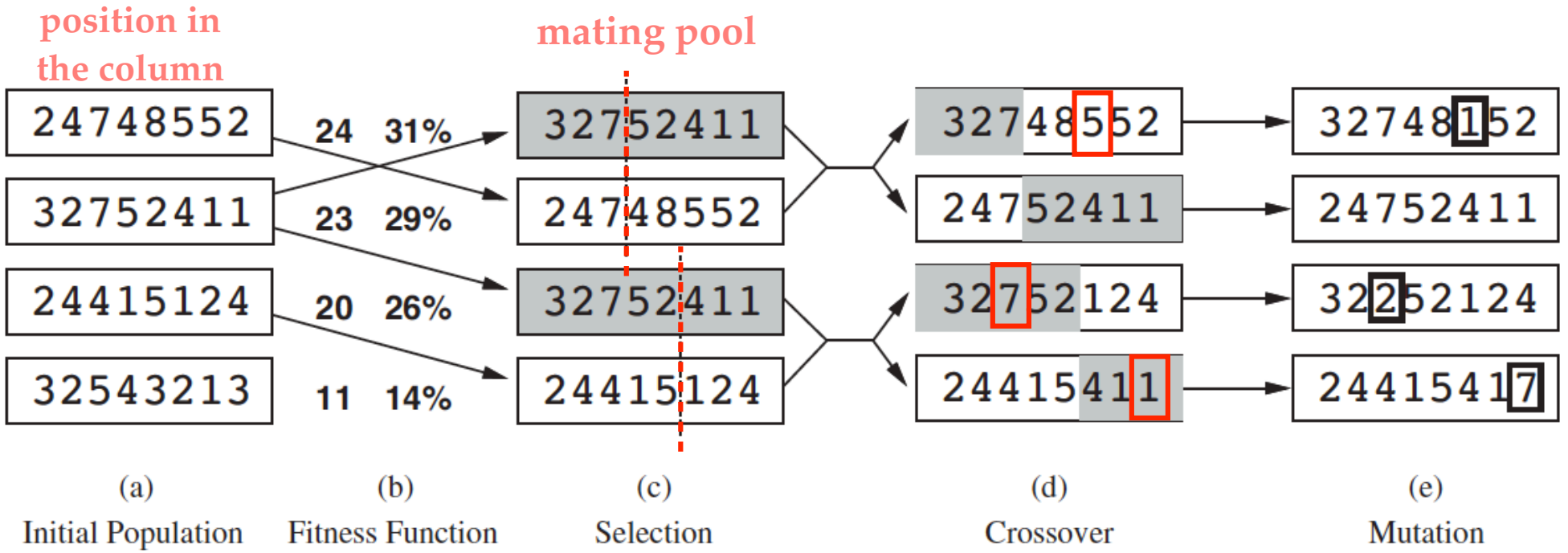
Genetic Algorithms

- 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
- **State**: a string over a finite alphabet (an **individual**)
- Start with ***k*** randomly generated states (a **population**)
- **Fitness function** (= heuristic objective function)
  - **Higher fitness** values for **better states**
- **Select** individuals for next generation based on **fitness**
  - $p(\textit{individual in next gen.}) = (\textit{individual fitness}) / \sum (\textit{population fitness})$
- **Crossover** fit parents to yield next generation (**off-spring**)
- **Mutate** the offspring randomly with some **low probability**

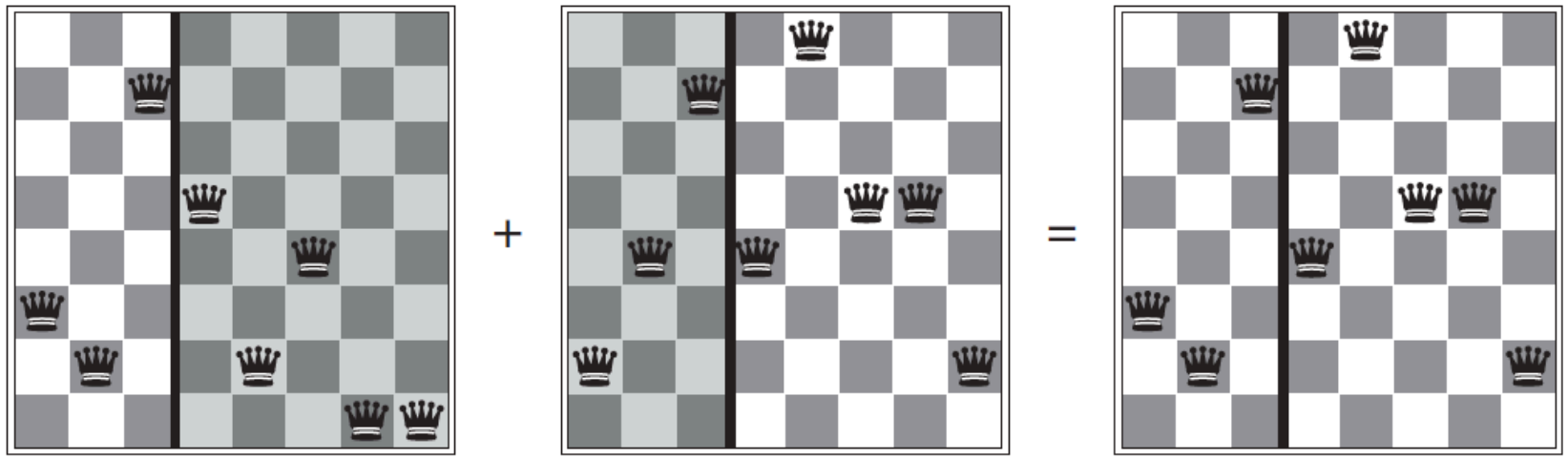
- **Simple Genetic Algorithm (SGA)** [1975-1990]
  - **Encoding**
    - Solution candidates are **encoded** into **chromosomes**
    - Chromosomes can be binary, integral, real-valued, permutations (e.g. TSP), etc.
  - **Initialization**
    - At the beginning, the GA creates a **population** of **chromosomes**
    - The alleles of the chromosomes can be **randomly initialized** or **assigned** based on **prior knowledge**



- **Evaluation**
  - After initialization, the GA applies the **fitness function** to **evaluate** the quality of each chromosome in the current population
- **Selection**
  - Select **promising chromosomes** to have their information passed on to the next generation
- **Recombination**
  - Several **parental chromosomes** are **recombined** to generate **new chromosomes** with characteristics inherited from parents
  - Traditional recombination consists of **crossover** and **mutation**

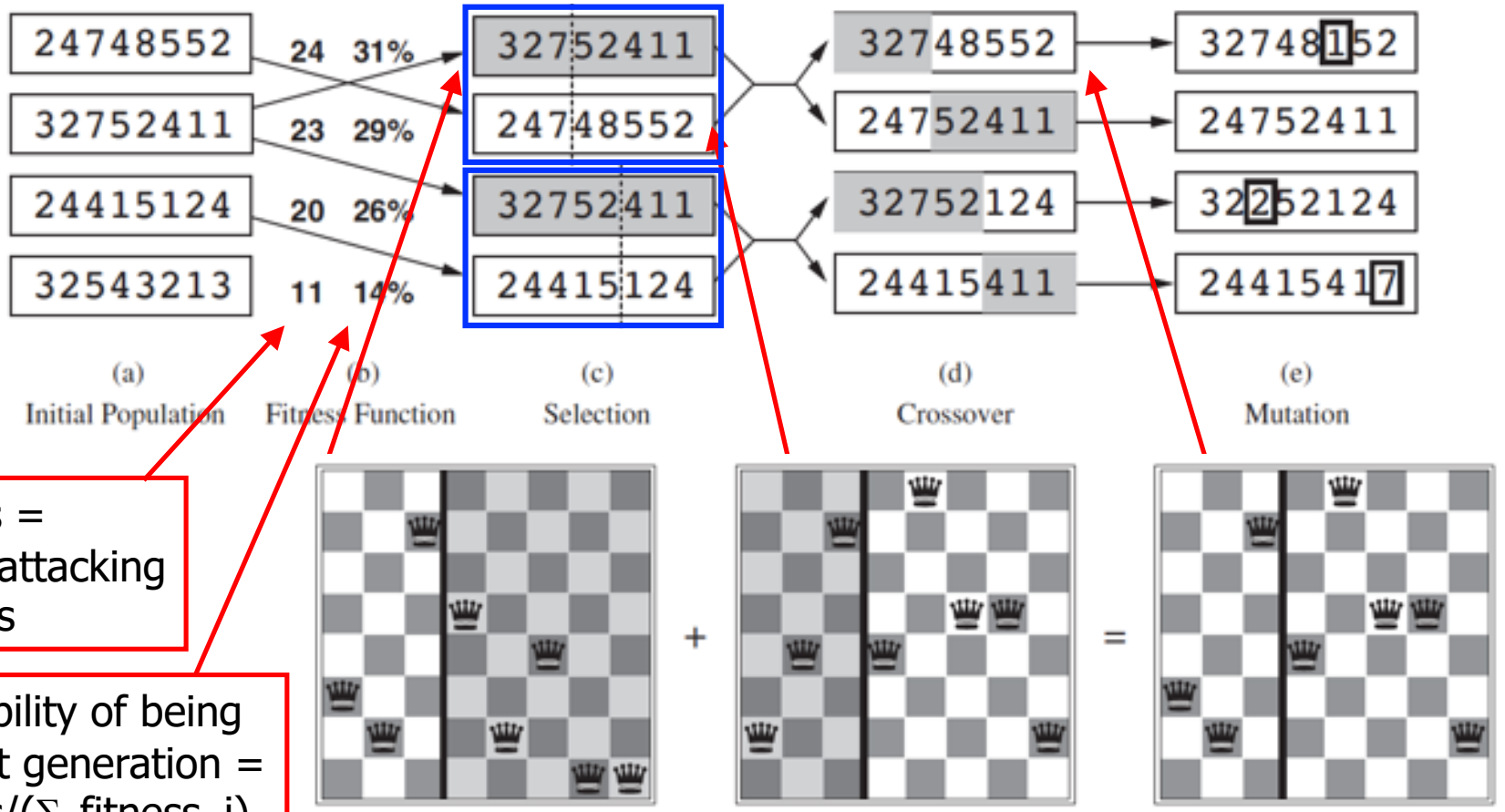


- The genetic algorithm, illustrated for **digit strings** representing *8* **queens states**
- The **initial population** in (a) is ranked by the **fitness** function (#non-attacking queen pairs) in (b), resulting in pairs for **mating** in (c)
- They produce **offspring** in (d), which are subject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



3 2 7 5 2 4 1 1 2 4 7 4 8 5 5 2 3 2 7 4 8 5 5 2

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



fitness = #non-attacking queens

probability of being in next generation = fitness/(Σ_i fitness_i)

How to convert a fitness value into a probability of being in the next generation.

