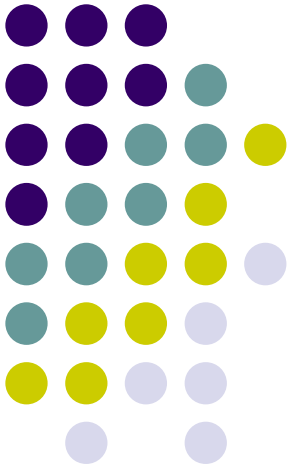
Introduction to AI

Chapter04
Beyond Classical Search

Pengju Ren@IAIR

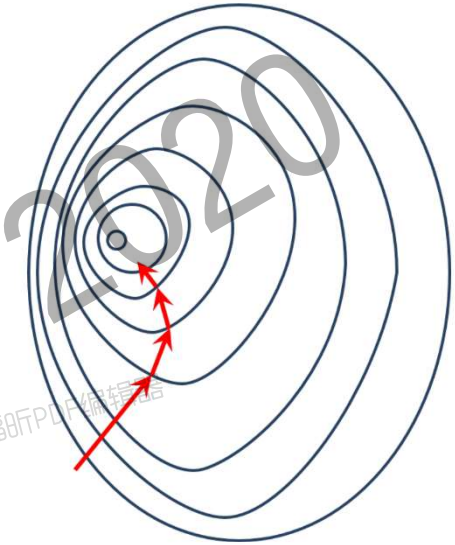


Outline

- Steepest Descent (Hill-climbing)
- Simulated Annealing
- Evolutionary Computation
- Non-deterministic Actions
  - And-OR search
- Partial Observations
  - Sensor-less
  - With Sensors
  - Unknown Environments

Steepest Descent (Ascent)

- A.k.a. **Gradient descent.**
- "Like climbing Everest in thick fog with amnesia."
- Allowing **sideway moves** is usually good, but need to put a limit to prevent infinite loop.

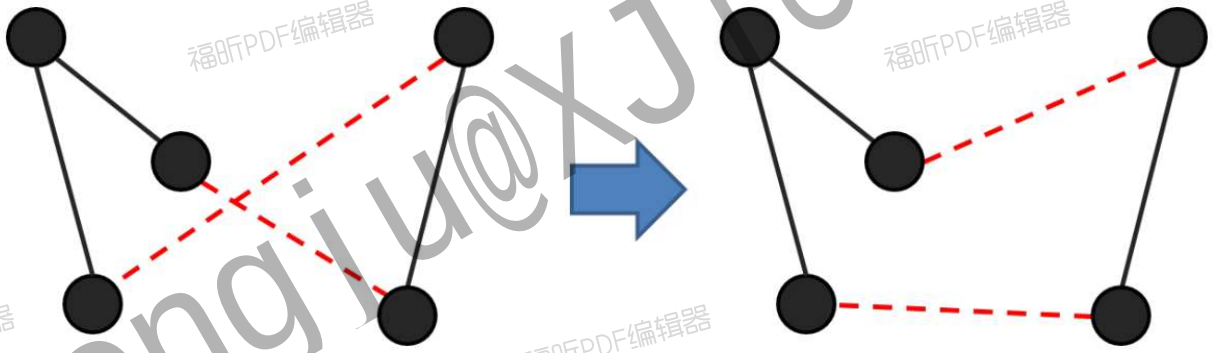


HILL-CLIMBING(*problem*)

```
1  current = MAKE-NODE(problem.initial_state)
2  repeat
3      neighbor = a highest-valued successor of current
4      if neighbor.value ≤ current.value
5          return current.state
6      current = neighbor
```

Example: TSP

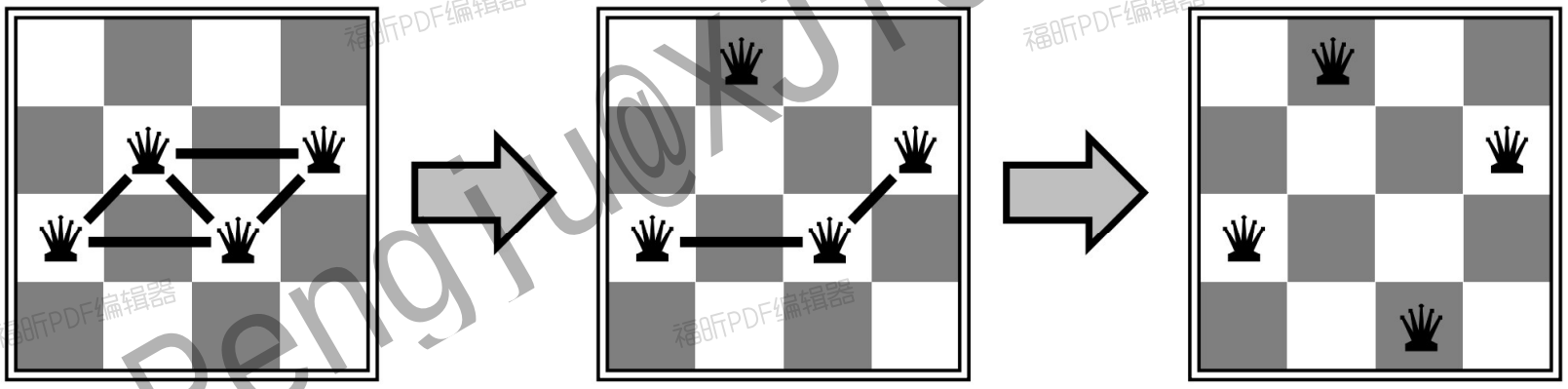
- Begin with any complete tour (random).
- Check if any pairwise exchange (*swap-2*) shorten the tour.
- We can check swap-*k* of course, but it takes $O(n^k)$ time.



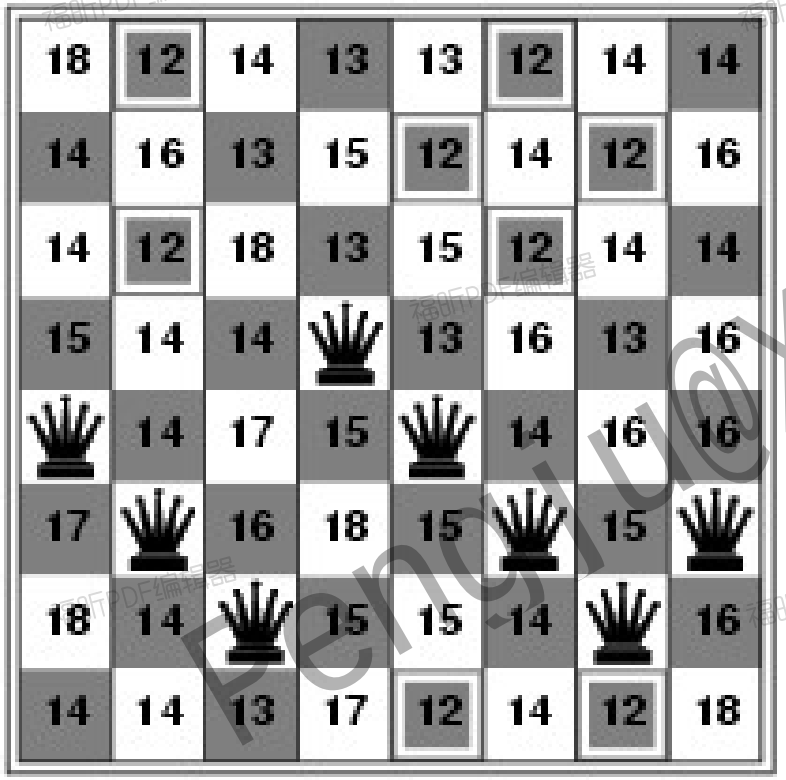
- Empirically, *swap-2* and *swap-3* makes a good TSP searcher.

Example：n-queens

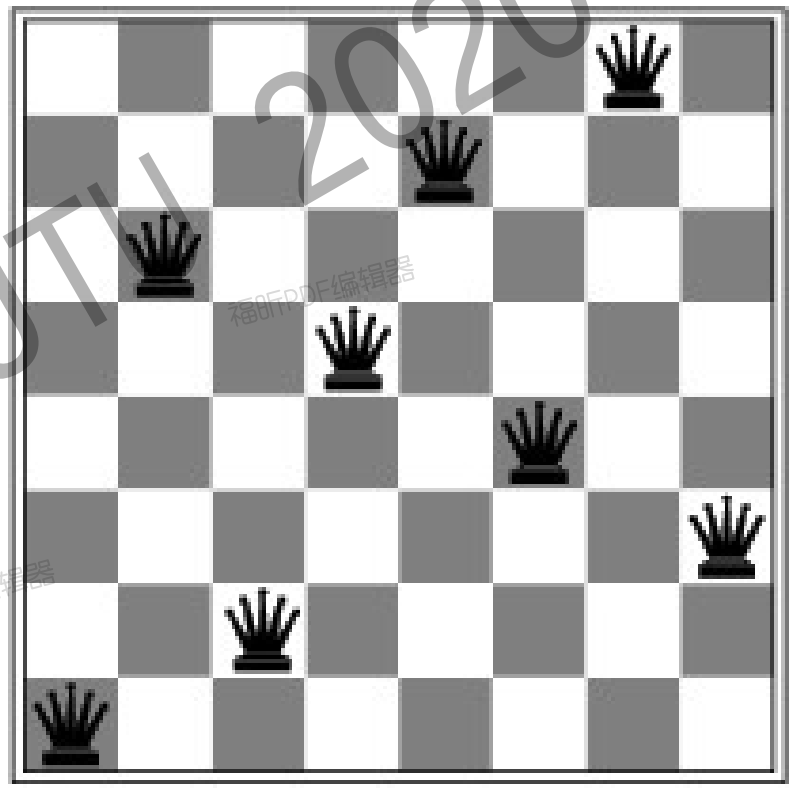
- Put n queens on an n × n board with no two queens on the same row, column, or diagonal.
- Move a queen to reduce number of conflicts.
- Almost always solve *n*-queen puzzle immediately even for $n \simeq 10^6$.



