



WPI

CS 4341
Introduction to Artificial Intelligence

**Lecture 6 and 7: Search in Complex
Environments**

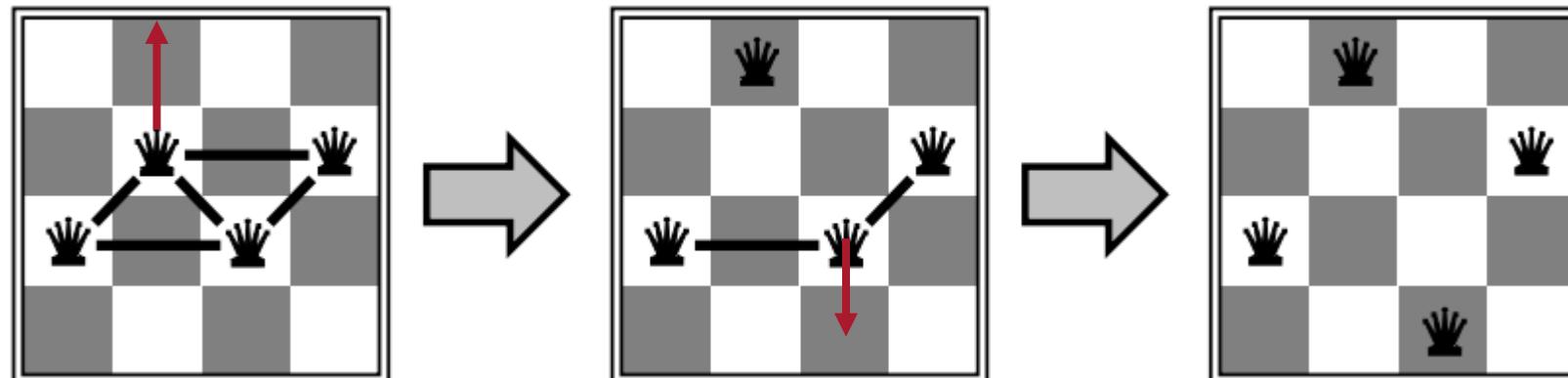
By

Ben C.K. Ngan

Worcester Polytechnic Institute

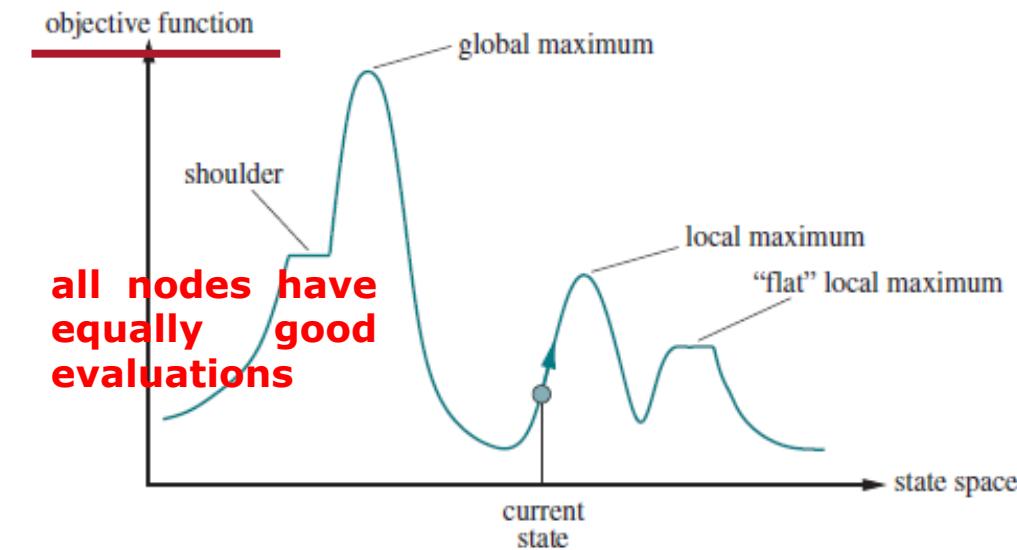
Local Search and Optimization Problems

- A **local search** algorithm operates by searching **from a start state to neighboring states, without keeping track the content of the paths**, i.e., **the path from the initial to the goal state is irrelevant**, according to an objective function..
 - Use **iterative** improvement algorithms, as it keeps a single “current” state and try to improve it.
 - Use a single current node as a state and move to a neighbor of that node. It is a **heuristic search**.
 - The updated node value is closer to the goal than the previous node value. It is **known as a local search**.
 - A local search algorithm is a type of heuristic search algorithm.
- Local search algorithms can solve **optimization problems**, in which the aim is to find **the best state** according to an **objective function** that calculates the quality of each update and all the **constraints** being satisfied.
- Example: **n-Queens**
 - Put n queens on an $n \times n$ chessboard
 - No two queens on the same row, column, or diagonal that are the constraints.
 - Start with one queen in each column
 - Move a queen to reduce number of conflicts