- Fitness function: #non-attacking queen pairs
  - min = 0, max = 8 × 7/2 = 28 C_2^8
- Σ_i fitness$_i$ = 24+23+20+11 = 78
- p(child$_1$ in next gen.) = fitness$_1$/(Σ_i fitness$_i$) = 24/78 = 31%
- p(child$_2$ in next gen.) = fitness$_2$/(Σ_i fitness$_i$) = 23/78 = 29%; etc

```
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
```

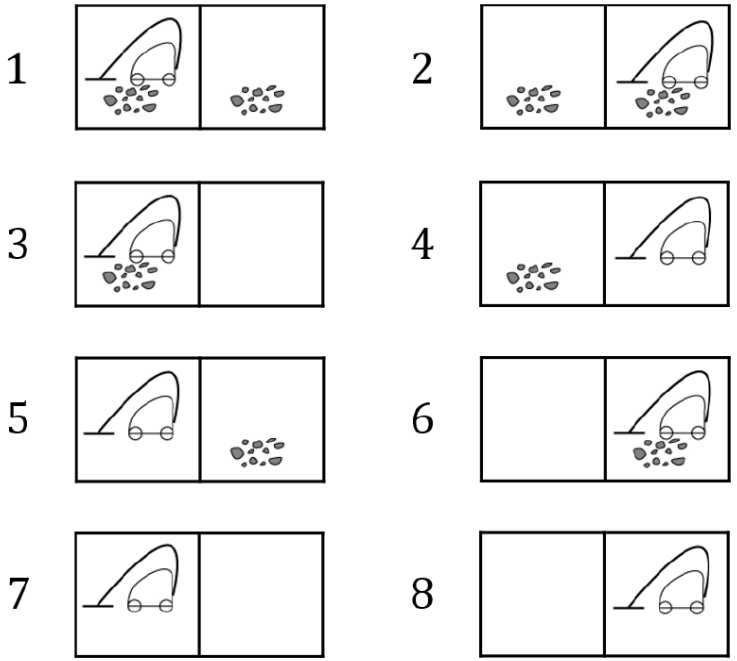
```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

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

- In this more popular version, each **mating** of two parents produces only **one offspring**, not two.

3. Searching with Non-deterministic Actions

- Vacuum world
  - Actions = {LEFT, RIGHT, SUCK}
- SUCK in an **erratic** vacuum world
  - When applied to a dirty square, the action cleans the square and **sometimes** cleans the dirt in an **adjacent square** as well
  - When applied to a clean square, the action **sometimes deposits dirt** on the square



Generalization of State-Space Model

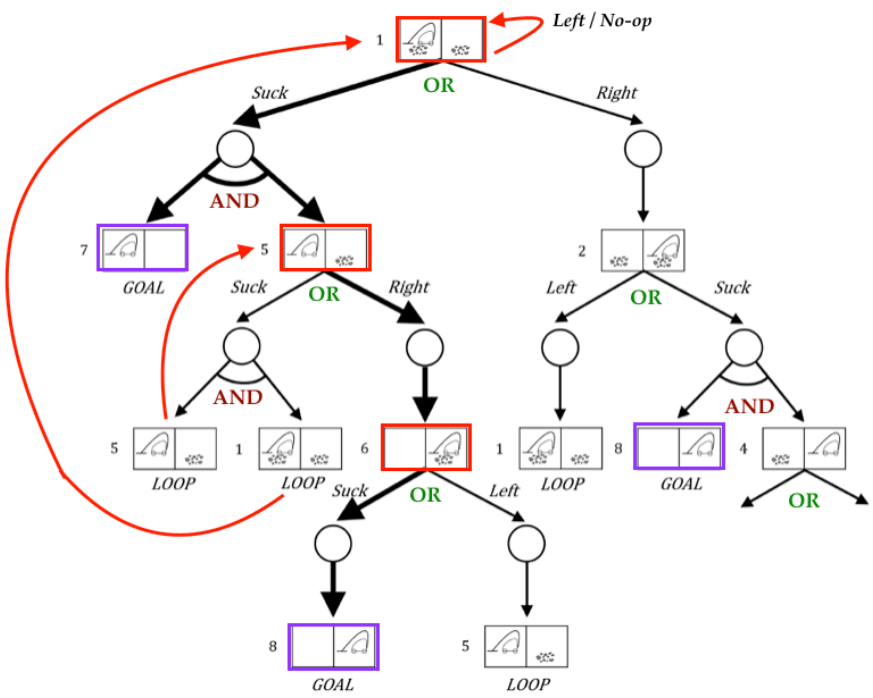
- Generalize the **transition function** to return a set of possible outcomes
  - Old function: $S \times A \rightarrow S$
  - New function: $S \times A \rightarrow 2^S$ [a family of subsets of S, or a family of sets over S]
- Generalize the **solution** to a contingency plan

  if state = s then action-set-1 else action-set-2

  e.g. if Bstate = {6} then Suck else []
- Generalize the search tree to an **AND-OR tree**

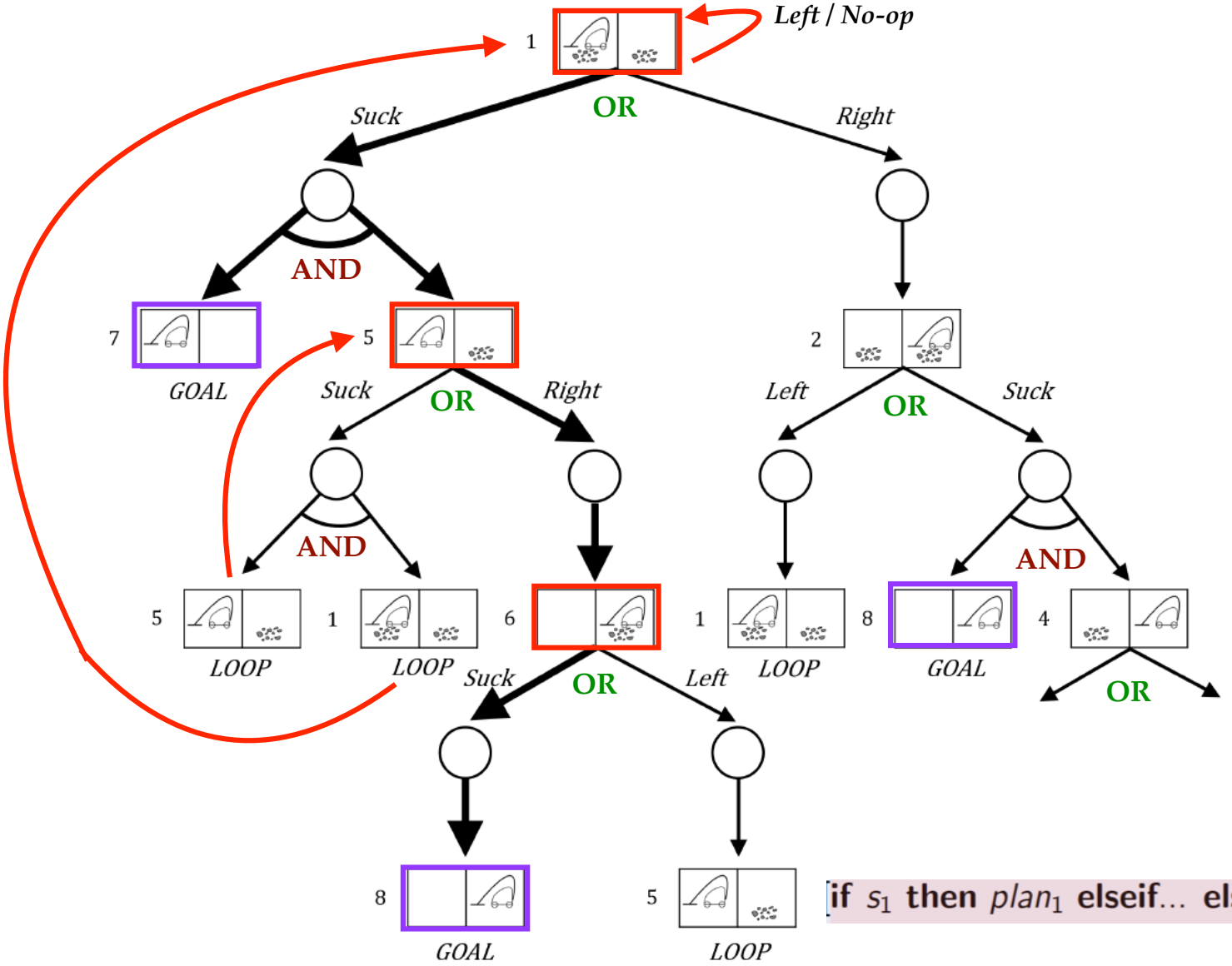
- **AND-OR search tree**
  - **AND** nodes: actions (**circles**)
    - You can end up in **multiple states** as the **result** of an action
    - You have to find a **path** from **all** of these states
    - Each **leaf** of the partial search tree of **AND** node should be a **goal** (e.g. 7, 8)
  - **OR** nodes: states (**squares**)
    - Try each action
    - Any one action can lead to the goal state(s)
  - Need to reach the **goal state** at EVERY **leaf** □

Erratic vacuum world



1st solution: SUCK
2nd solution: SUCK → RIGHT → SUCK

Erratic vacuum world



if s_1 **then** $plan_1$ **elseif**... **elseif** s_{n-1} **then** $plan_{n-1}$ **else** $plan_n$

1_{st} solution: SUCK

2_{nd} solution: SUCK → RIGHT → SUCK

OR-SEARCH(*state*, *problem*, *path*)

```
1  if problem.GOAL-TEST(state)
2        return the empty plan
3  if state is on path return failure
4  for each action in problem.ACTIONS(state)
5        plan = AND-SEARCH(RESULTS(state, action), problem, [state|path])
6        if plan ≠ failure
7              return [action|plan]
8  return failure
```

Recursive calls: OR → AND → OR .....