Example：n-queens



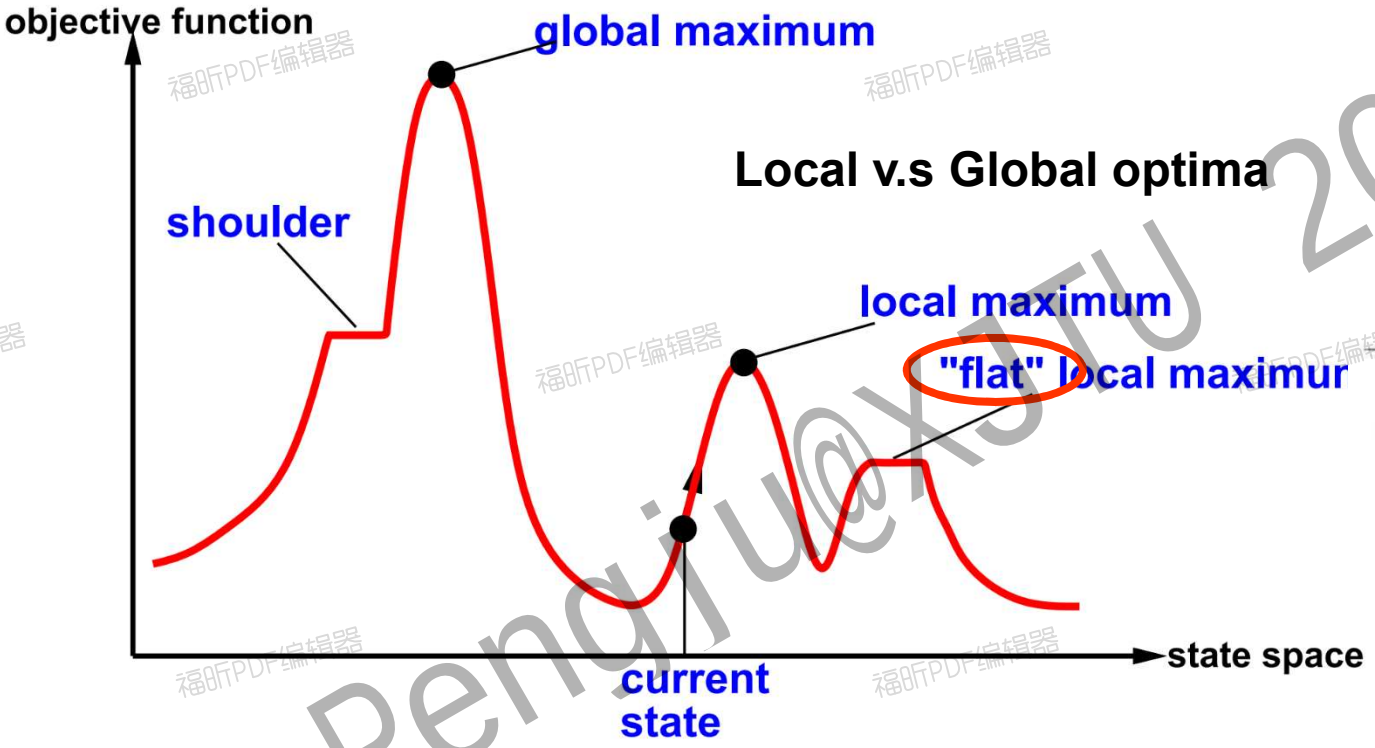
h = 17



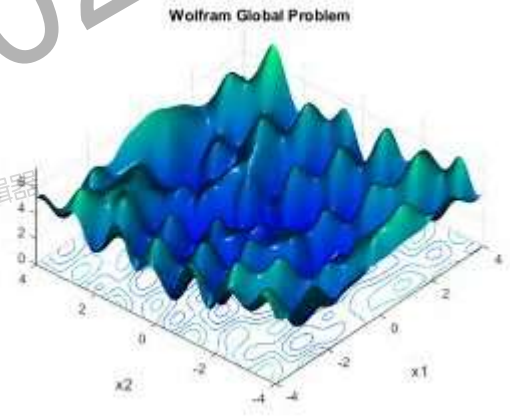
h =1

- h is the number of conflicts

Hill-climbing contd.



Local v.s Global optima

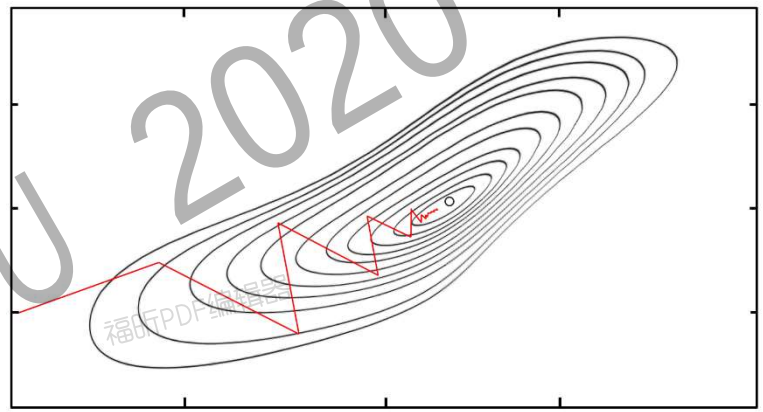


- Useful to consider state space landscape.
- Random-restart hill climbing overcomes local maxima—trivially complete
- Random sideways moves

 escape from shoulders

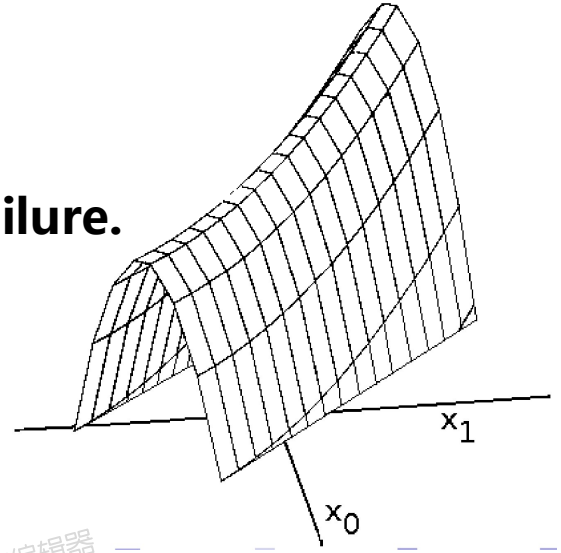
 loop on flat maxima

Performance of Steepest Descent



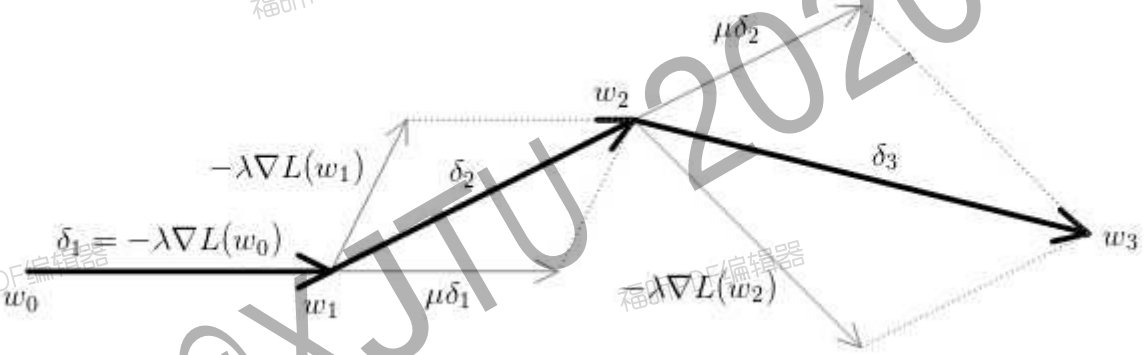
- Find only the nearest local optimum.
- May suffer from slow convergence due to the zig-zagging behavior.
- Ridges and plateau are difficult too.

- Random restart helps
  - Prob. *p* to succeed.
  - 1/*p* restarts needed.
  - Cost = cost-of-success + (1-*p*)/*p* *x* cost-of-failure.