Local Search and Optimization Problems

- Local search solve optimization problems
 - Always find the next best state, the path from the initial to the goal state is irrelevant, according to an objective function.
 - Each point (state) in the state-space landscape has an “elevation”, i.e., the value computed by the objective function.
 - If elevation corresponds to an objective function (e.g., **profit**), the aim is to find the highest peak – **a "global"/"local" maximum → A Hill Climbing Process**
 - If elevation corresponds to a **cost**, the aim is to find the lowest valley – **a "global"/"local" minimum → A Gradient Descent Process.**
- Local Search Algorithms can be:
 - Hill-climbing Search
 - Simulated Annealing
 - Local Beam Search
 - Genetic Algorithms



State Space Diagram for Hill Climbing

Hill-Climbing Search

- Keeps track of one current state and one each iteration moves to the neighboring state with the highest value of the objective function:
 - Start **an initial state (e.g., an initial solution)**
 - A greedy local search: select **the nearest** neighbor state/solution without thinking ahead about where to go next.
 - Repeat: move to **the best** neighboring state/solution in the direction of increasing value (uphill)
 - Terminate when it reaches a "**peak**" where no neighbor has a higher value than the current one.
- The **hill_climbing(problem)** function in search.py.
- **Adv: Low computation power and Lesser time**
- **Problem:** Depending on the initial state, it can get stuck for any of the following reasons.

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current  $\leftarrow$  problem.INITIAL
  while true do
    neighbor  $\leftarrow$  a highest-valued successor state of current
    if VALUE(neighbor)  $\leq$  VALUE(current) then return current
    current  $\leftarrow$  neighbor

def hill_climbing(problem):
    """
    [Figure 4.2]
    From the initial node, keep choosing the neighbor with highest value,
    stopping when no neighbor is better.
    """