AND-SEARCH(*state*, *problem*, *path*)

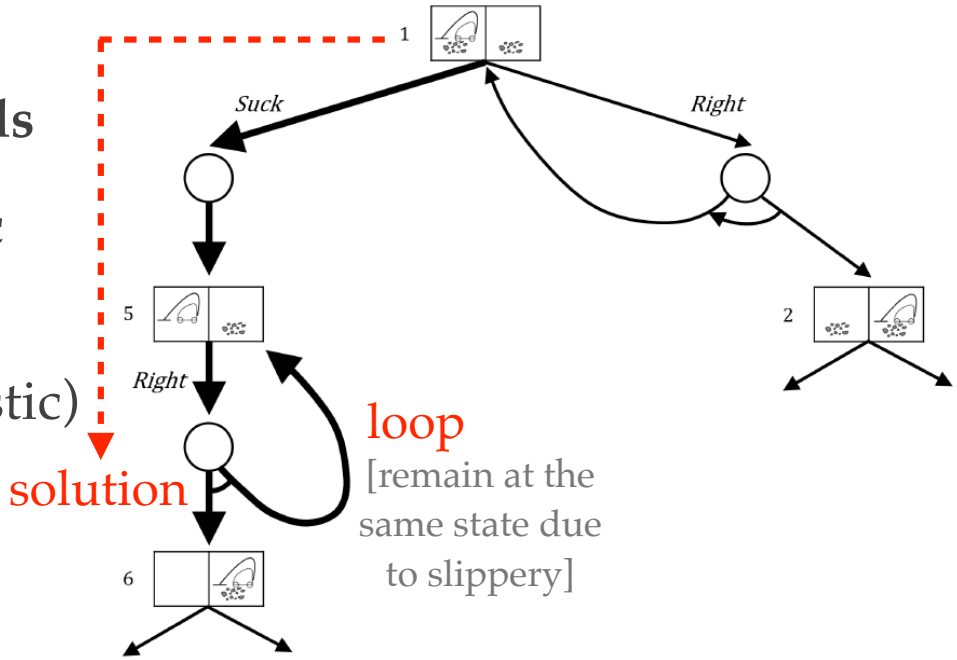
```
1  for each S_i in states
2        plan_i = OR-SEARCH(S_i, problem, path)
3        if plan_i == failure
4              return failure
5  return [if s_1 then plan_1 elseif... elseif s_{n-1} then plan_{n-1} else plan_n]
```

[solution]

e.g. if Bstate = {6} then Suck else []

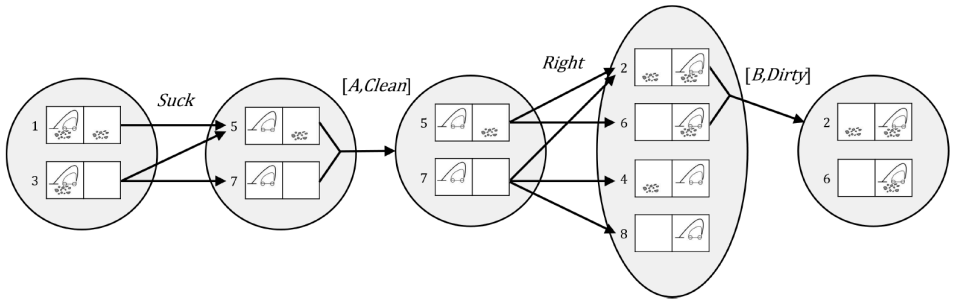
Keep Trying Or Not

- The **slippery** vacuum world (non-deterministic)
  - Identical to the ordinary vacuum world except movement actions **sometimes fails**
- Results in a **cyclic search graph** and a **cyclic solution**
  - Causes of **failure** (due to non-deterministic)
    - **If stochastic** (success with certain probability) ⇒ **keep trying**
    - If **unobservable property** (partial observable, e.g. can't move into a room due to room closed) ⇒ **stop trying** after a certain number (limit) of trials



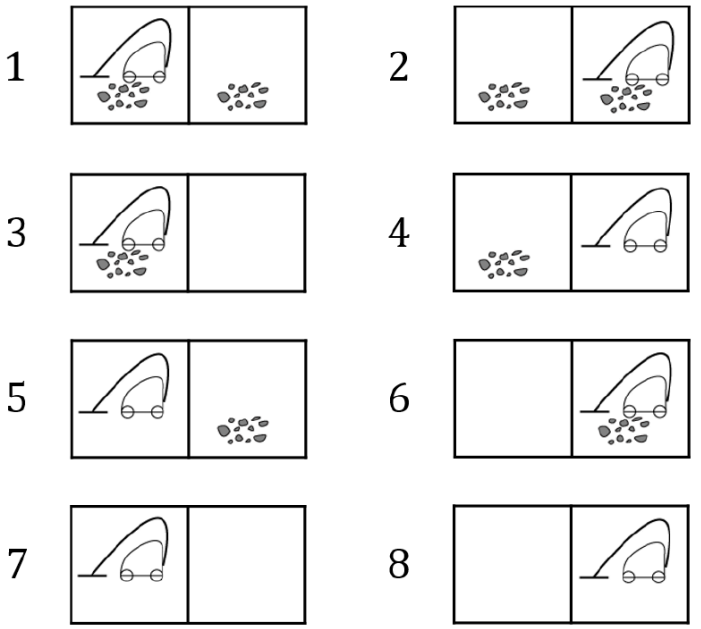
4. Searching with Partial Observations

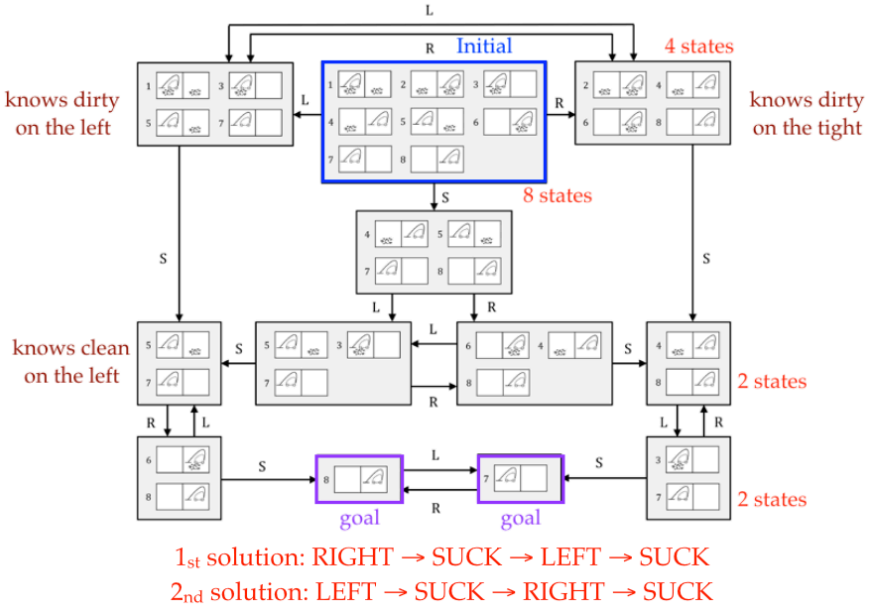
- The agent does not always know its state!
- **Belief state** [set of states]
  - **Represent** the agent's **current belief** about the **possible states** it might be in
  - **Search** for a **sequence of belief states** that leads to a **goal**
- **Example**



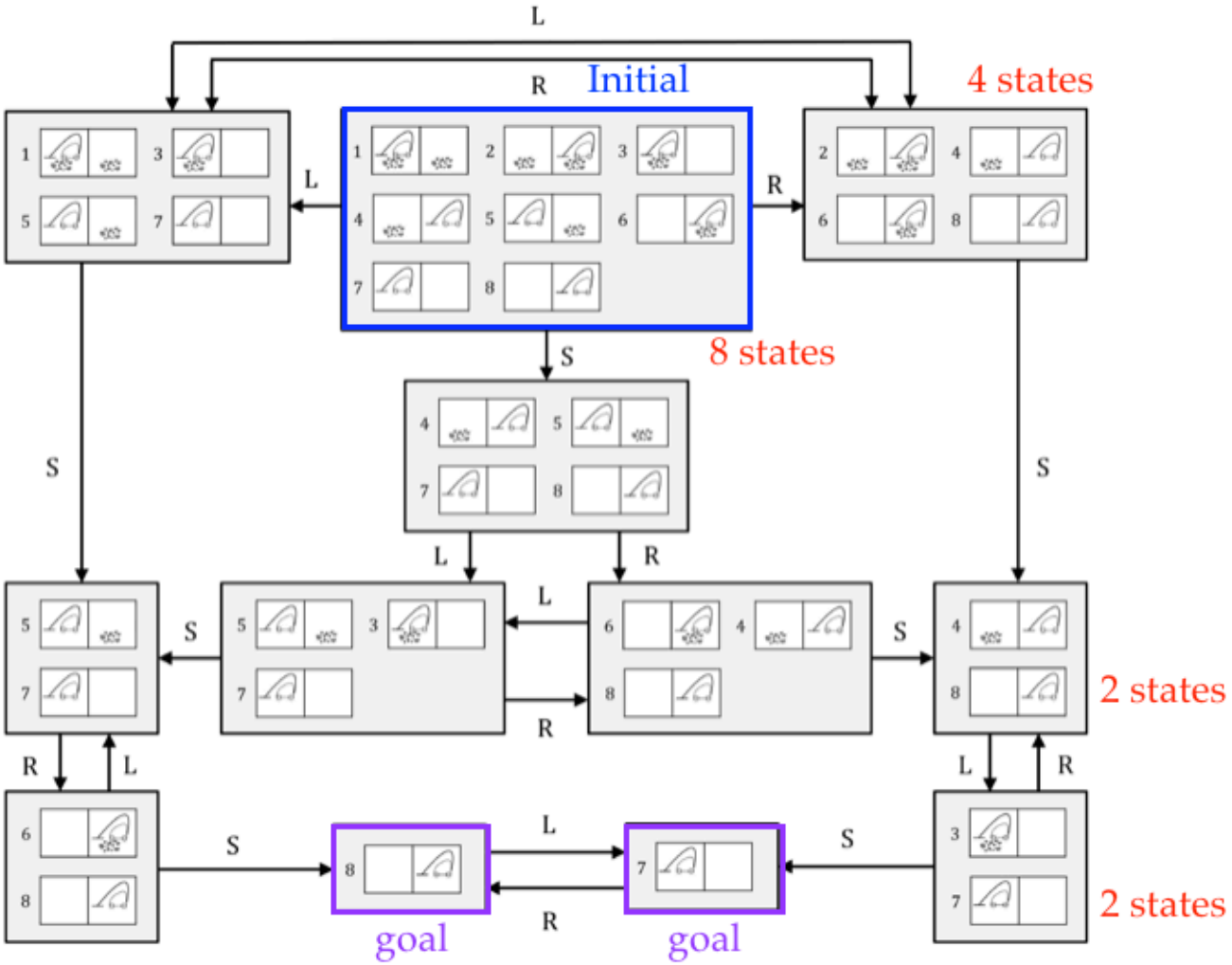
  - A robot can be used to build a map of a **hostile environment**. It will have **sensors** that allow it to "see" the world

- Searching with **no observation** (**sensor-less**) in the vacuum world
  - Assume **not erratic** vacuum world
  - Assume belief states are the same but **no location or dust sensors**
  - Initial belief states: {1, 2, 3, 4, 5, 6, 7, 8}
  - After [**RIGHT**(R)], belief states: {2, 4, 6, 8}
  - After [RIGHT(R), **SUCK**(S)], belief states: {4, 8}
  - After [RIGHT(R), SUCK(S), **LEFT**(L)], belief states: {3, 7}
  - After [RIGHT(R), SUCK(S), LEFT(L), **SUCK**(S)], belief states: {7}, which is **goal**





Belief States Transitions in **Sensor-less** Vacuum World



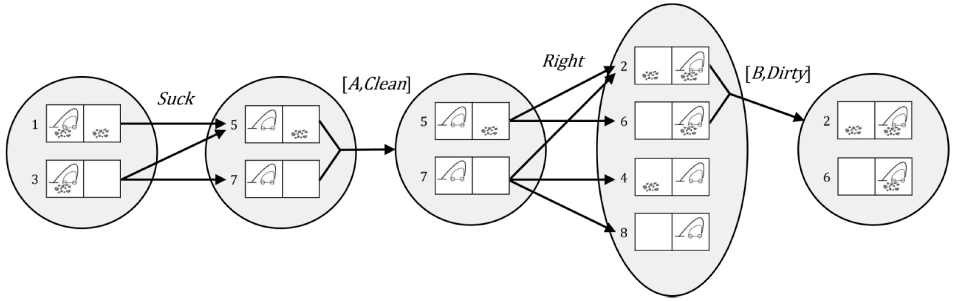
Each box is a belief states

1_{st} solution: RIGHT → SUCK → LEFT → SUCK

2_{nd} solution: LEFT → SUCK → RIGHT → SUCK

Sensor-less Search (No Observations)

- Search in **belief state space**, the **solution** is a **sequence**, even if the environment is **non-deterministic**!