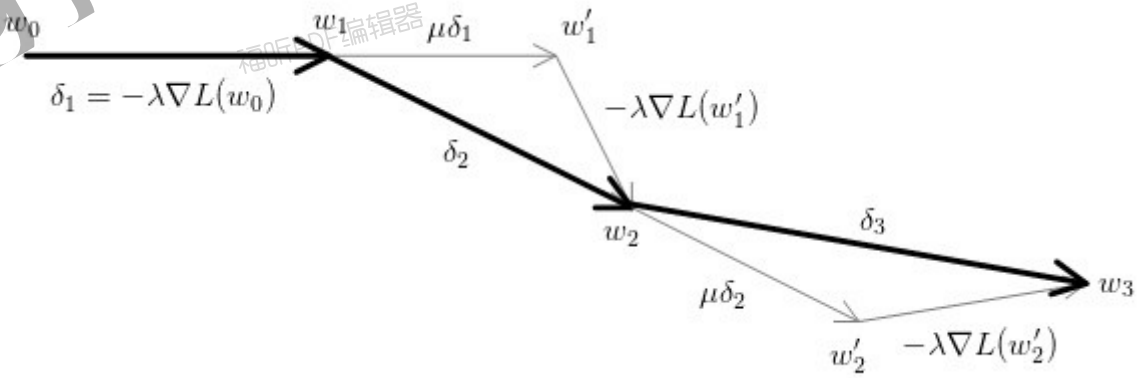
Gradient and momentum

$$\delta_{t+1} = \mu\delta_t - \lambda\nabla L(w_t)$$
$$w_{t+1} = w_t + \delta_{t+1}$$



momentum

$$\delta_{t+1} = \mu\delta_t - \lambda\nabla L(w_t + \mu\delta_t)$$
$$w_{t+1} = w_t + \delta_{t+1}$$



Nesterov momentum

Simulated Annealing (SA)

- Steepest descent gets stuck at local optimum.
- Some random walk behavior is desired.
- Simulated annealing introduces a temperature parameter T, which cools down as time goes by.
- If the new state has a lower energy (better, $\Delta E > 0$), SA accepts the new state.
- Otherwise, SA accepts the new state with a probability $e^{\Delta E/T}$.

- It has been proven that with T decreases slowly enough, SA always finds the global optimum (not practically useful, why?).

Simulated Annealing (SA)

SIMULATED-ANNEALING(*problem*, *schedule*)

```
1  current = MAKE-NODE(problem.initial_state)
2  for t = 1 to ∞
3      T = schedule(t)
4      if T == 0
5          return current
6      next = a randomly selected successor of current
7      △E = next.value − current.value
8      if △E > 0
9          current = next
10     else
11         current = next only with probability e^(△E/T)
```


Idea: escape local maxima by allowing some "bad" moves but gradually decrease their size and frequency.

Local Beam Search

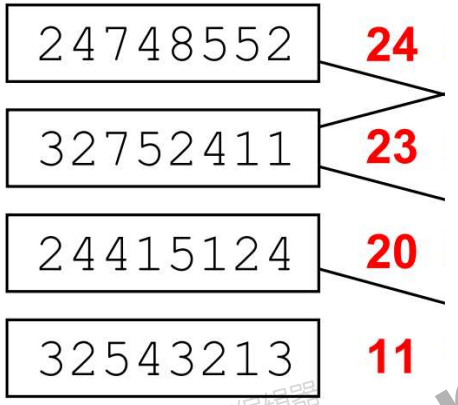
- Keep best *k* states instead of just one. Choose top *k* of all their successors
- What's different from simply running steepest descent *k* times with different initializations?

(Not the same as *k* searches run in parallel, Searches that find good states recruit other searches to join them)

- What if all *k* states become the same after awhile?
- Stochastic beam search randomly chooses *k* successors with a probability proportional to their goodness.

Genetic algorithms (GA)

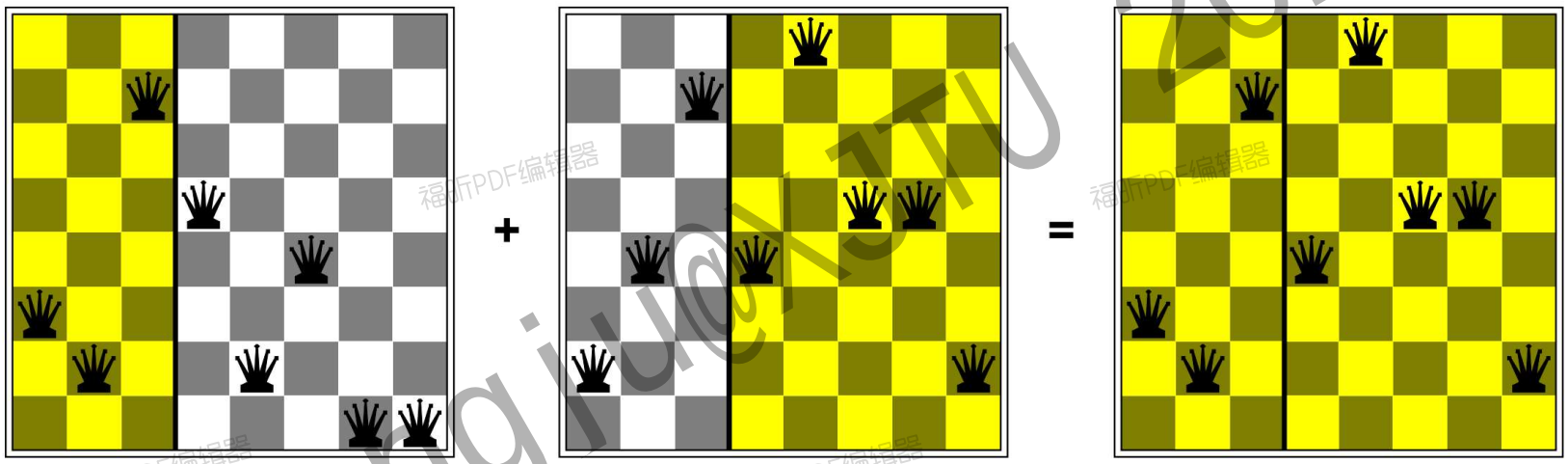
- = stochastic local beam search + generate successors from pairs of states



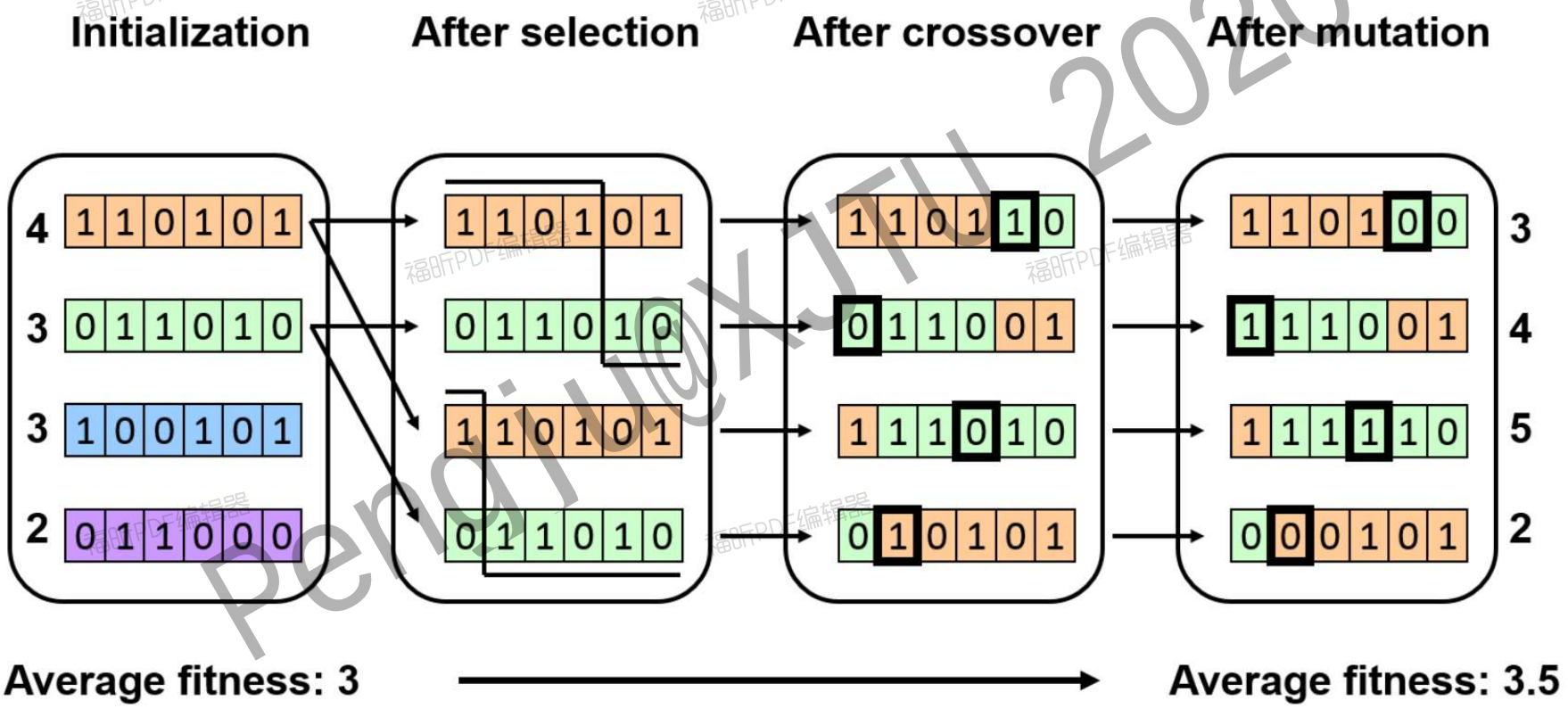
Fitness Selection Pairs Cross-Over Mutation