    current = Node(problem.initial)
    while True:
        neighbors = current.expand(problem)
        if not neighbors:
            break
        neighbor = argmax random tie(neighbors, key=lambda node: problem.value(node.state))
        if problem.value(neighbor.state)  $\leq$  problem.value(current.state):
            break
        current = neighbor
    return current.state
```

"Return an element with highest fn(seq[i]) score; break ties at random."

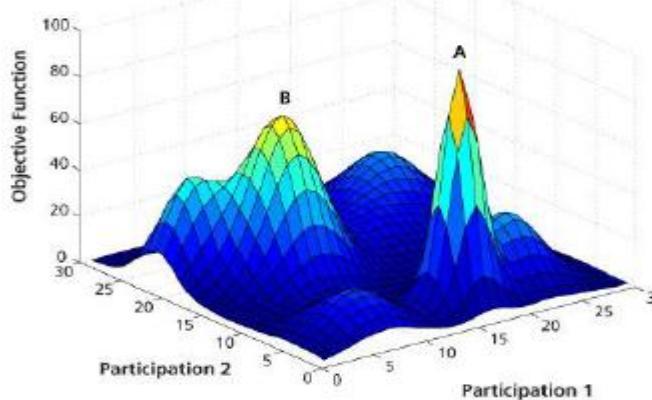