- The underlying **physical problem (*P*)** is defined by

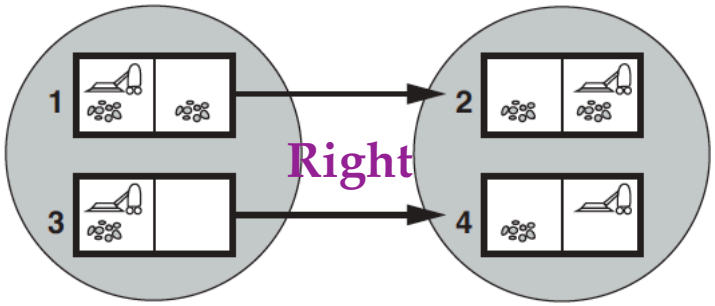
$$\{Actions_p, Result_p, Goal\text{–}Test_p, Step\text{–}Cost_p\}$$

- The corresponding **sensor-less problem**
  - **Belief state**
    - Every possible **set** of **physical states**
    - If N **physical states**, number of **belief states** can be 2^N
  - **Initial state**
    - Typically the set of **all states** in P

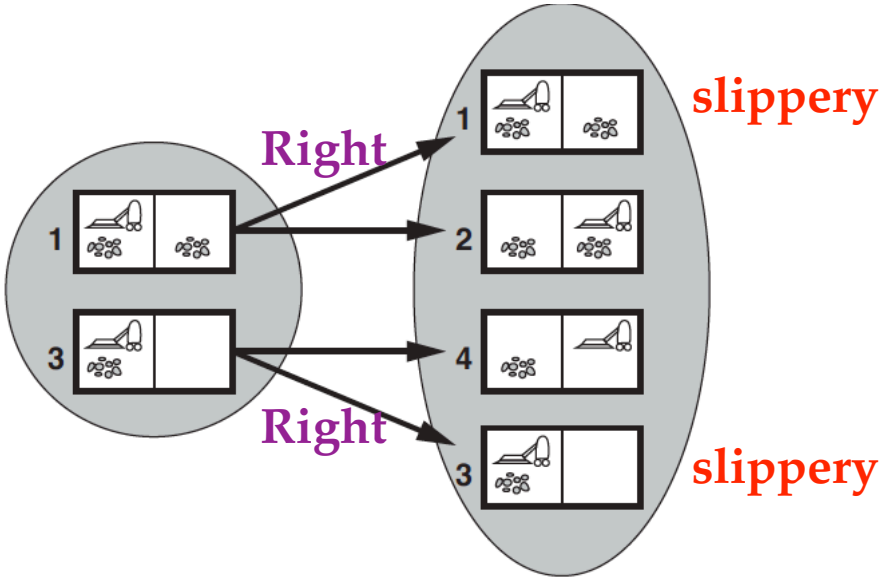
- **Actions**
  - Suppose the agent is in belief state $b = \{s_1, s_2\}$, but $Actions_p(s_1) \neq Actions_p(s_2)$; then the agent is **unsure** of which actions are **legal**
  - If we assume that **illegal actions have no effect on the environment**
    - It is safe to take the **union** of all the actions in any of the physical states in the current belief state b: $Actions(b) = \cup_{s \in b} Actions_P(s)$
  - If an **illegal action might be the end of the world**
    - It is safer to allow only the **intersection**, i.e., the set of actions **legal in all the states**

- **Transition model**
  - The process of generating the **new belief state** after the action is called the **prediction step**, $b' = Result_p(b, a)$
  - For **deterministic actions**, the set of states that might be reached is
    - $b' = Result(b, a) = \{s' \mid s' = Result_p(s, a) \text{ and } s \in b\}$
    - b' is **never larger** than b
  - With **nondeterminism**, we have
    - $b' = Result(b, a) = \{s' \mid s' \in Result_p(s, a) \text{ and } s \in b\}$
      $= \cup_{s \in b} Results_P(s, a)$
    - b' **may be larger** than b

- Transition of belief states
  - Transition with a **deterministic** action **right** [(a)]
    - The **number** of **belief states** usually **decreases** or **remains the same** after a **deterministic** action
  - Transition with a **non-deterministic** action **right** [(b)]
    - The **number** of **belief states** usually **increases** after a **non-deterministic** action

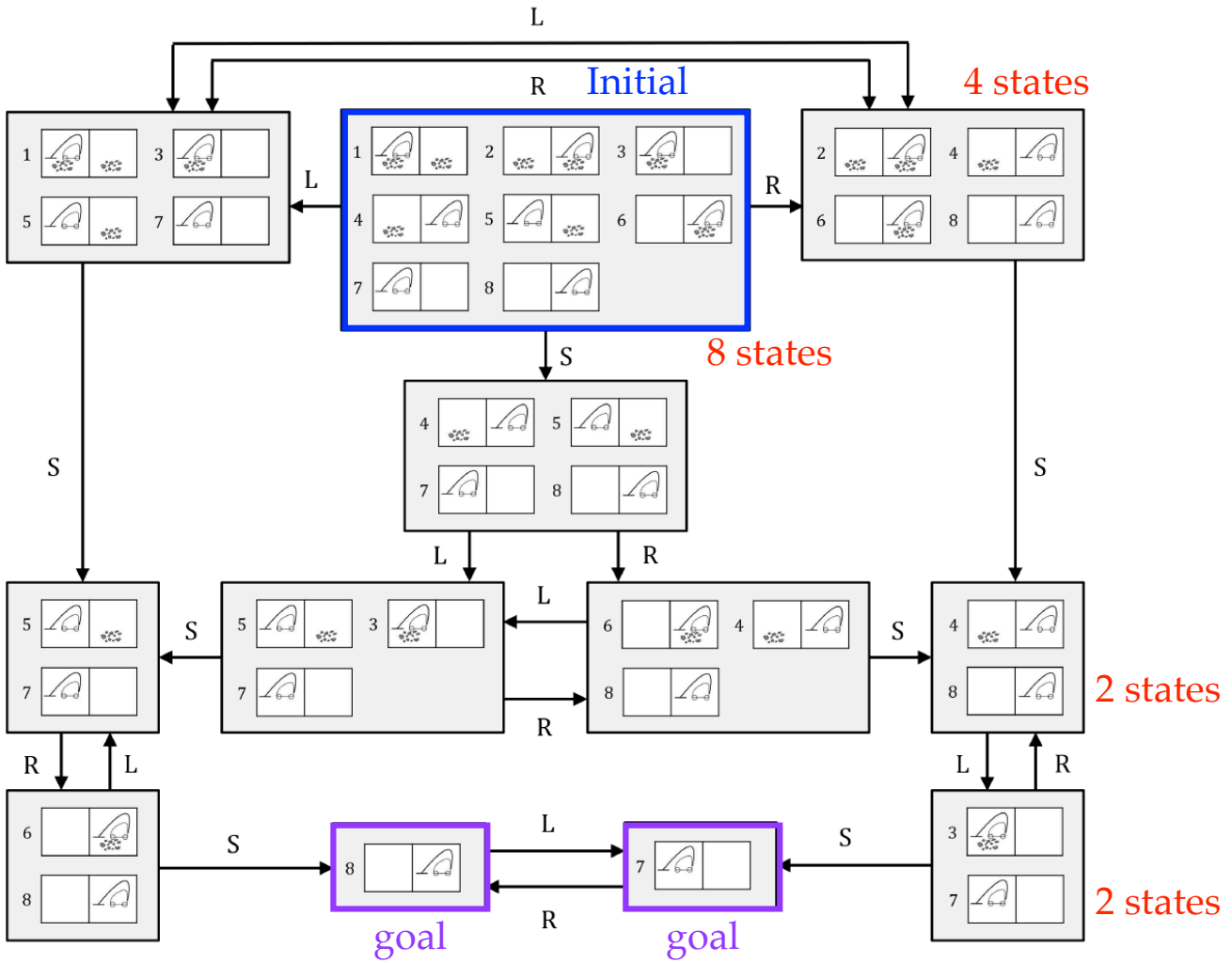


(a) Predicting the next belief state with a **deterministic** action.