24/(24+23+20+11) = 31%; 23/(24+23+20+11)=29%. etc

Genetic algorithms (GA)

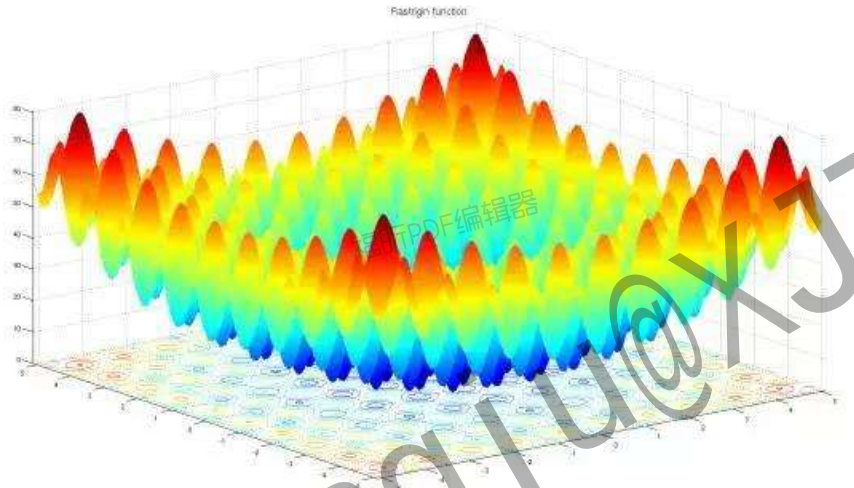


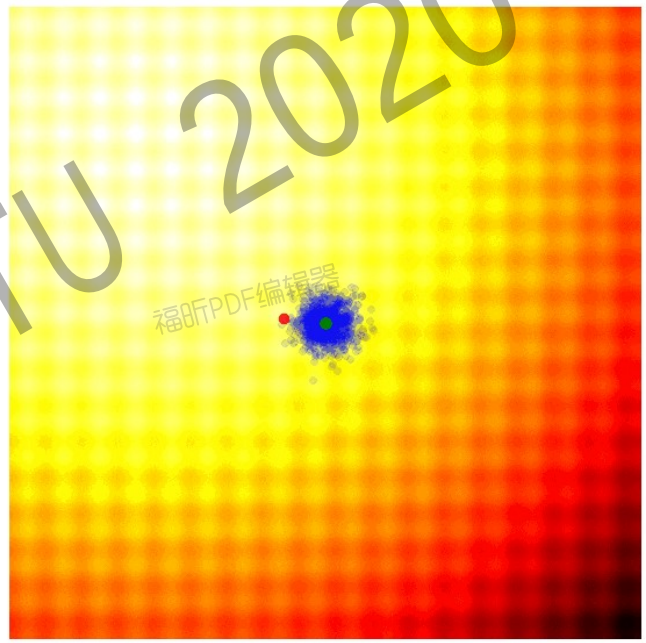
Genetic algorithms (GA)



Fitness for ONEMAX : $f(x) = \sum_i x_i$

Evolutionary Computation & Machine Learning



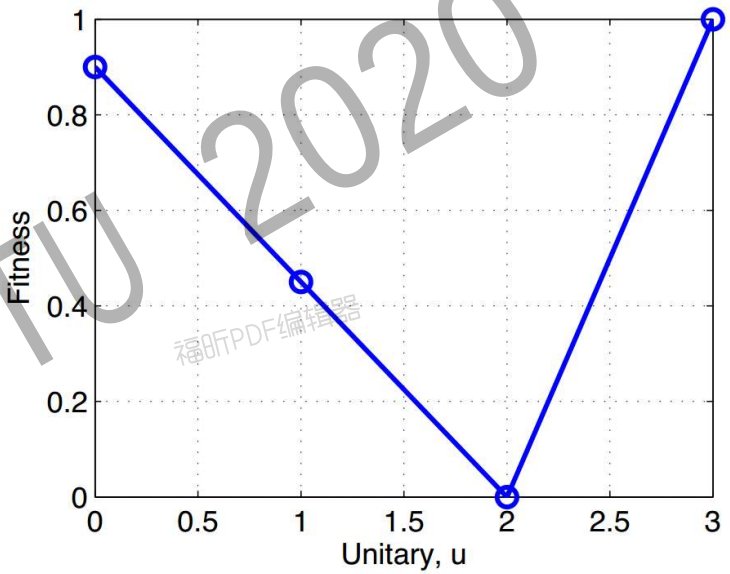


- **Large-Scale Evolution of Image Classifiers** https://arxiv.org/abs/1703.01041
- **Evolution Strategies as a Scalable Alternative to Reinforcement Learning** https://arxiv.org/abs/1703.03864

Is Random Recombination Good Enough ?

- An adversary function

$$trap_3(z) = \begin{cases} 1, & z = 111 \\ 0, & z = 110, 101, 011 \\ 0.45, & z = 100, 010, 001 \\ 0.9, & z = 000 \end{cases}$$

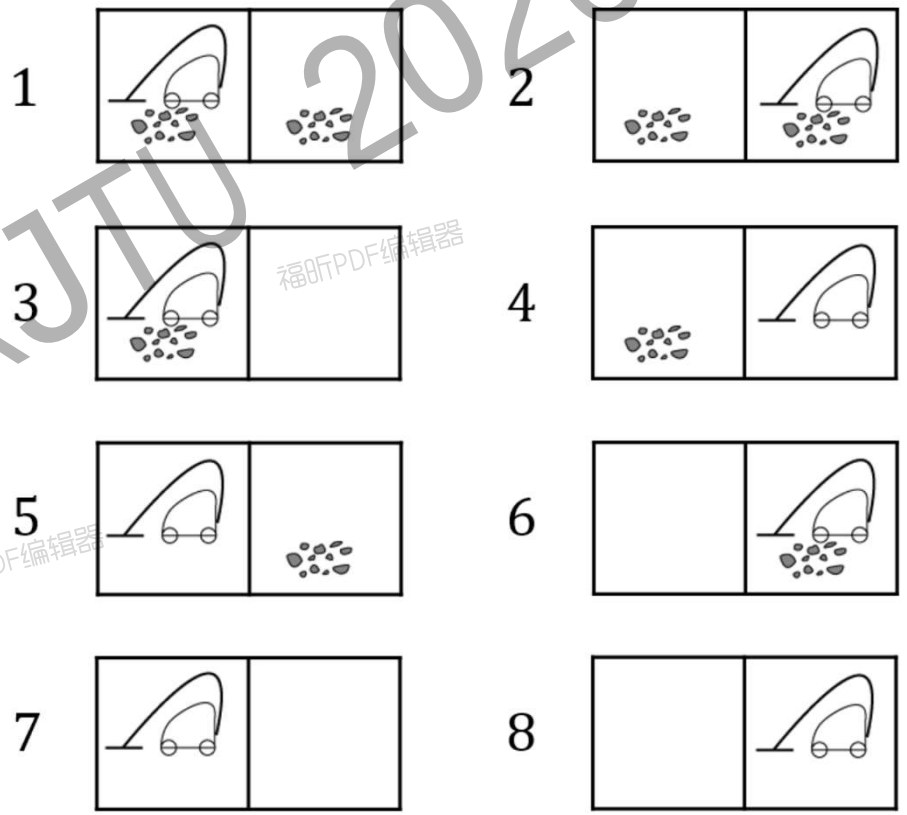


- What if *f*(*x*) = *trap*3(*x*1*x*10*x*22) + *trap*3(*x*2*x*7*x*29) + • • •
- Crossing 111 with 000 always disrupts 111 (sub-solution).
- Modern recombination involves problem decomposition via machine learning.