Class Exercise 1

Review the **hill_climbing(problem)** in search.py, <https://github.com/aimacode/aima-python/blob/master/search.py>, and do the following with your teammates/classmates

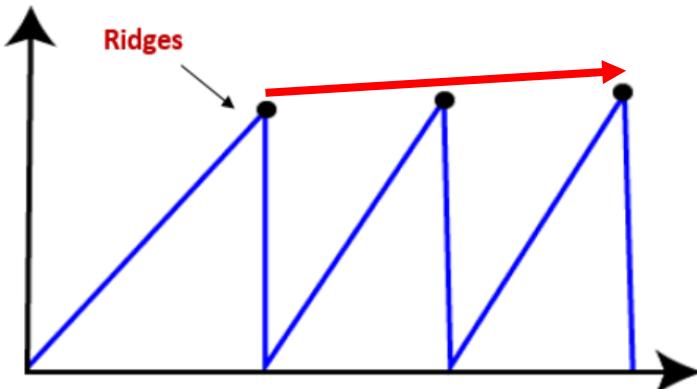
- If the solution/state value computed is a profit value, do you need to do any modification on the code? Why?
- If the solution/state value computed is a +ve cost value, do you need to do any modification on the code? Why?
- If the solution/state value computed is a -ve cost value, do you need to do any modification on the code? Why?

Problems of Hill-Climbing Search

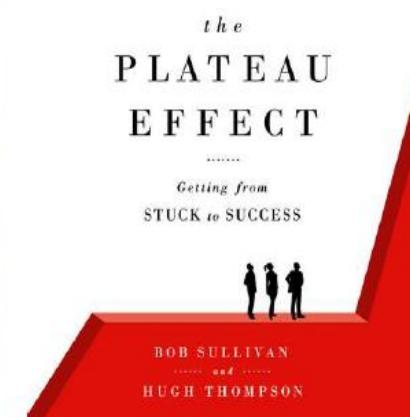
- **Local Maxima and Minima:** They are the peak that is better than each of its neighboring states but worse than the global maxima and global minima.



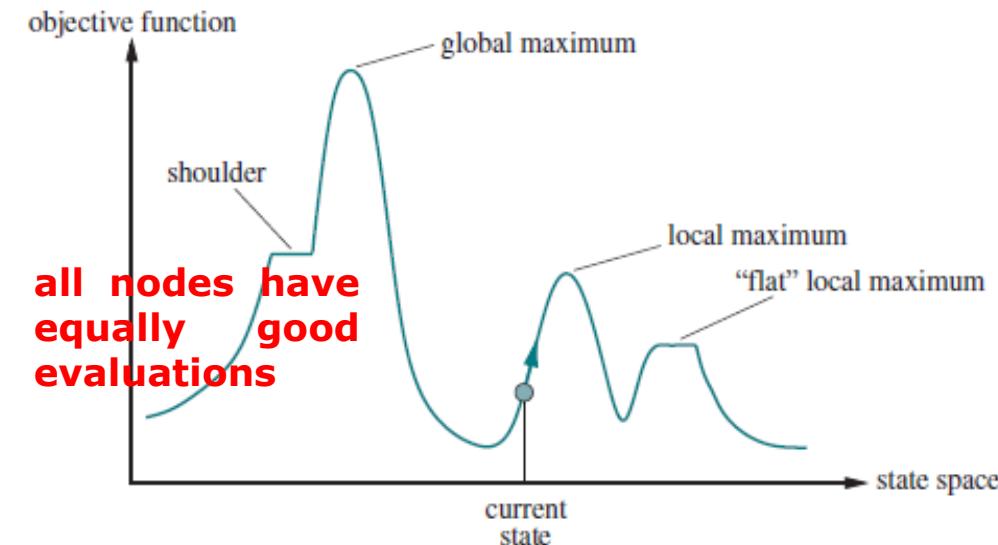
- **Ridges:** It is like a knife edge that gives a false sense of top of the hill, as no slope change appears.