(b) Prediction for the same belief state with a **non-deterministic** action in the **slippery** version, **(Slippery-RIGHT)**

- **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-TESTp***
- **Path cost**
  - 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



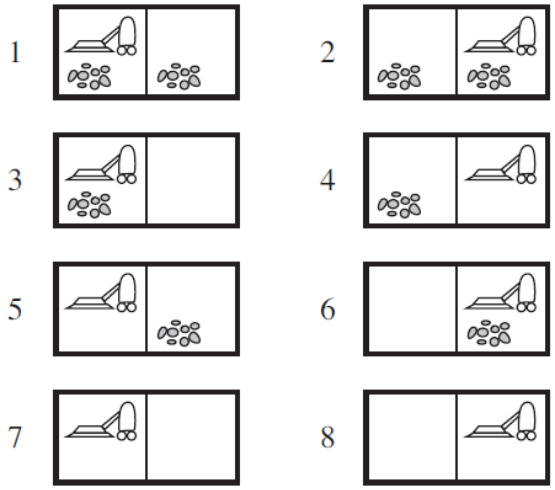
1_{st} solution: RIGHT → SUCK → LEFT → SUCK

2_{nd} solution: LEFT → SUCK → RIGHT → SUCK

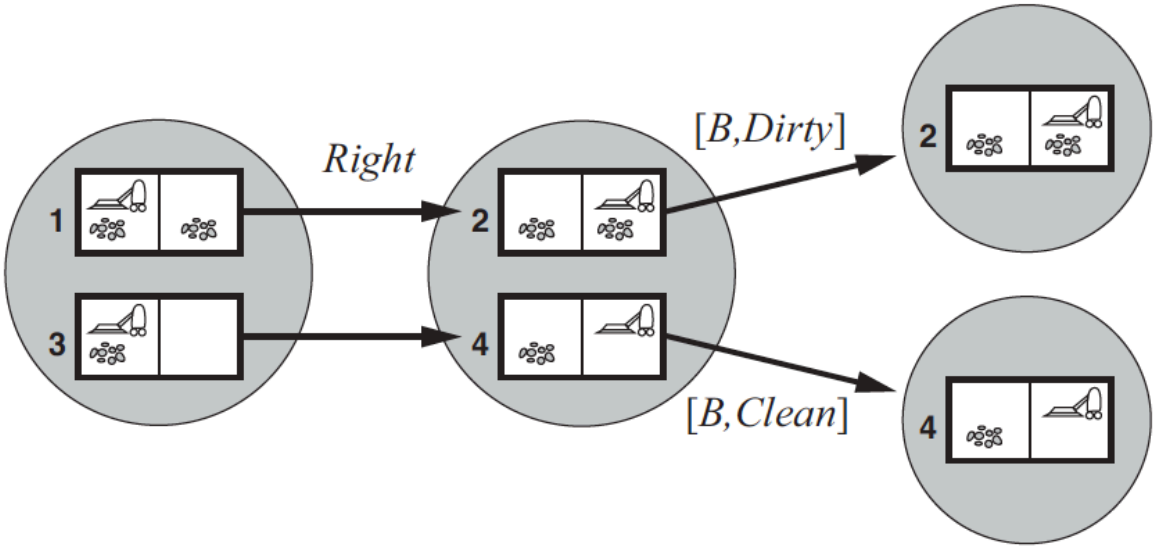
- The **reachable** portion of the belief-state space for the **deterministic**, **sensor-less** vacuum world
- Each **shaded box** corresponds to a **single belief state**
- At any given point, the agent is in a particular belief state but does not know which physical state it is in
- The **initial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

Searching with Observations

- Many problems require **sensors**
  - ***Percept*(s)** or ***Percepts*(s)** function
- Vacuum world example
  - Location sensor
  - Current location dirt sensor (cannot detect dirt in other square)
    - e.g., $Percept(s_1)$ = [A, Dirty]
- **Observability**
  - Sensor-less problems → ***Percepts(s)*** = **null** for all ***s***
  - Fully observable → ***Percepts(s)*** = ***s'*** for every ***s***



- **Example**
  - If you get [A, Dirty] you could be in {1, 3}
  - Result({1, 3}, Right) is {2, 4}



  - Now
    - If you see (observe) [B, Dirty] you are in {2}
    - If you observe [B, Clean] you are in {4}

- **3-stage transition model**
  - 1. **Prediction stage**
    - Same as **sensor-less** (no observation)

$$\hat{b} = Predict(b, a)$$

  - 2. **Observation prediction stage**
    - Determine the **set of observations** in the predicted belief states

$$Possible\text{-}Percepts(\hat{b}) = \{o \mid o = Percept(s), s \in \hat{b}\}$$

- 3. **Update stage**
  - Determine b_o, a subset of $\hat{b}$ which produces observation o

shrink belief $b_o = Update(\hat{b}, o) = \{s \mid Percept(s) = o, s \in \hat{b}\}$

set size

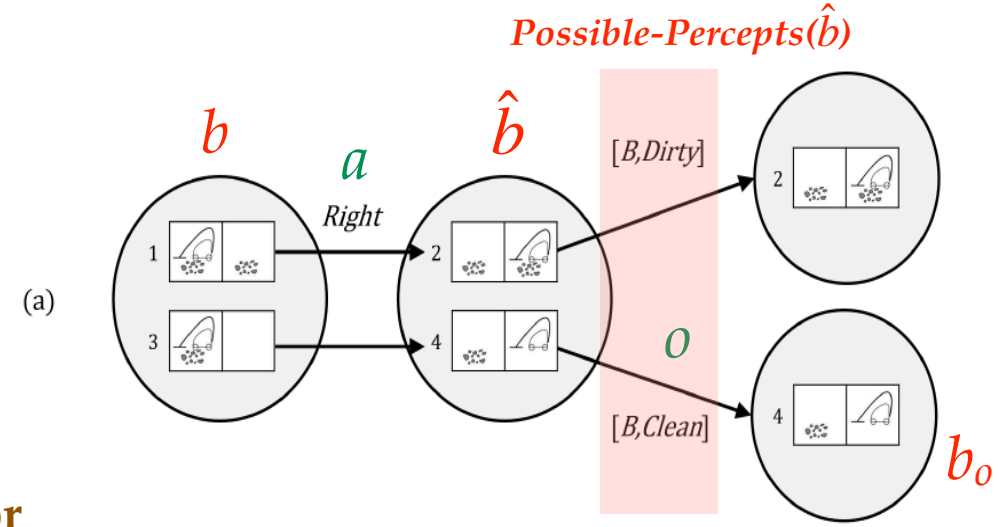
use o to update $\hat{b}$

- $Results(b, a) = \{b_o \mid b_o = Update(Predict(b, a), o)$ and $o \in Possible\text{-}Percepts(Predict(b, a))\}$

$\hat{b}$

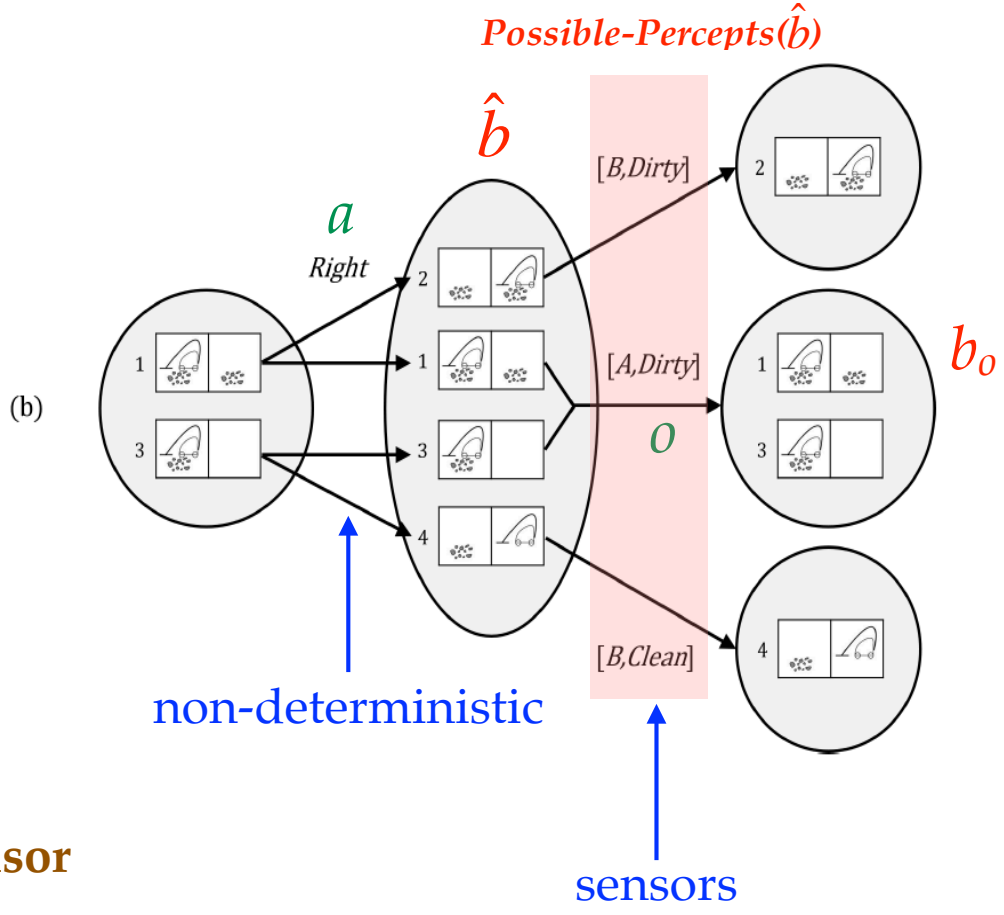
Vacuum World with Local Sensing

- **Sensor** senses **location** and **dirt**
- (a) In the **deterministic world**, **Right** is applied in the initial belief state {1,3}, resulting in a new belief state with **two possible physical states** {2,4}; for those states, the possible percepts are [B, Dirty] and [B, Clean], leading to **two belief states** ({2}, {4}), each of winch is a singleton



Deterministic + sensor