Non-deterministic Actions

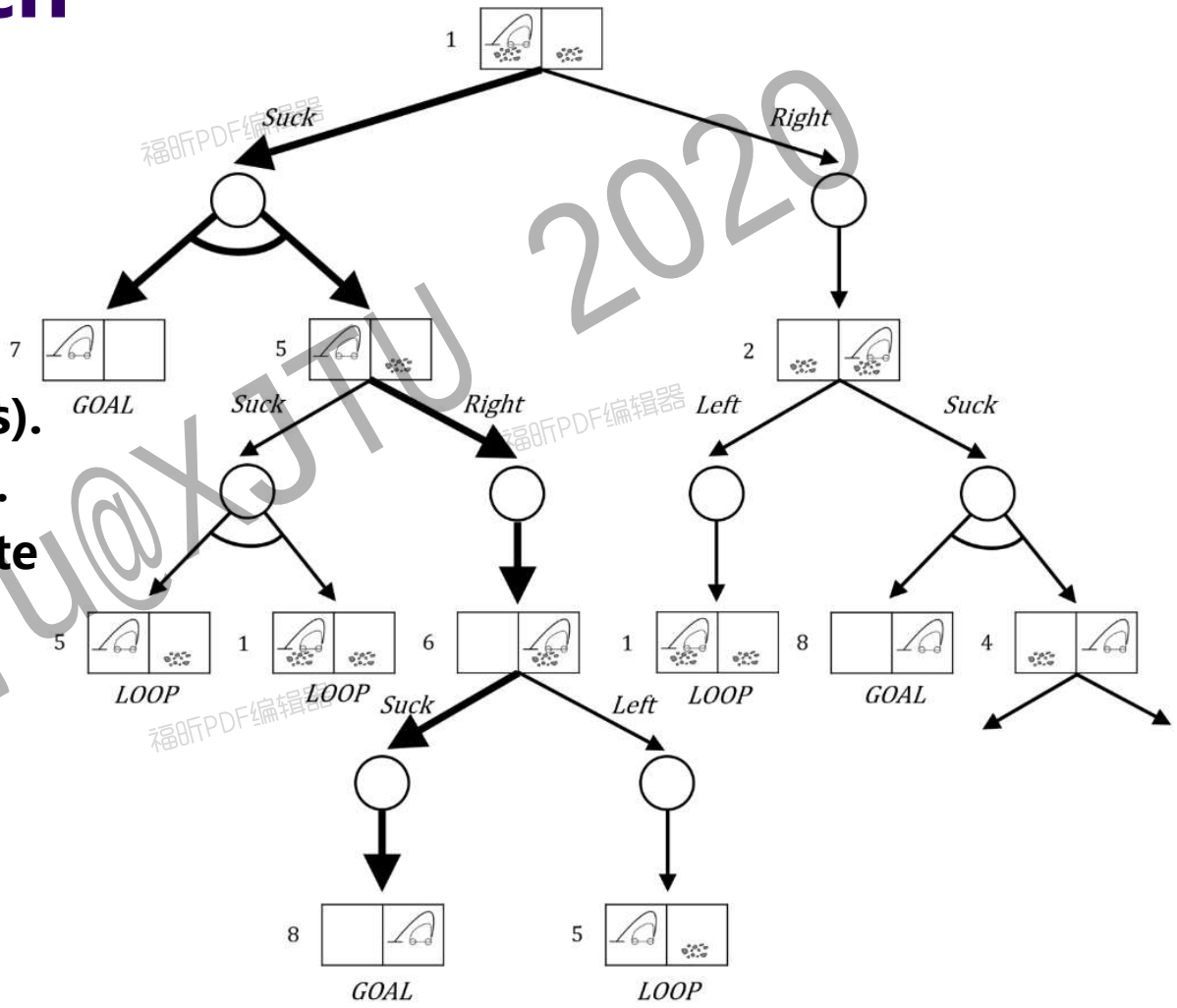
SUCK in the erratic vacuum world:

- **When applied to a dirty square, the action clean the square and sometimes clean the dirt in an adjacent square as well.**
- **When applied to a clean square, the action sometimes deposits dirt on the square.**



AND-OR Search

- ***AND* nodes**: actions (circles).
- ***OR* nodes:** states (squares).
- **Need to reach the goal state at EVERY leaf.**