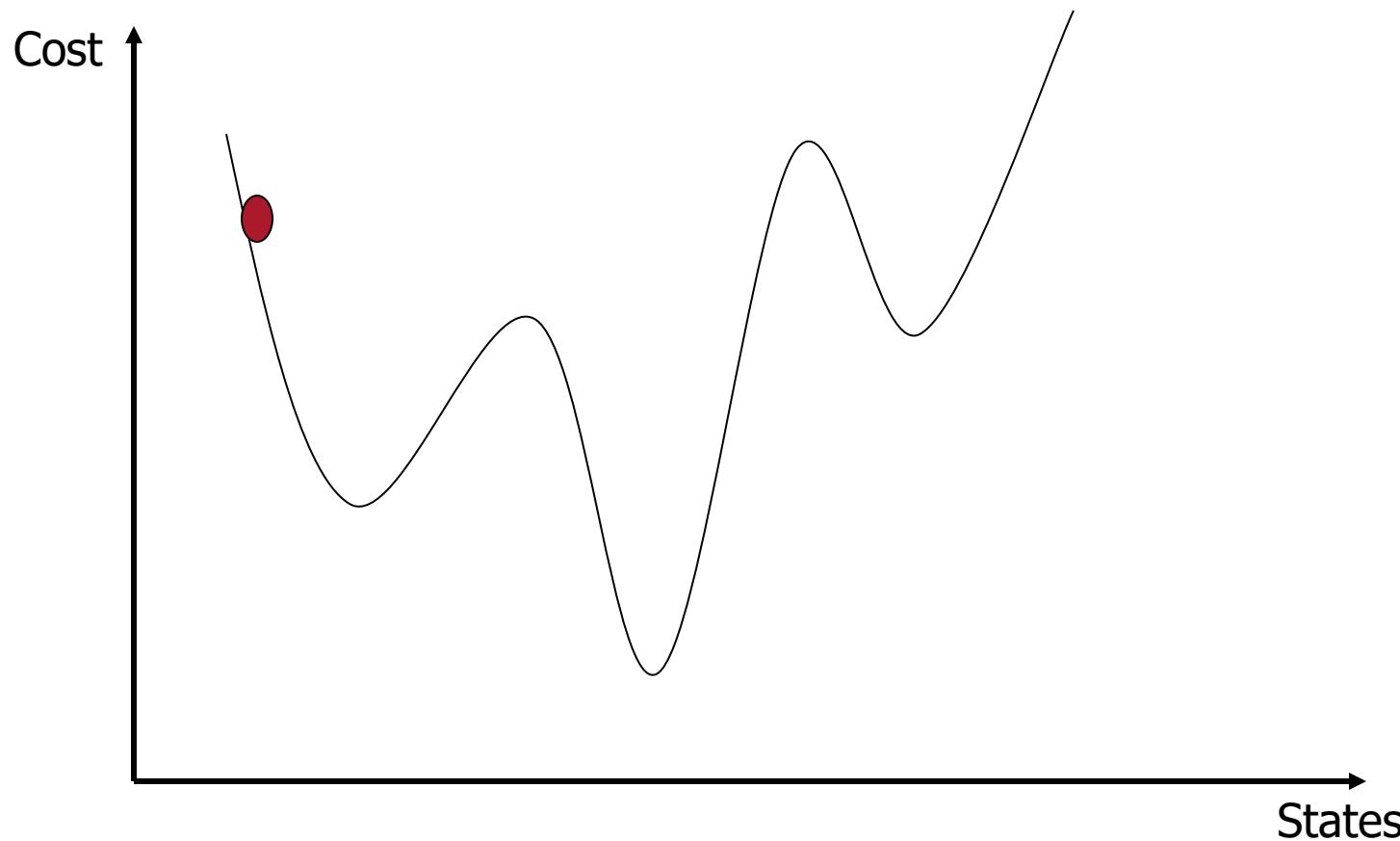
- **Plateaus:** All nodes have equally good evaluations, called a shoulder.



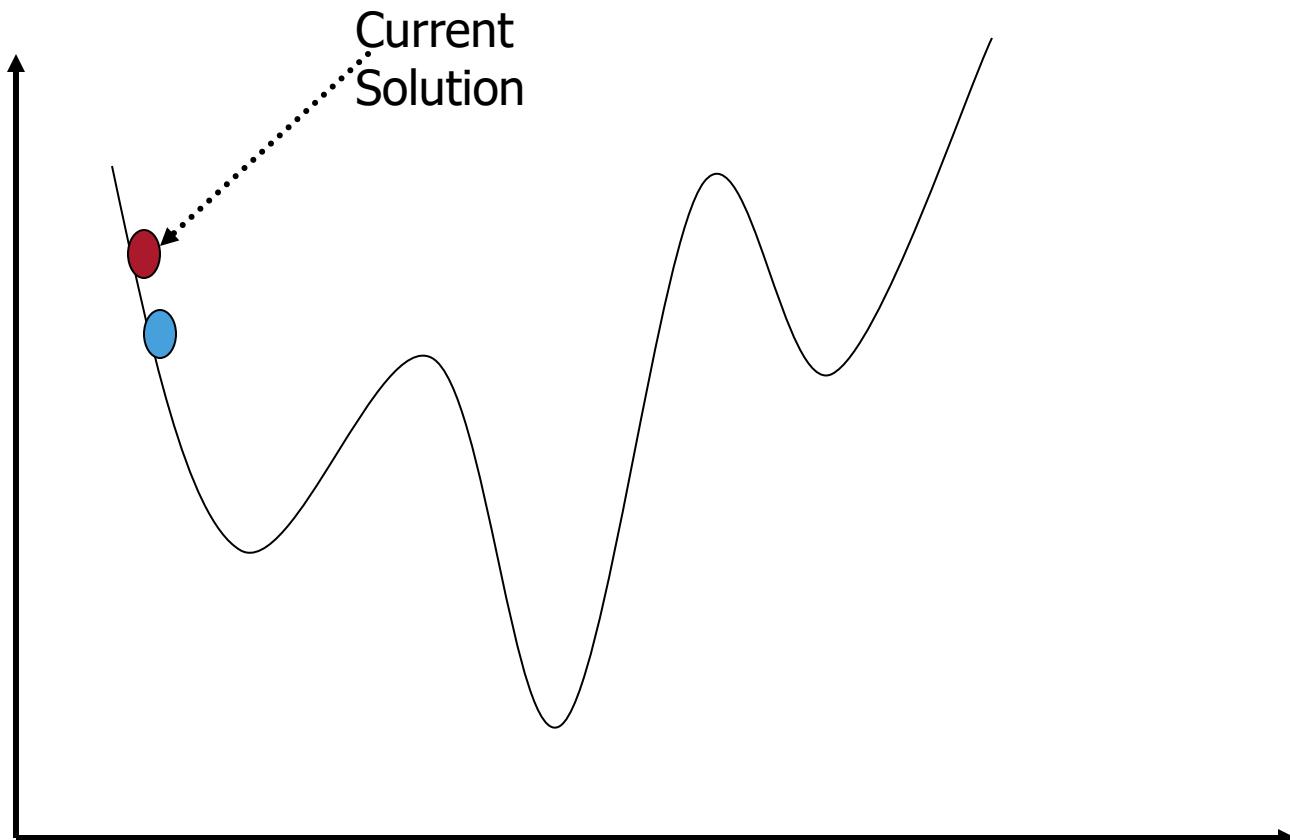
Less optimal solution and the solution is not guaranteed



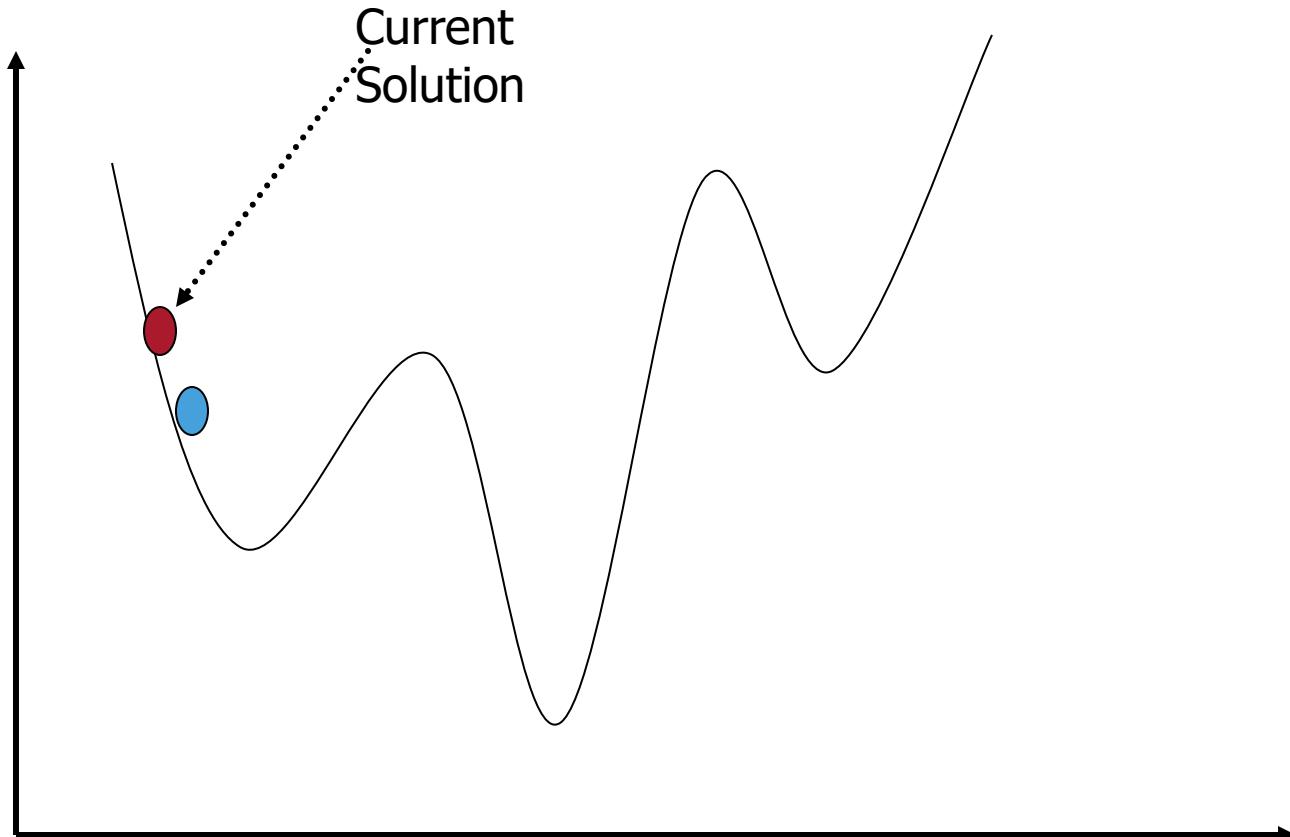
Hill-Climbing Search



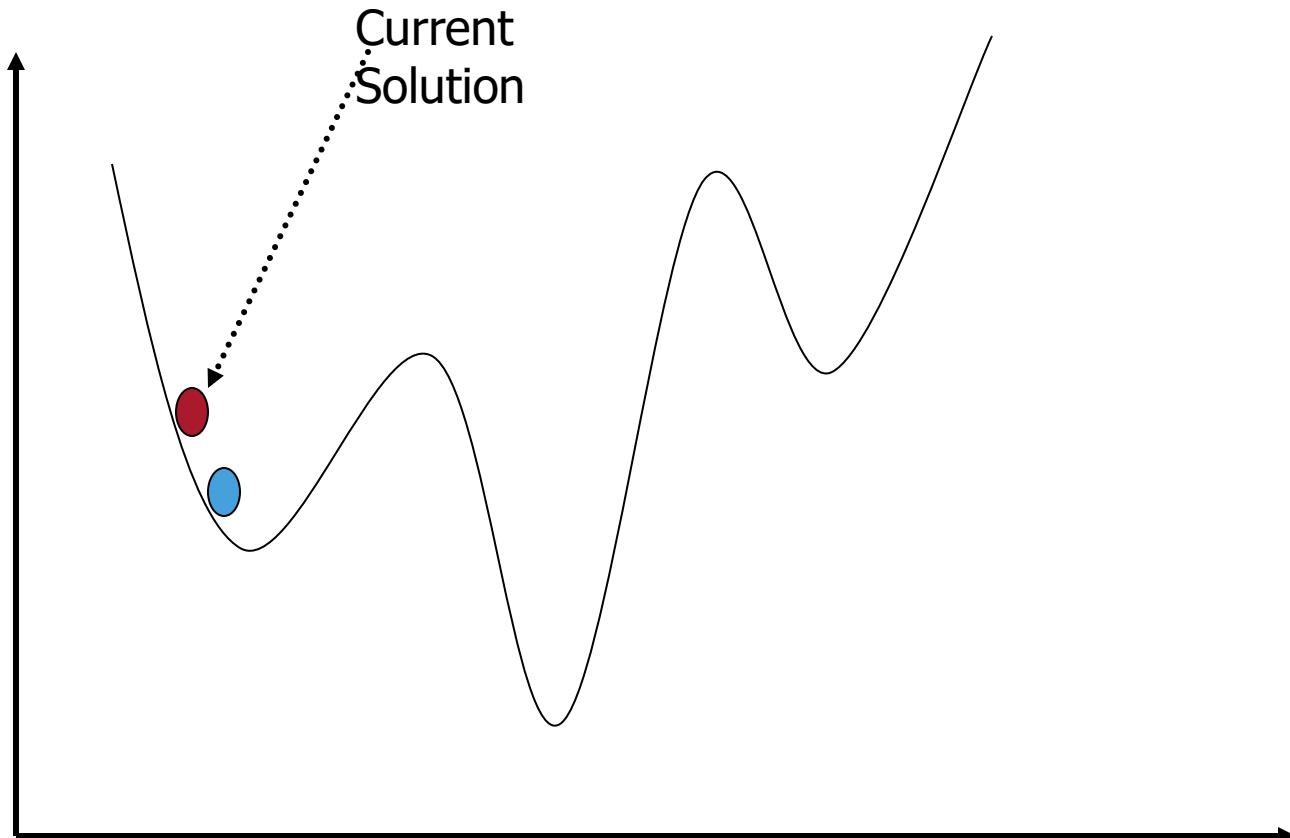
Hill-Climbing Search



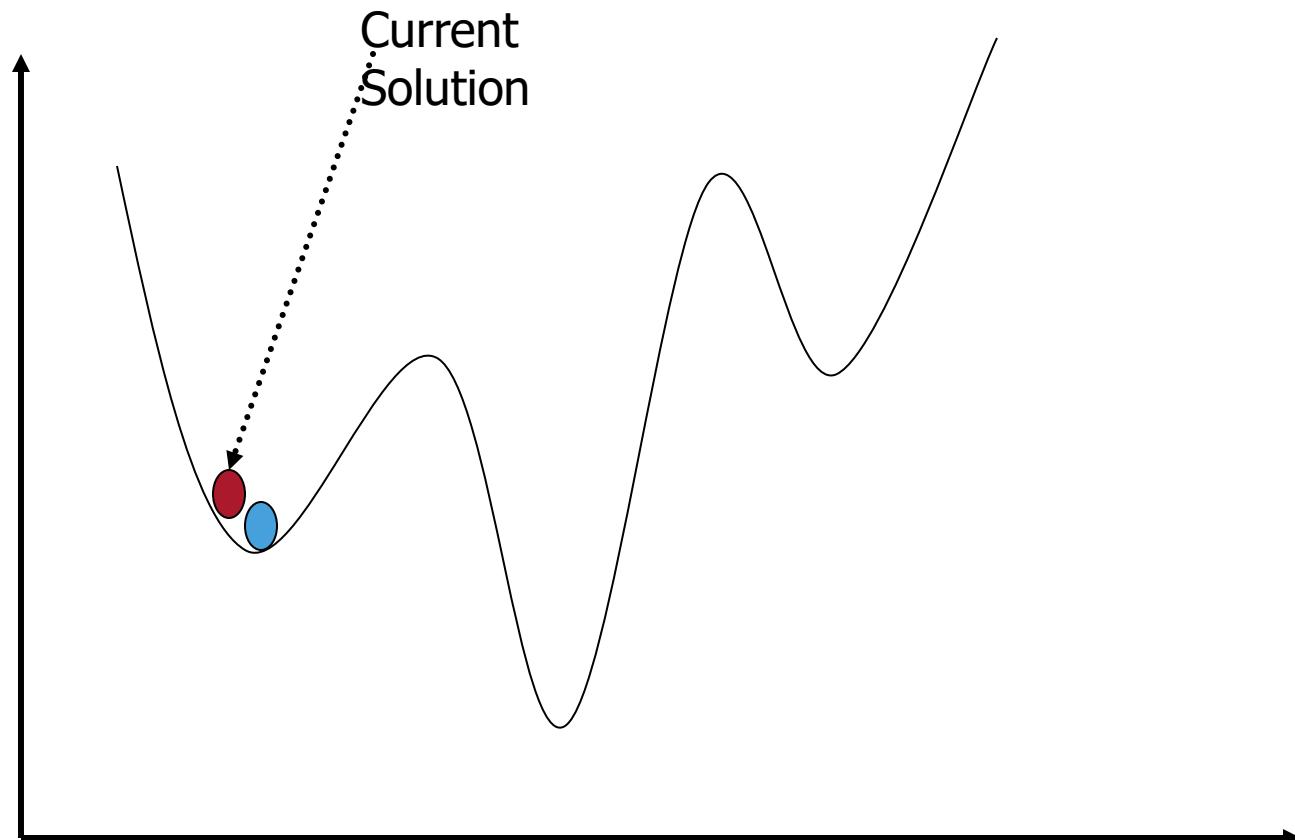
Hill-Climbing Search



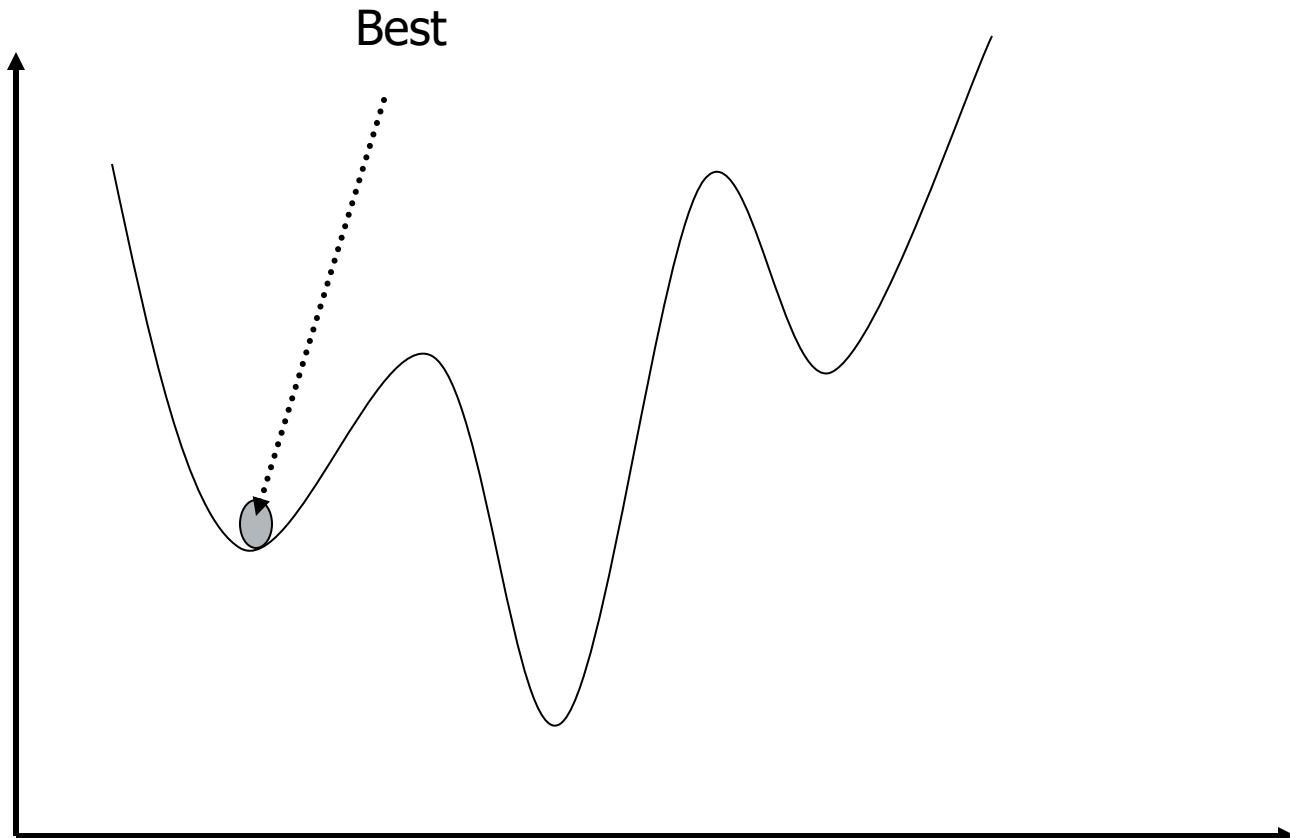
Hill-Climbing Search