- (b) In the **slippery world**, **Right** is applied in the initial belief state {1,3}, giving a new belief state with **four physical states** {2,1,3,4}; for those states, the possible percepts are [A, Dirty], [B, Dirty], and [B, Clean], leading to **three belief states** ({2}, {1,3}, {4})as shown

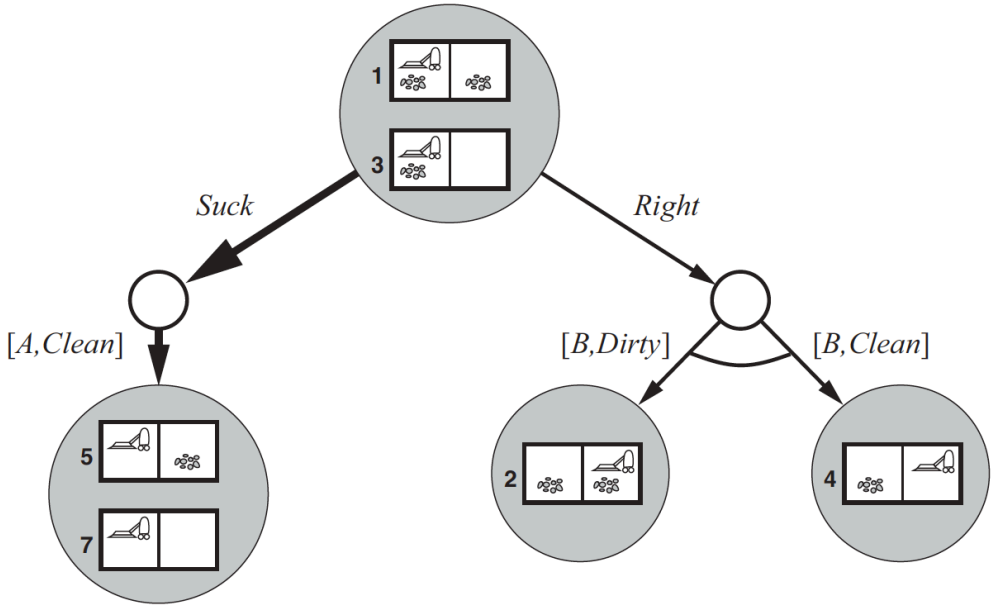


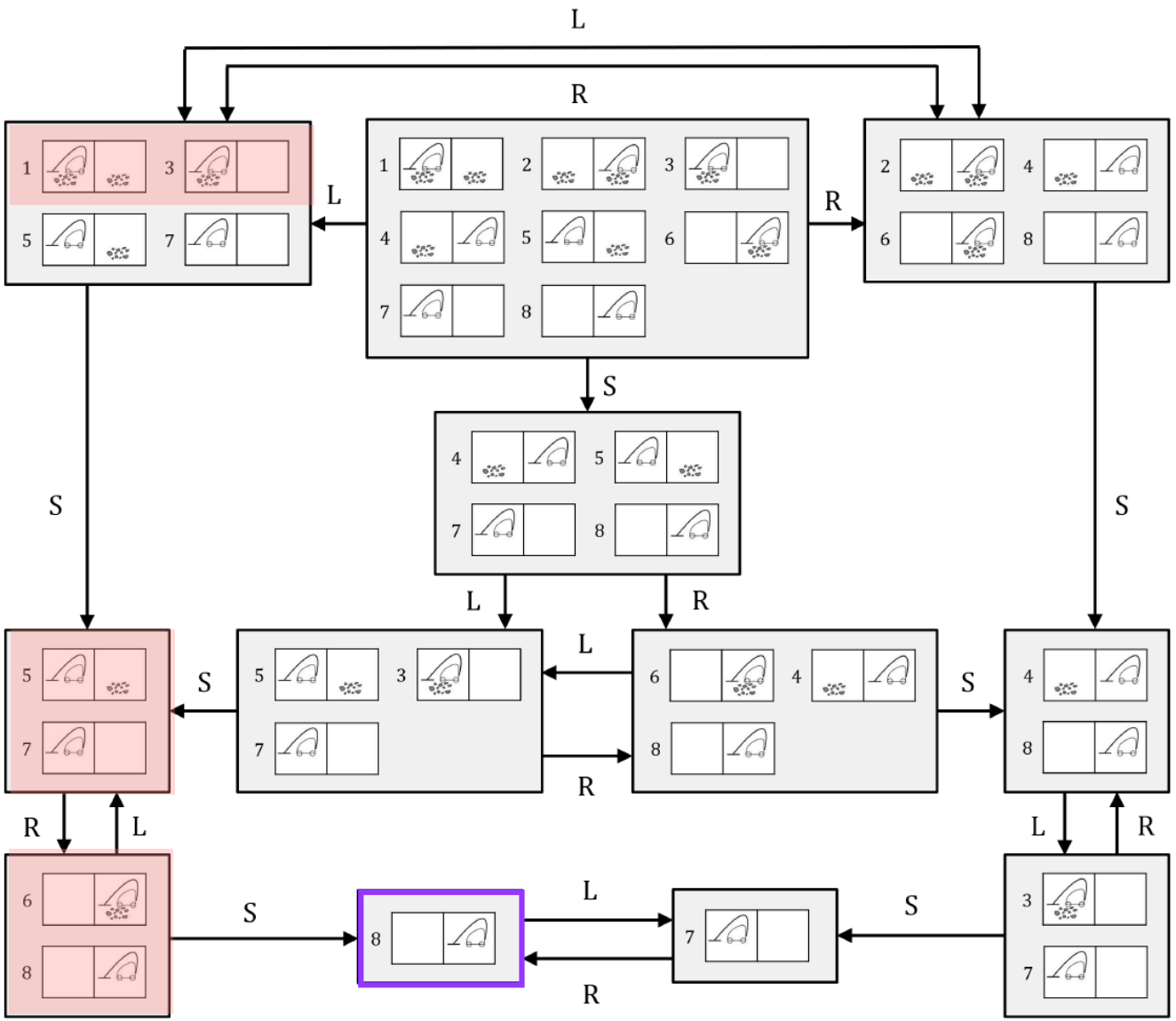
Non-deterministic + sensor

AND-OR Solution

- The preceding **RESULTS** function for a **nondeterministic belief-state problem** can be derived 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
- Given an initial percept [A, Dirty], the solution = **{Suck, Right, if Bstate = {6} then Suck else []}**

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.





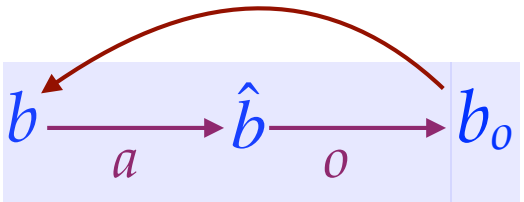
Given an initial percept [A, Dirty],
the solution = **{Suck, Right, if Bstate = {6} then Suck else []}**

Agents for Partially Observable Environment

- An agent in a **partially observable environment** must **update** belief state from **percept**

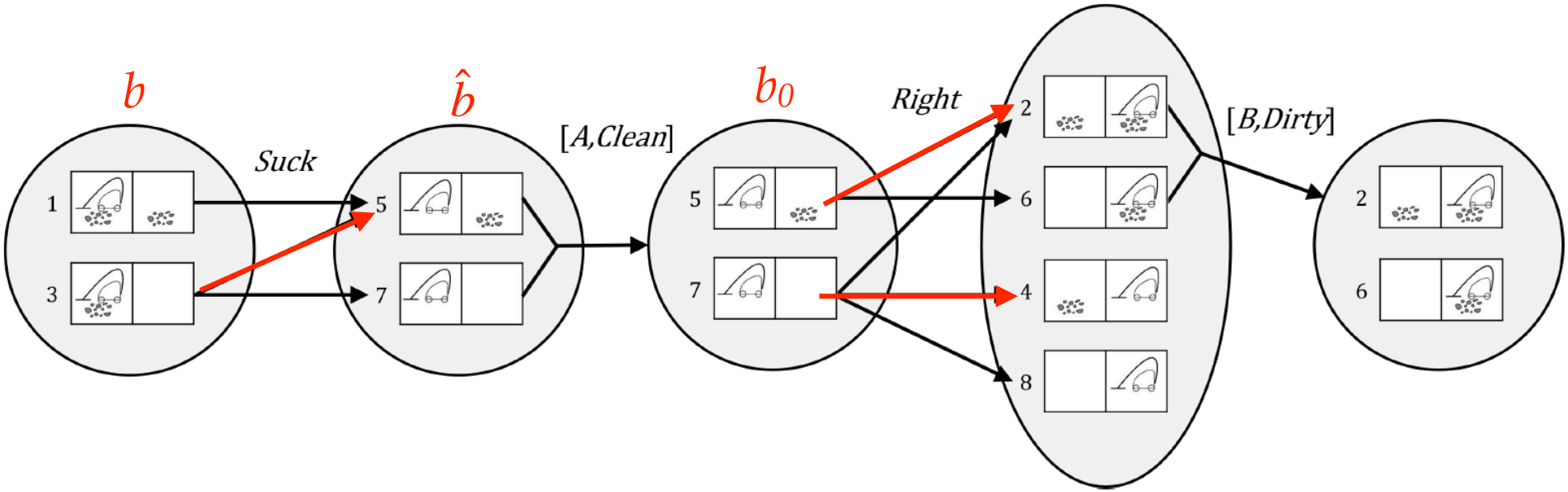
$$b_o = Update(\hat{b}, o) \quad \text{where } \hat{b} = Predict(b, a)$$

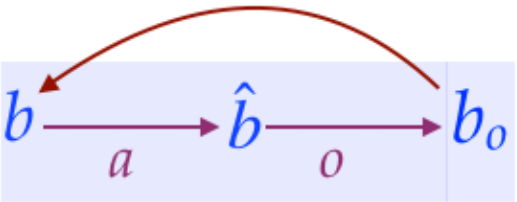
use o to update $\hat{b}$



  - The agent is only looking at the **current *o* (percept)** not the entire history
  - This is **recursive state estimation**, i.e., keep estimating the **belief states** and **AND-OR searching** for solutions to reach goal states

- **Kindergarten** vacuum world with **local sensing**
  - Any square may **become dirty at any time** unless the agent is actively cleaning it at that **moment**

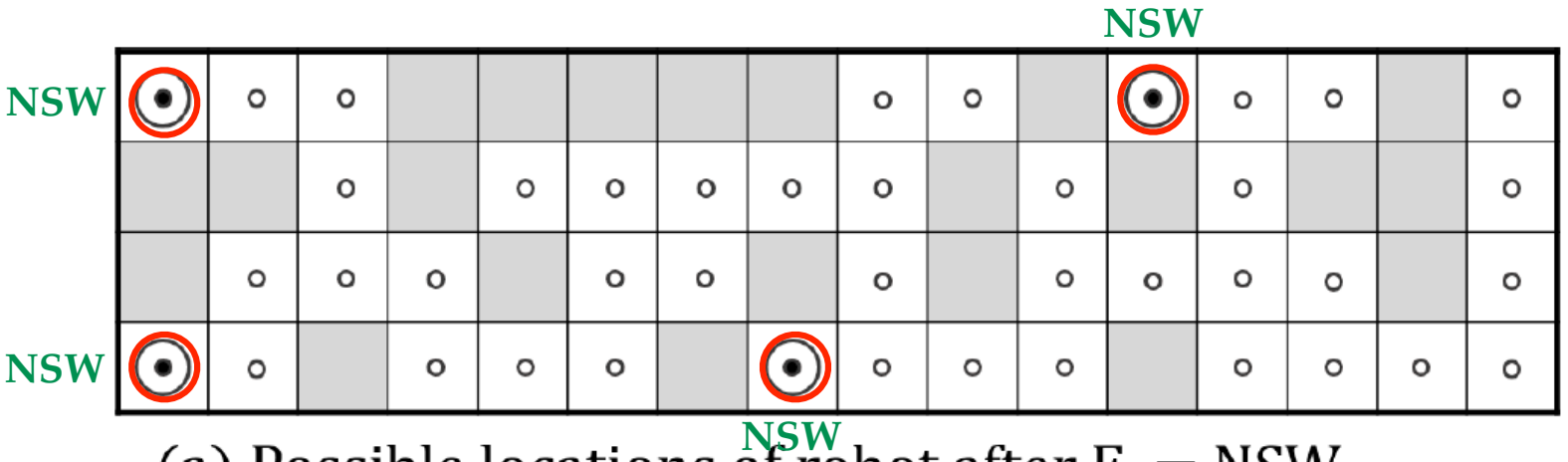




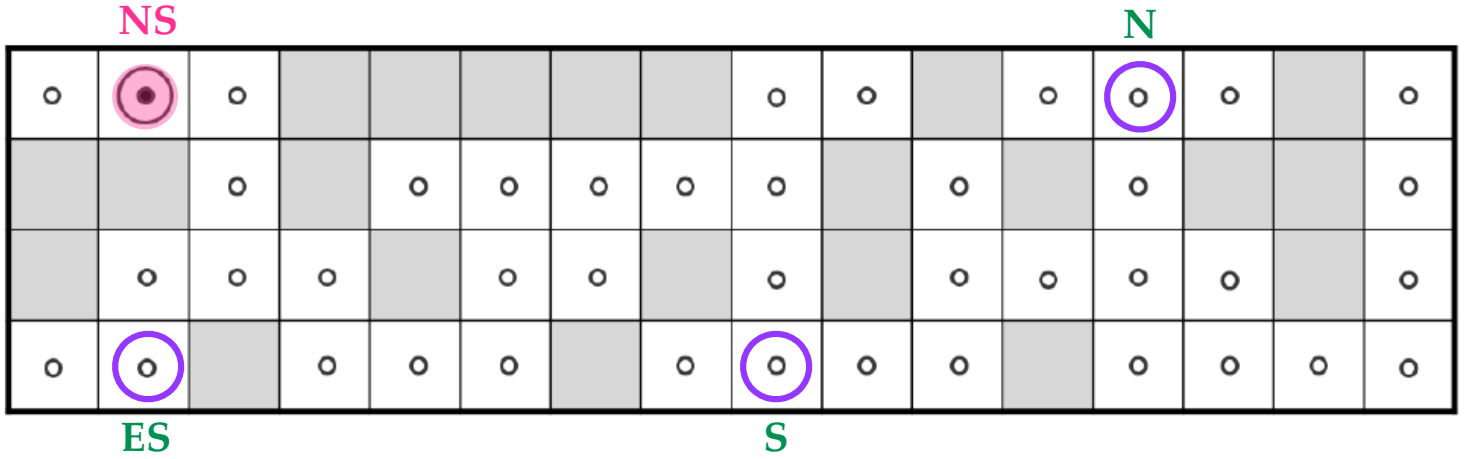
Localization in a Maze

- Given a **map** of the world and a sequence of **percepts** and **actions**, a robot with the task of **localization** working out **where it is**
- The robot is placed in the **maze-like environment** and is equipped with **four sonar sensors** that tell whether there is an **obstacle**—the **outer wall** or a **black square** in the figure
- Assume that the sensors give **perfectly correct data**, and that the robot has a **correct map** of the environment
- But 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**

Map is known, 4 sensors, EWNS = (East, West, North, South)



(a) Possible locations of robot after E_1 = NSW



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

- *Update(Predict(Update(b, NSW),Move), NS)*

5. Online Search Agents and Unknown Environments

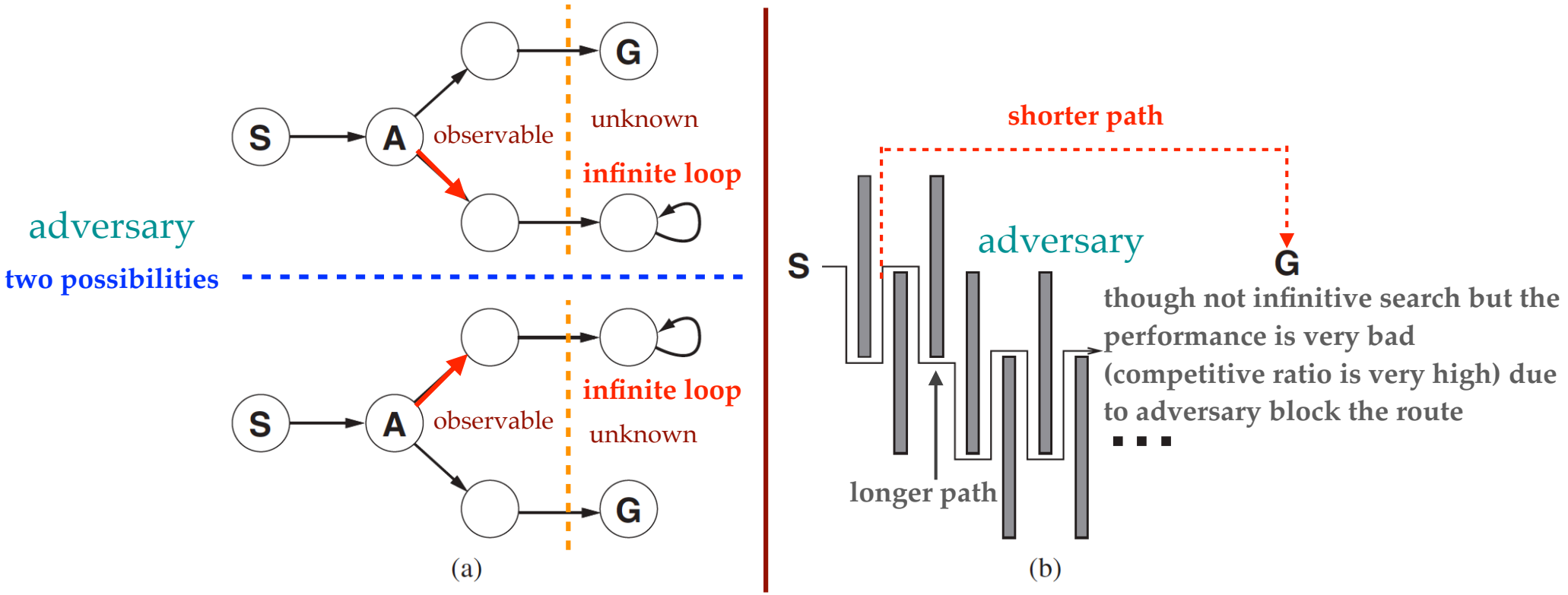
- Partial observable problem → unknown environment problem (online search)
- An online search problem must be solved by an agent **executing actions**, rather than by pure computation
- Assume a **deterministic** and **fully observable** environment
- Acquire **percepts** from sensors