AND-OR Search

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

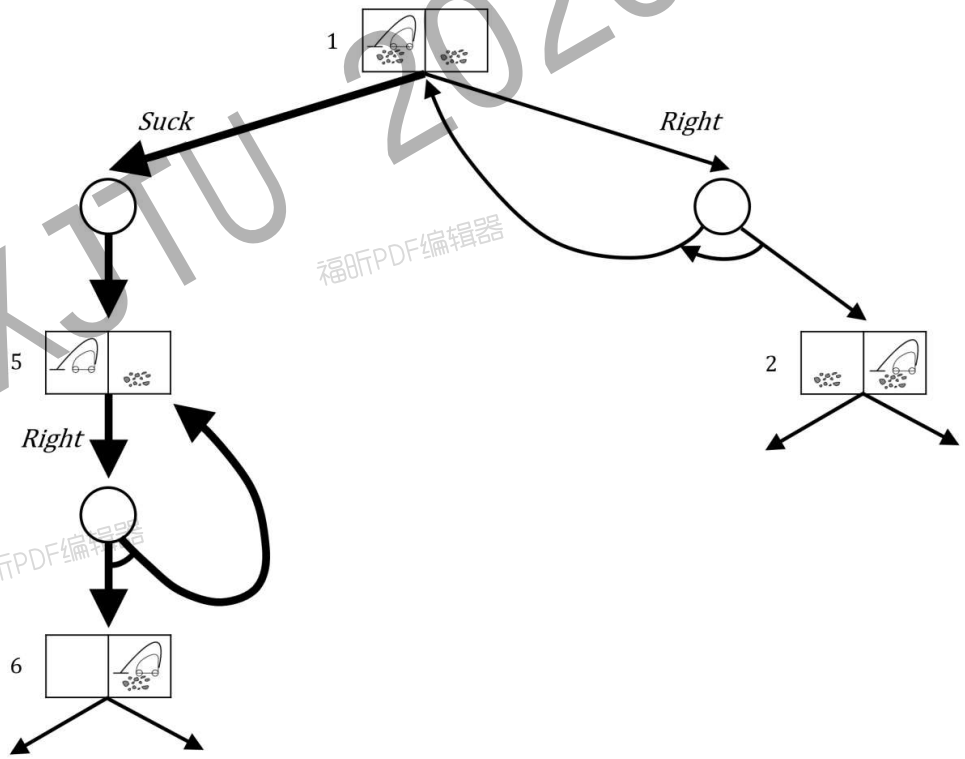
```
1  if problem.GOAL-TEST(state)
2        return the empty plan
3  if state is on path return failure
4  for each action in problem.ACTIONS(state)
5        plan = AND-SEARCH(RESULTS(state, action), problem, [state|path])
6        if plan ≠ failure
7              return [action|plan]
8  return failure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

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

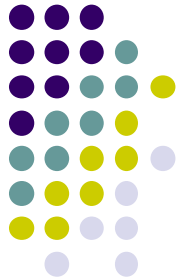
```
1  for each S_i in states
2        plan_i = OR-SEARCH(S_i, problem, path)
3        if plan_i == failure
4              return failure
5  return [if s_1 then plan_1 elseif... elseif s_{n-1} then plan_{n-1} else plan_n]
```

Keep Trying Or Not

- The slippery vacuum world: identical to the ordinary vacuum world except movement actions sometimes fails.
- Results in a cyclic search graph and a cyclic solution.
- Cause of failure: stochastic => keep trying.
- Cause of failure: unobservable property => Stop trying after a certain number of trials.

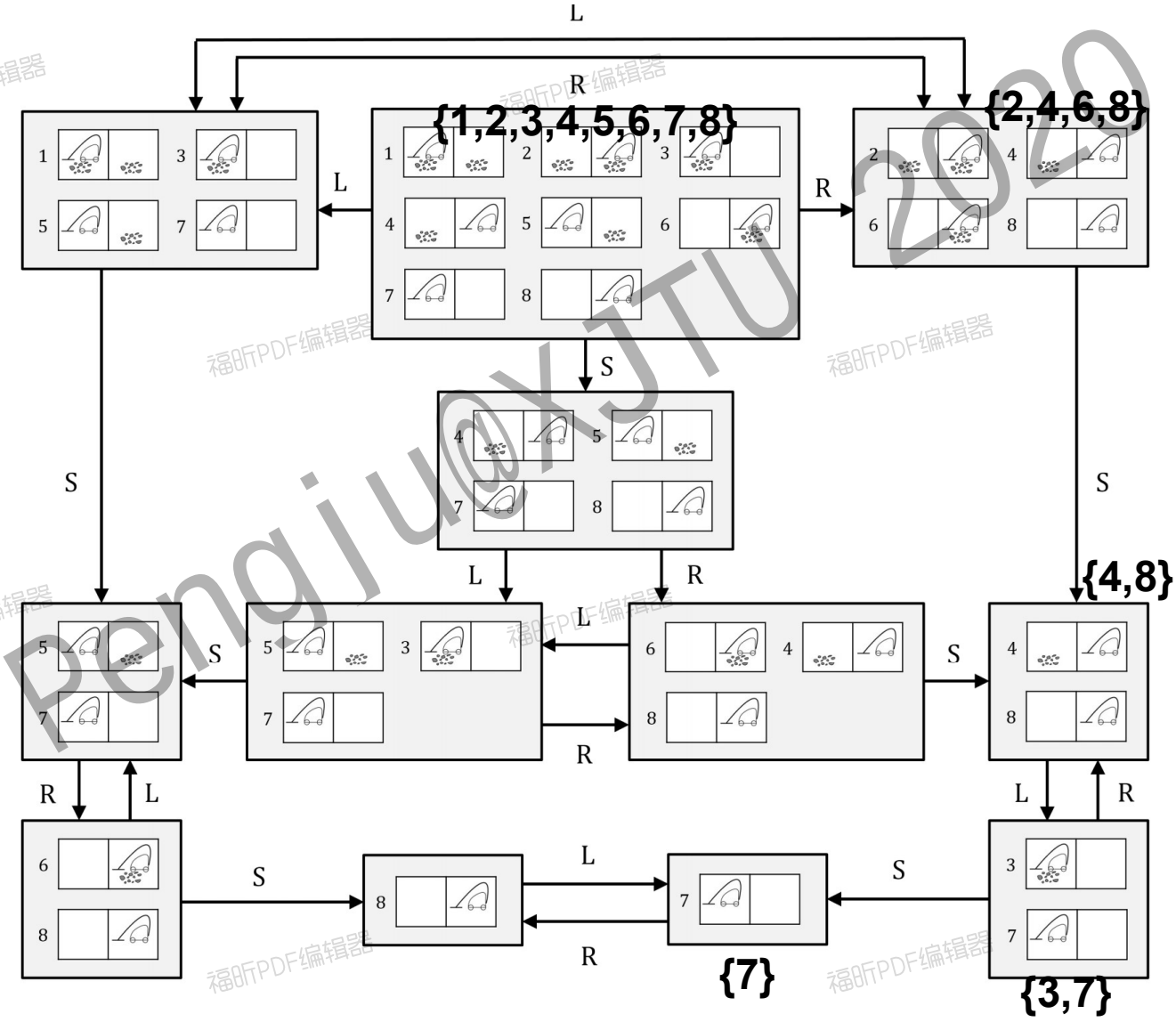


Partial Observations

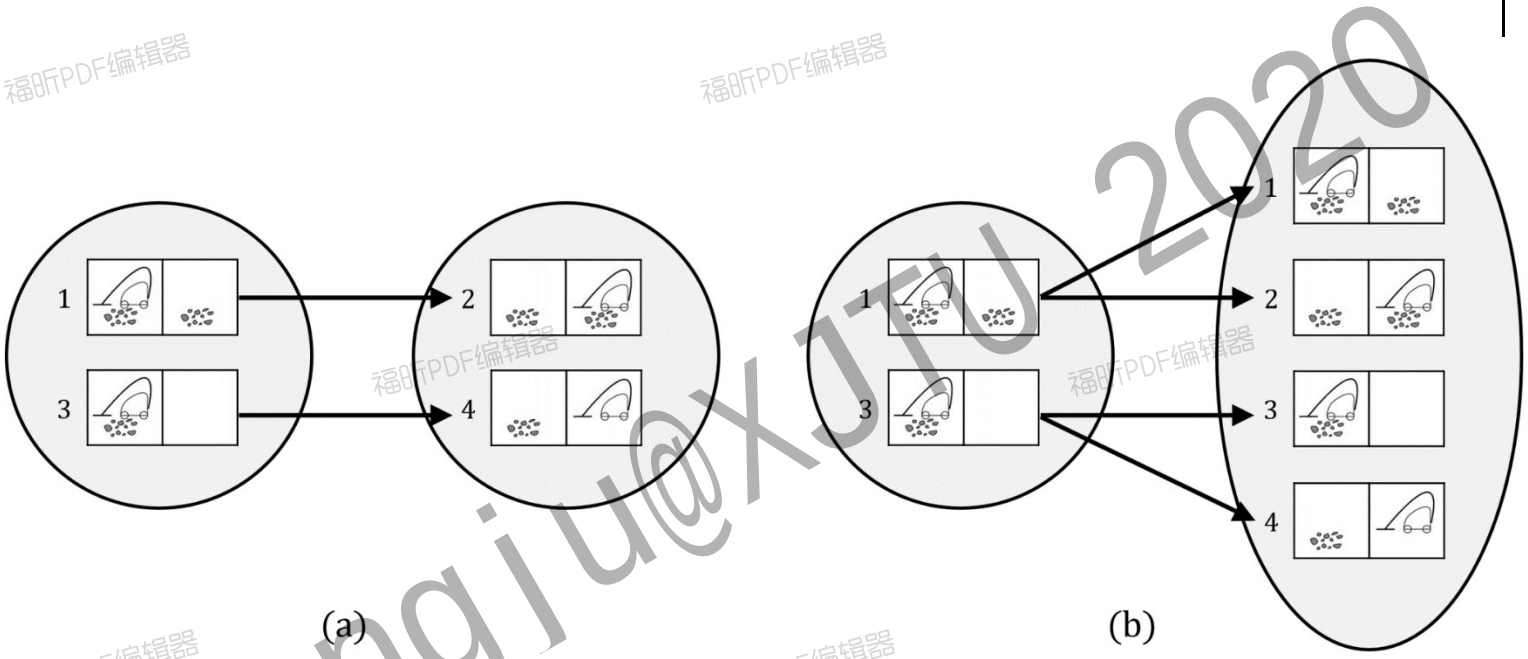


- Key idea: belief states.
- Represents the agent’s current belief about the possible states it might be in.
- Searching with no observation (sensor-less) in the vacuum world. Actually pretty easy.
  - Initial belief states are *{1, 2, 3, 4, 5, 6, 7, 8}*.
  - After [right], belief states are *{2, 4, 6, 8}*.
  - After [right,suck], belief states are *{4, 8}*.
  - After [right,suck,left], belief states are *{3, 7}*.
  - After [right,suck,left,suck], belief states are *{7}*, which is goal.

Belief States Transitions in Sensor-less Vacuum World



Transition of Belief States



- Transition with a deterministic action *RIGHT*.

- Transition with a non-deterministic action *RIGHT*.
- The number belief states usually increases after a non-deterministic action.

Searching with no observation

Belief States: The entire belief-state space contains every possible set of physical states.
Initial State: Typically the set of all states in P, although in some cases the agent will have more knowledge than this.
Actions: identify the illegal actions.
Transition model: Determinism v.s Non-determinism
Goal test: A belief state satisfies the goal only if all the physical states in it satisfy GOAL-TEST.
Path test: Whether the cost of taking an action in a given belief state is the same or have different costs in different states.

The preceding definitions enable the automatic construction of the **belief-state problem formulation** from the definition of the underlying physical problems.

Searching with Observations

- Transition of belief states consists of three stages:

Prediction Stage (same as sensor-less)

$b\hat{} = \textit{PREDICT}(b, a)$

Observation Prediction Stage determines the set of observations in the predicted belief states.

$\textit{POSSIBLE-PERCEPTS}(b\hat{}) = \{o | o = \textit{PERCEPT}(s), s \in b\hat{}\}$

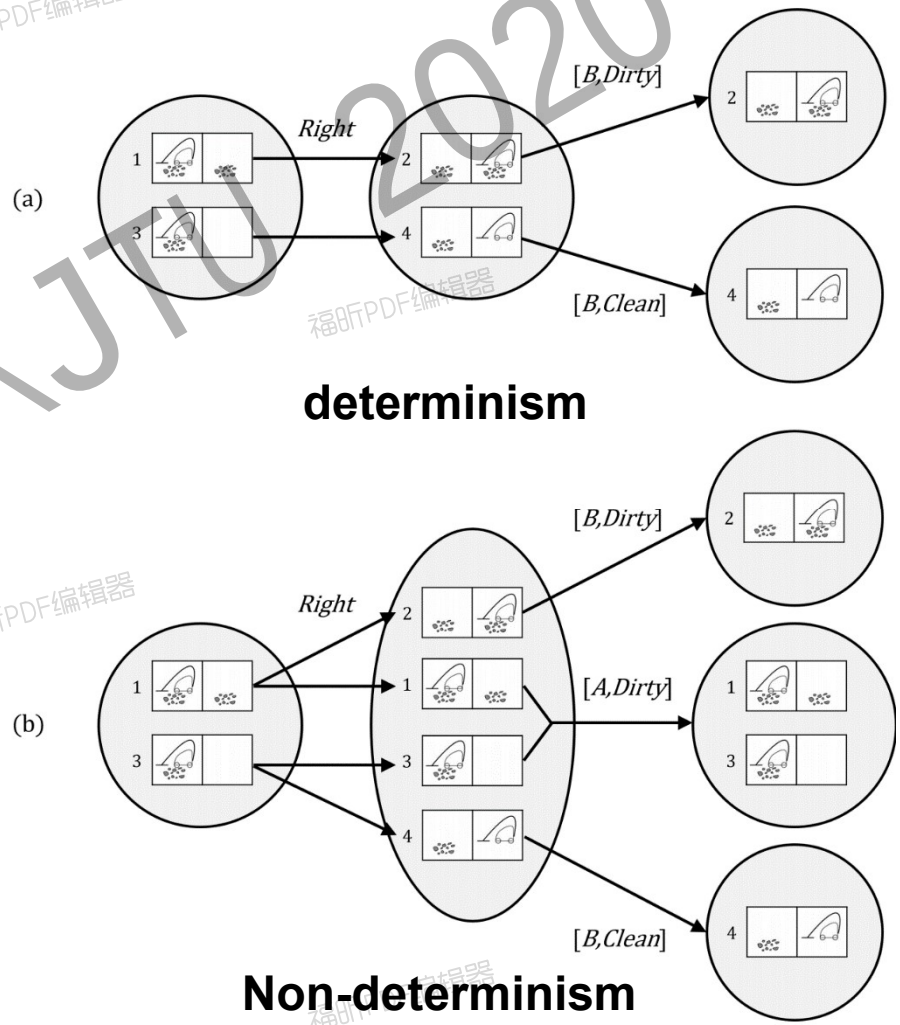
Update Stage determines bo, a subset of $b\hat{}$ which produces observation o.

$bo = \textit{UPDATE}(b\hat{}, o) = \{s | \textit{PERCEPT}(s) = o, s \in b\hat{}\}$

$$\textsc{Results}(b, a) = \{b_o | b_o = \textsc{Update}(\textsc{Predict}(b, a), o), o \in \textsc{Possible-Percepts}(\textsc{Predict}(b, a))\}$$

Vacuum World with Sensing

- **Sensor** senses location and dirt.
- ***RIGHT*** causes in two sets of belief states in the ordinary vacuum world.
- ***RIGHT*** causes in three sets of belief states in the slippery vacuum world.



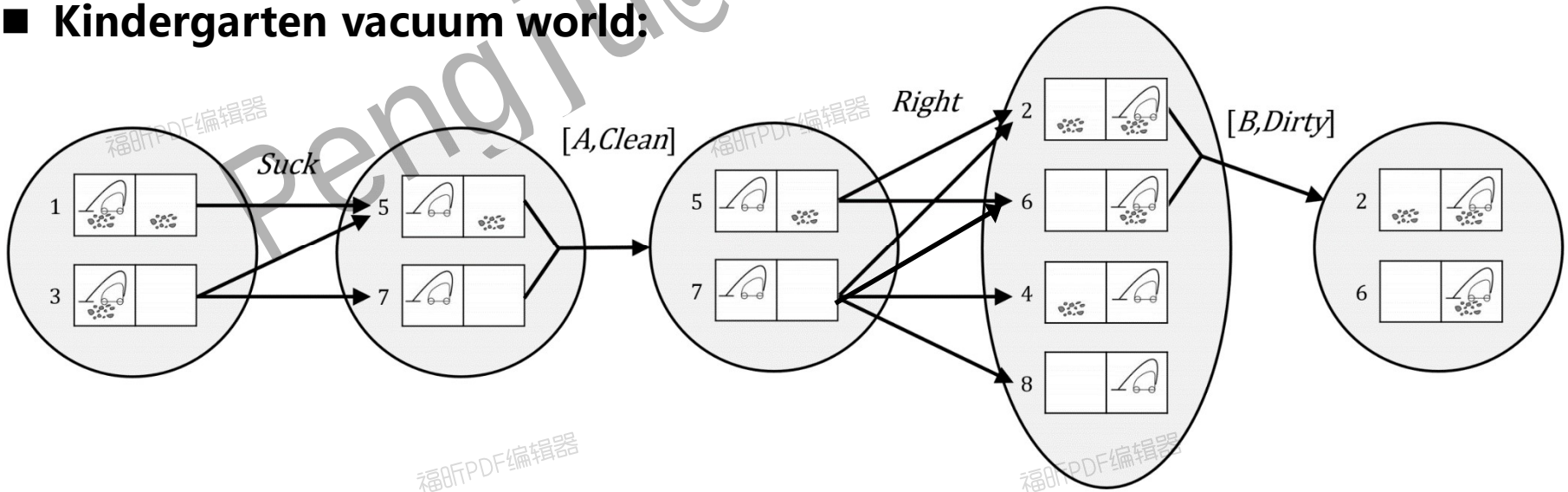
Agents for Partially Observable Environment

- **Keep estimating the belief states and *AND-OR* searching for solutions to reach goal states..**

Recursive State Estimator

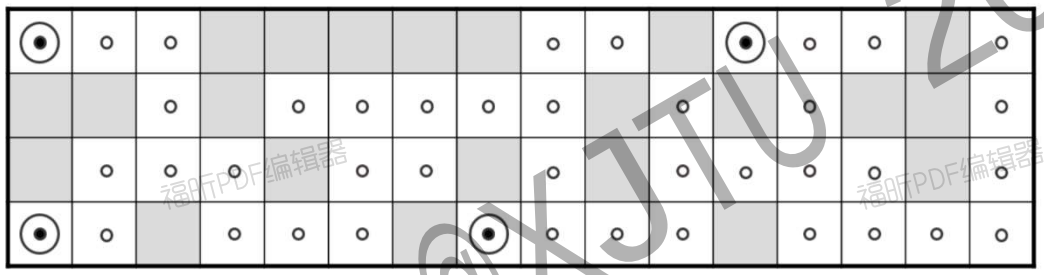
$$b' = \text{UPDATE}(\text{PREDICT}(b, a), o)$$

- **Kindergarten vacuum world:**

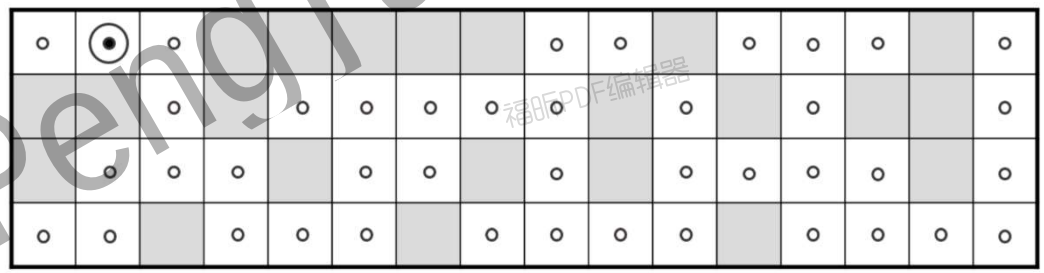


Localization in a Maze

- Map is known, and 4 sonar sensors work perfectly.
- *MOVE* moves the robot randomly to one of the adjacent squares.



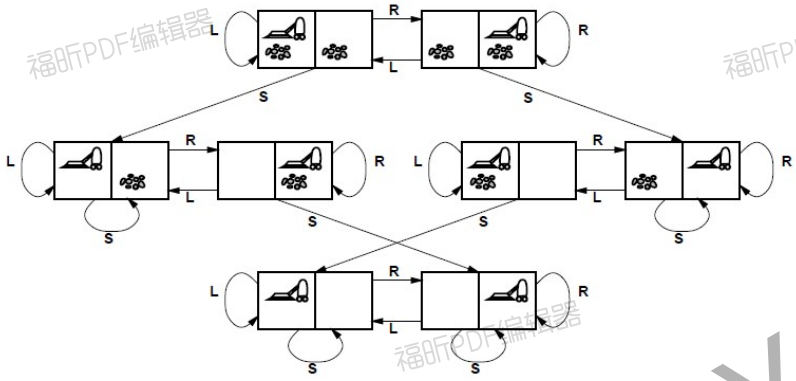
(a) Possible locations of robot after $E_1 = NSW$



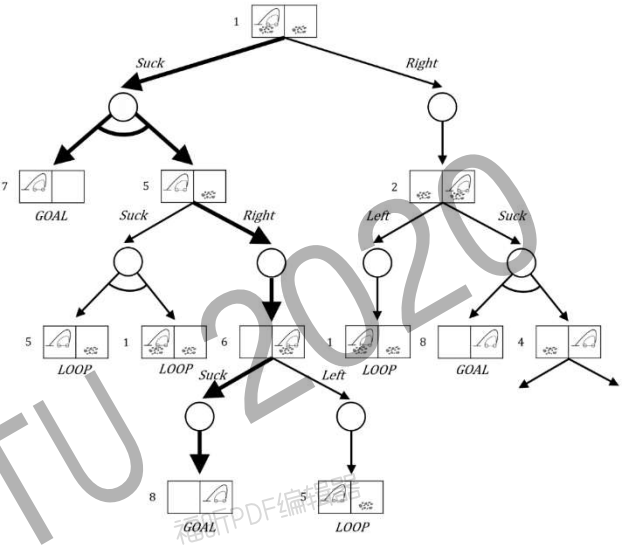
(b) Possible locations of robot after $E_1 = NSW$, $E_2 = NS$

$$\text{UPDATE}(\text{PREDICT}(\text{UPDATE}(b, NSW), \text{MOVE}), NS)$$

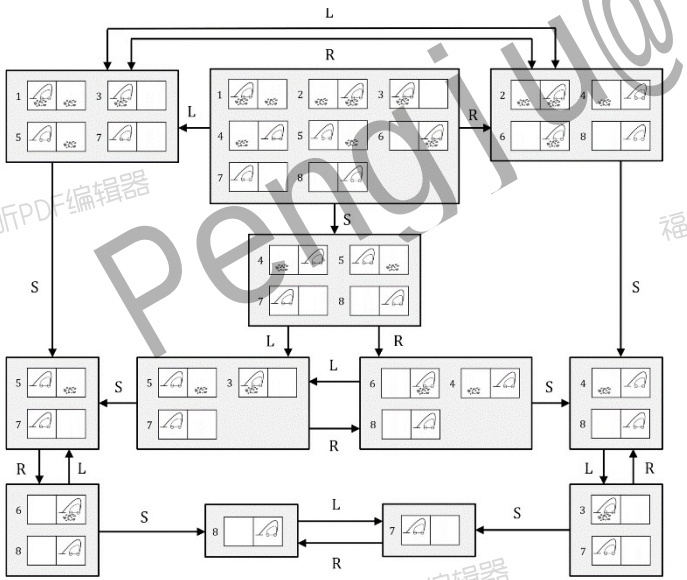
Review：vacuum world



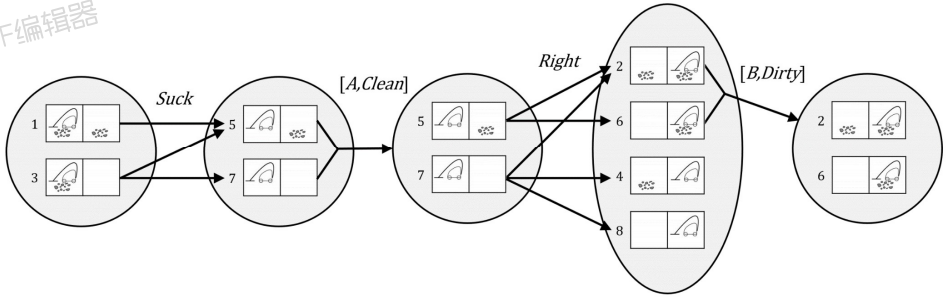
Deterministic with Sensors



Non-Deterministic with Sensors



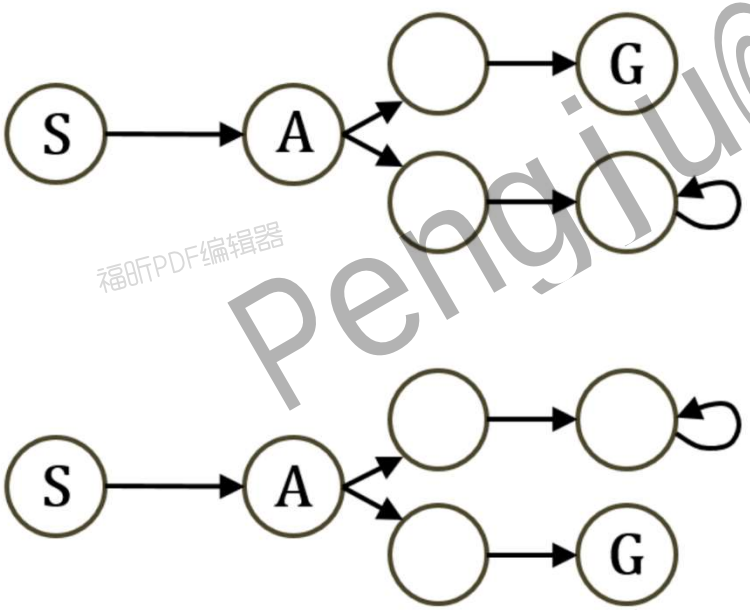
Sensor-less (belief states)

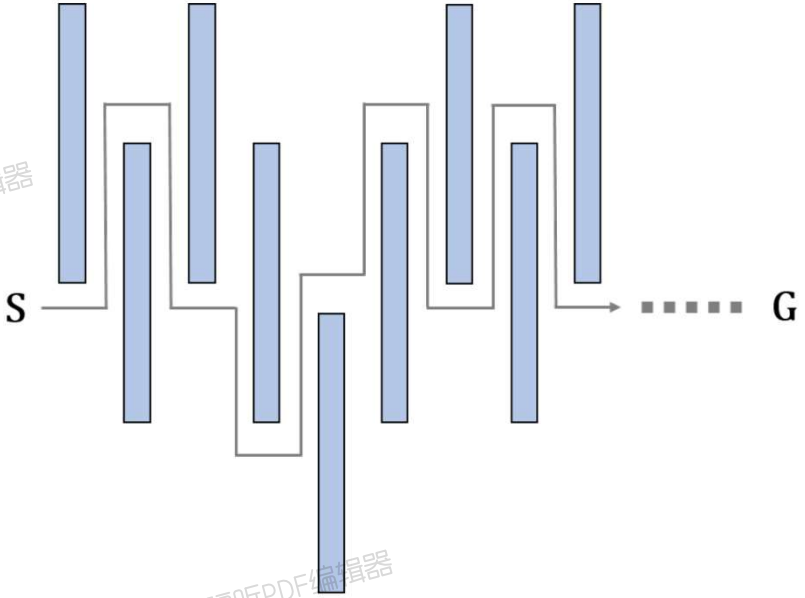


Partially Observable

Online Search with Unknown Environments

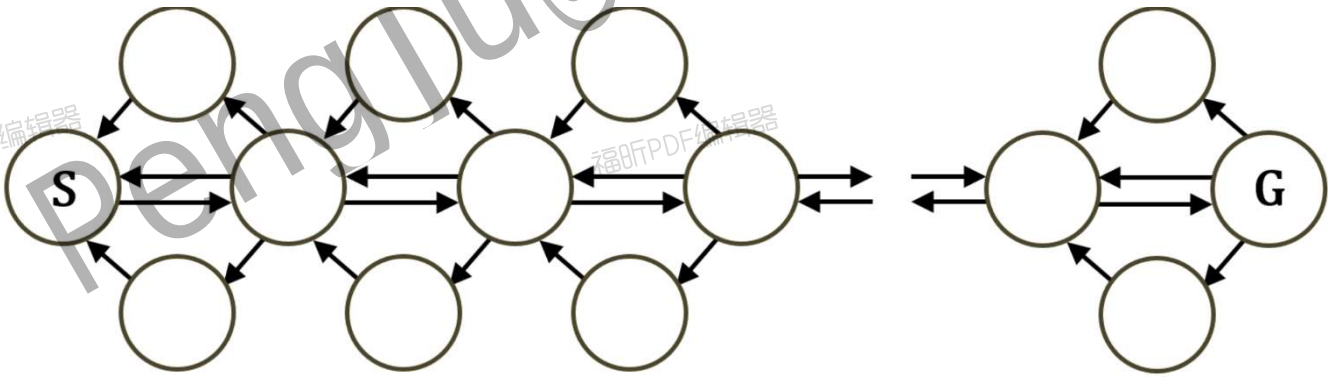
- **Competitive ratio** = $\frac{actual - cost}{minimum - cost}$. We'd like to minimize this.
- If all actions are **reversible**, **online-DFS** visits every states exactly twice in the worst case with **enough memory**.
- If some actions are **irreversible**, a small (or even finite!) competitive ration can be difficult to achieve.





Search with Limited Memory

- Only one or a few states are stored.
- 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 in the below example).
- Random walk is complete for finite state spaces.



Learning Real-Time A*(LRTA*)

- *H*[s]: a table of cost estimates indexed by state, initially empty.
- *result*[s, a]: a table indexed by state and action, initially empty.

LRTA*-AGENT(s')