- The agent cannot determine ***RESULT*(*s*, *a*)** except by actually being in ***s*** and doing ***a*** (has to perform an action to know the outcome)
- **Examples**
  - Web search, autonomous vehicle

- **Agent's objective**: reach a goal state while **minimizing cost**
  - The **total path cost** of the path that the agent **actually travels** (**actual cost**)
  - The path cost of the path the agent **would follow** if it knew the search space in advance - i.e., the **actual shortest path** (**minimum cost**)

$$\textbf{Competitive ratio} = \frac{\text{actual cost}}{\text{minimum cost}} \geq 1$$

  - We'd like to **minimize** this

- If all **actions** are **reversible**
  - **Online-DFS** visits every states exactly **twice** in the **worst case** with **enough memory**
- If **some** actions are **irreversible**
  - Lead to a state from which **no action leads back** to the previous state
  - Accidentally reach a **dead-end state** from which no goal state is reachable
    - A **small** (or even **finite**!) **competitive ratio** can be difficult to achieve
    - The achievable **competitive ratio** is **infinite** in some eases



(a) Two state spaces that might lead an online search agent into a **dead end**

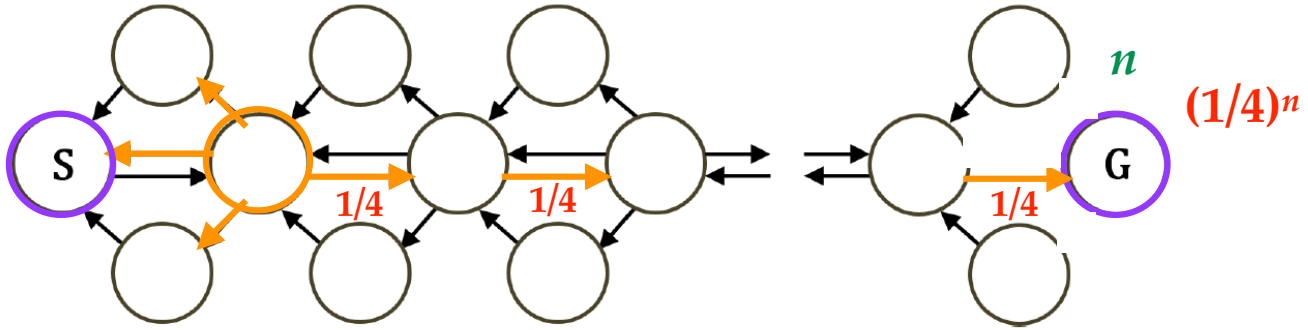
- Any given agent will **fail** in at least one of these spaces

(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 (**adversary search** means **the worst case can be very bad**)

Online Local Search

- Only one or few states are stored (search with **limited memory**)
- **Single-point hill-climbing** gets stuck at a local optimum, causing the **competitive ratio** to be **infinite**
- We may add some **random walk** (like **simulated annealing**), but still can be **inefficient** (**exponential long** in the below example)
- Random walk is **complete** for **finite** state spaces



Random walk

From **S** to **G**
(a) n steps
(b) prob = $(1/4)^n$
(c) times = 4^n

Learning Real-Time A* (LRTA*)

- Augmenting hill climbing with **memory** (store every state - a big problem) **rather than randomness**
- Basic idea: store a **"current best estimate"**, ***H*[*s*]**, of the cost to reach the goal from each state that has been visited
- ***H*[*s*] starts out** being just the **heuristic estimate**, ***h*(*s*)**, and is **updated** as the agent gains experience in the state space
  - ***H*[*s*]**: a **table** of **cost estimates** indexed by **state**, initially empty
    H[*s*]: stores |S| states (very large), use feature domain to reduce |*S*|. Features of the same value are thought to be the same state (define S → feature)
  - ***result*[*s*, *a*]**: a **table** indexed by **state** and **action**, initially empty
    result[*s*, *a*]: stores |*S*| × |*A*| states

function LRTA*-AGENT(s') **returns** an action

inputs: s', a percept that identifies the current state

persistent: $result$, a table, indexed by state and action, initially empty
H, a table of cost estimates indexed by state, initially empty
s, a, the previous state and action, initially null

if GOAL-TEST(s') **then return** $stop$

if s' is a new state (not in H) **then** $H[s'] \leftarrow h(s')$ 

if s is not null

new state, assign h(s') to H[s']

$result[s, a] \leftarrow s'$

$H[s] \leftarrow \min_{b \in \text{ACTIONS}(s)}$ LRTA*-COST($s, b, result[s, b], H$) old state, update H[s]

$a \leftarrow$ an action b in ACTIONS(s') that minimizes LRTA*-COST($s', b, result[s', b], H$)

$s \leftarrow s'$

return a

function LRTA*-COST(s, a, s', H) **returns** a cost estimate

if s' is undefined **then return** $h(s)$

else return $c(s, a, s') + H[s']$

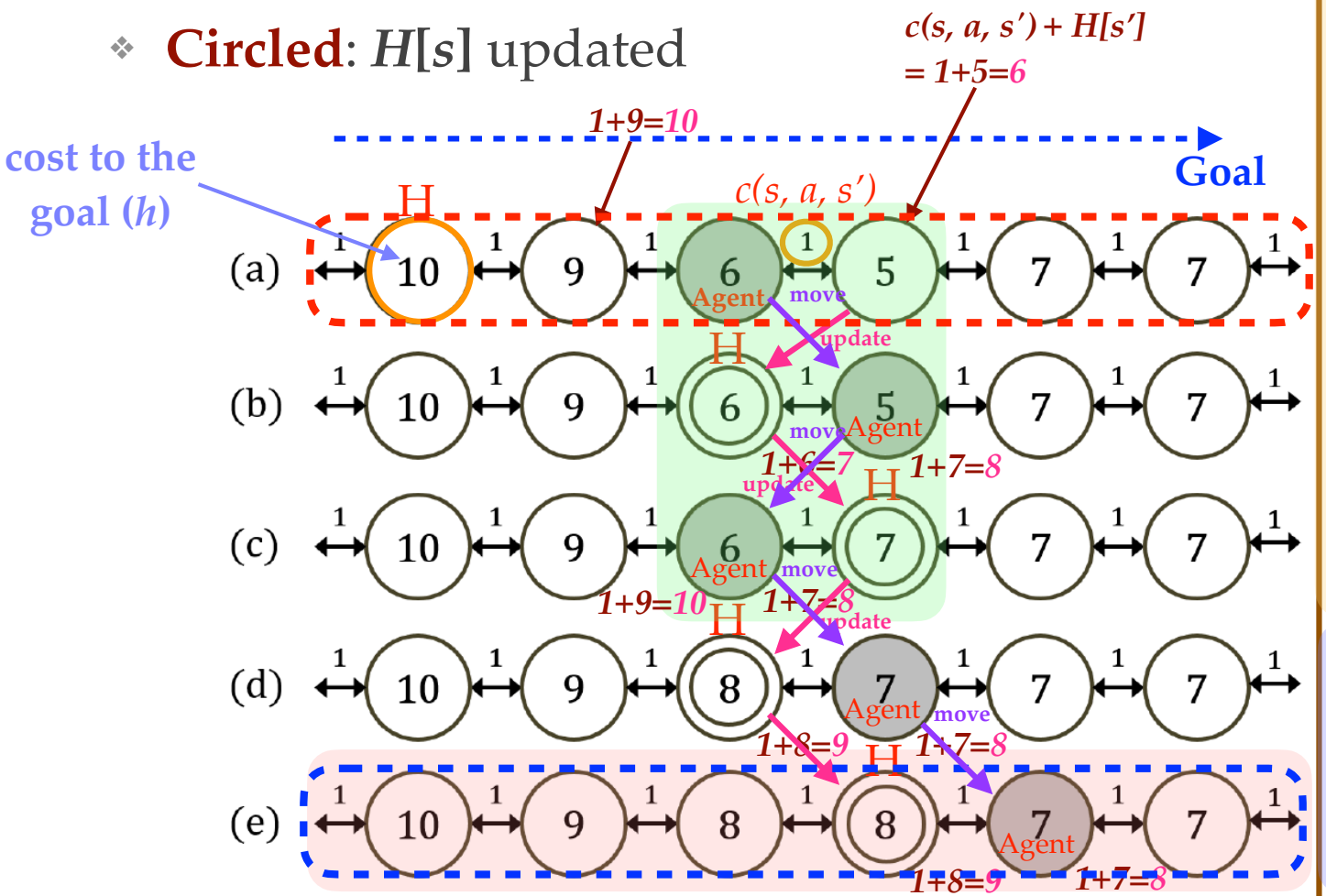
- LRTA* keeps updating ***H[s]***
- LRTA* always chooses the **apparently best action**
- LRTA* works well if **memory are enough** [a big problem]
- **Optimism** under **uncertainty**
  - If an **action** has **never tried** in a state, LRTA* assumes the **least possible cost - *h(s)***, in order to **encourage exploration** of new states

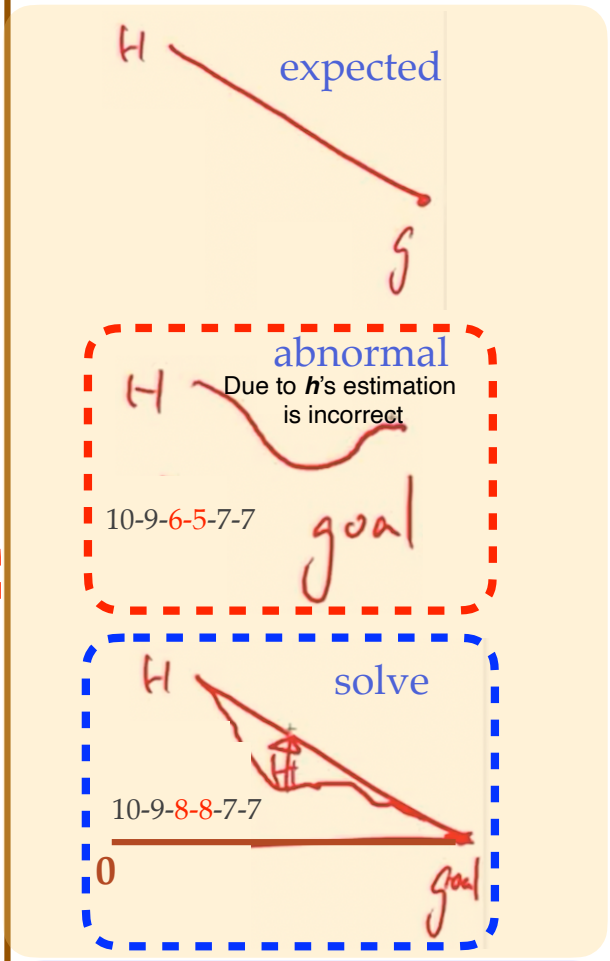
(ignore ***g(s)***)

function LRTA*-COST(s, a, s', H) **returns** a cost estimate
  if s' is undefined (never tried) **then return** $h(s)$
  else return $c(s, a, s') + H[s']$

c(s, a, s′): step cost

- Unlike A*, **LRTA*** is **NOT complete** for **infinite** state spaces
- With ***n* states**, LRTA* guarantees to find **optimum** within **O(n^2)** steps, but usually **much faster**
- Notations
  - **Shaded**: agent's current location
  - **Circled**: ***H*[*s*]** updated





Memory is a critical

H[s]: stores |S| states (very large)

result[s, a] : stores |S| × |A| states

use feature domain to reduce |S|, features of the same value are thought of the same state

6. Summary

- **Steepest descent** is **extremely fast** for simple problems, e.g. easy for $10^6 \times 10^6$ queens problem
- To avoid being trapped at local optima, **SA** adopts **random walk** behaviors
  - Still **quite fast** for simple problems
- Instead of one single state, **GA** adopts **a population of states**
- **AND-OR search** for **non-deterministic** actions

- **Sensor-less** agents performs very well on many real-world problems
  - They are **robust** since they don't rely on the accuracy of sensors
- **Sensors** reduce the size of the set of belief states, and may help agents create a shorter plan
- **On-line search** with limited memory can easily **fail** (adversary argument (worst case) resulting exponential time), but are most popular nowadays
- **LRTA*** works well if **memory (**$|S|, |S \times A|$**) are enough** (may be a problem)