```
1  if GOAL-TEST(s')
2        return stop
3  if s' is a new state (not in H)
4        H[s'] = h(s')
5  if s ≠ NULL
6        result[s, a] = s'
7        H[s] = min_{b∈ACTIONS(s)} LRTA*-COST(s, b, result[s, b], H)
8  a = argmin_{b∈ACTIONS(s')} LRTA*-COST(s', b, result[s', b], H)
9  s = s'
10 return a
```

Learning Real-Time A*(LRTA*)

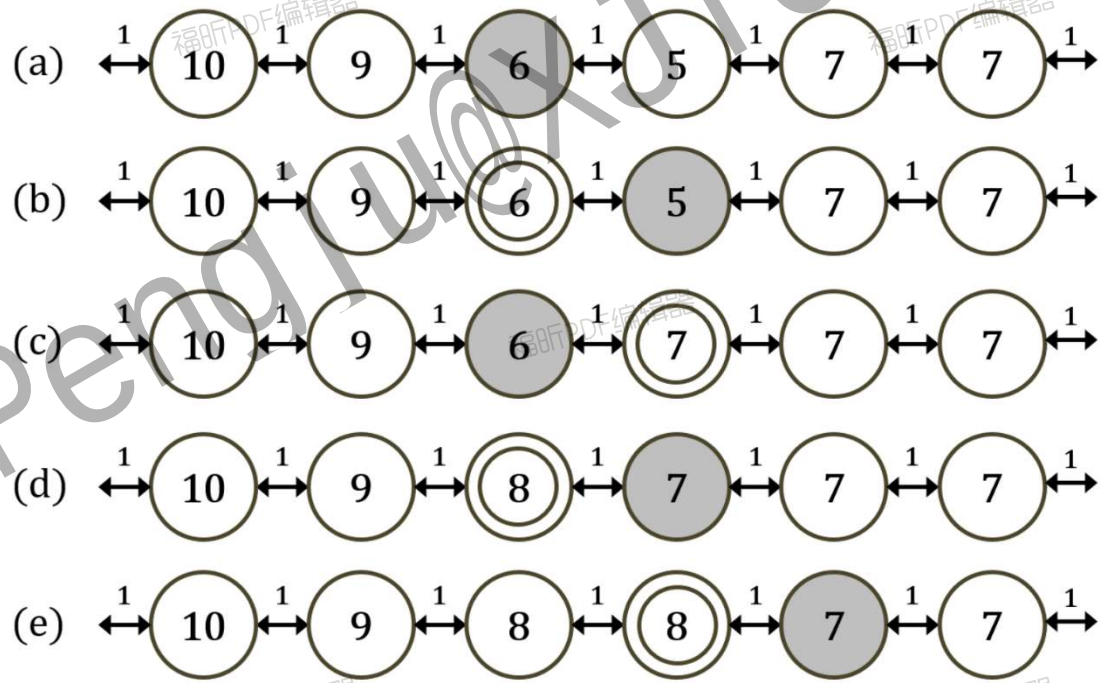
- LRTA* keeps updating $H[s]$.
- LRTA* always chooses the apparently best action.
- Optimism under uncertainty: If an action has never tried in a state, LRTA* assumes the least possible cost — $h(s)$. This encourages exploration.

LRTA*-Cost(s, a, s', H)

```
1  if s' is undefined return h(s)
2  else return c(s, a, s') + H[s']
```

Learning Real-Time A*(LRTA*)

- 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.
- Shaded: agent's location, circle: *H*[*s*] updated.



Summary

- **Steepest descent** is extremely fast for simple problems.
- 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. Difficult to analyze though.
- ***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), but are most popular nowadays.
- **LRTA*** works well if memory are enough.