Hill-Climbing Search



Hill-Climbing Search

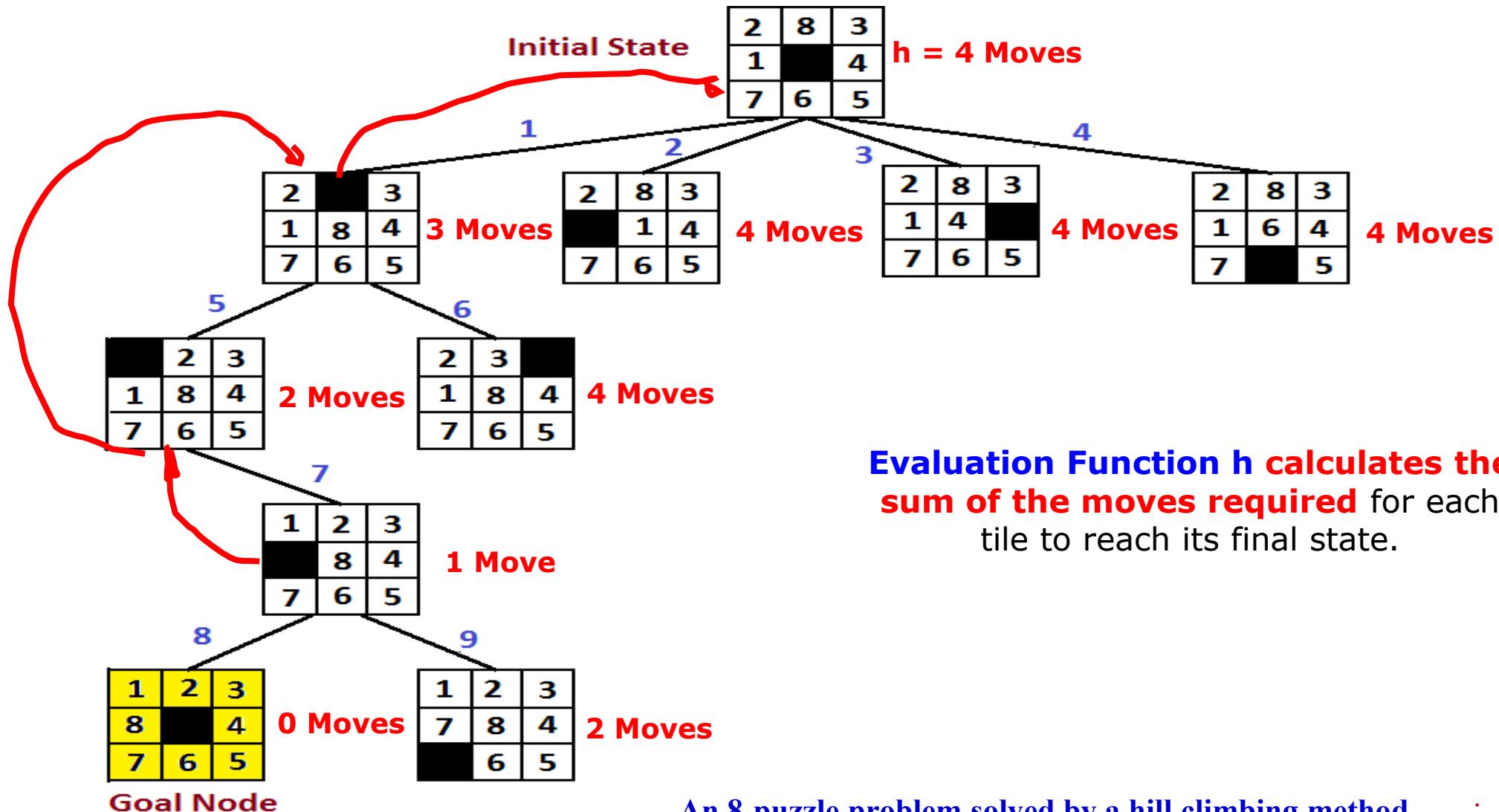


Hill-Climbing on the 8-Queens Problem

- Hill Climbing start with **an initial state** (i.e., a random configuration of the board) and chooses among the best successors. **Any move of a queen is one of the next possible states, i.e., a possible solution.**
- The **objective cost function h** is the number of pairs of queens that are attacking each other.
 - It counts as an attack if two states are in the same line, even if there is an intervening piece between them.
 - Objective Function = **$-h$** , i.e., the global maximum is **zero** at the perfect solution.
 - Objective Function = **h** , i.e., the global minimum is **zero** at the perfect solution.
- An 8-queens state with the current heuristic cost estimate **$h=17$** .
- The board shows the value of h for each possible successor obtained by moving a queen within its column.



Hill-Climbing on the 8-Puzzle Problem

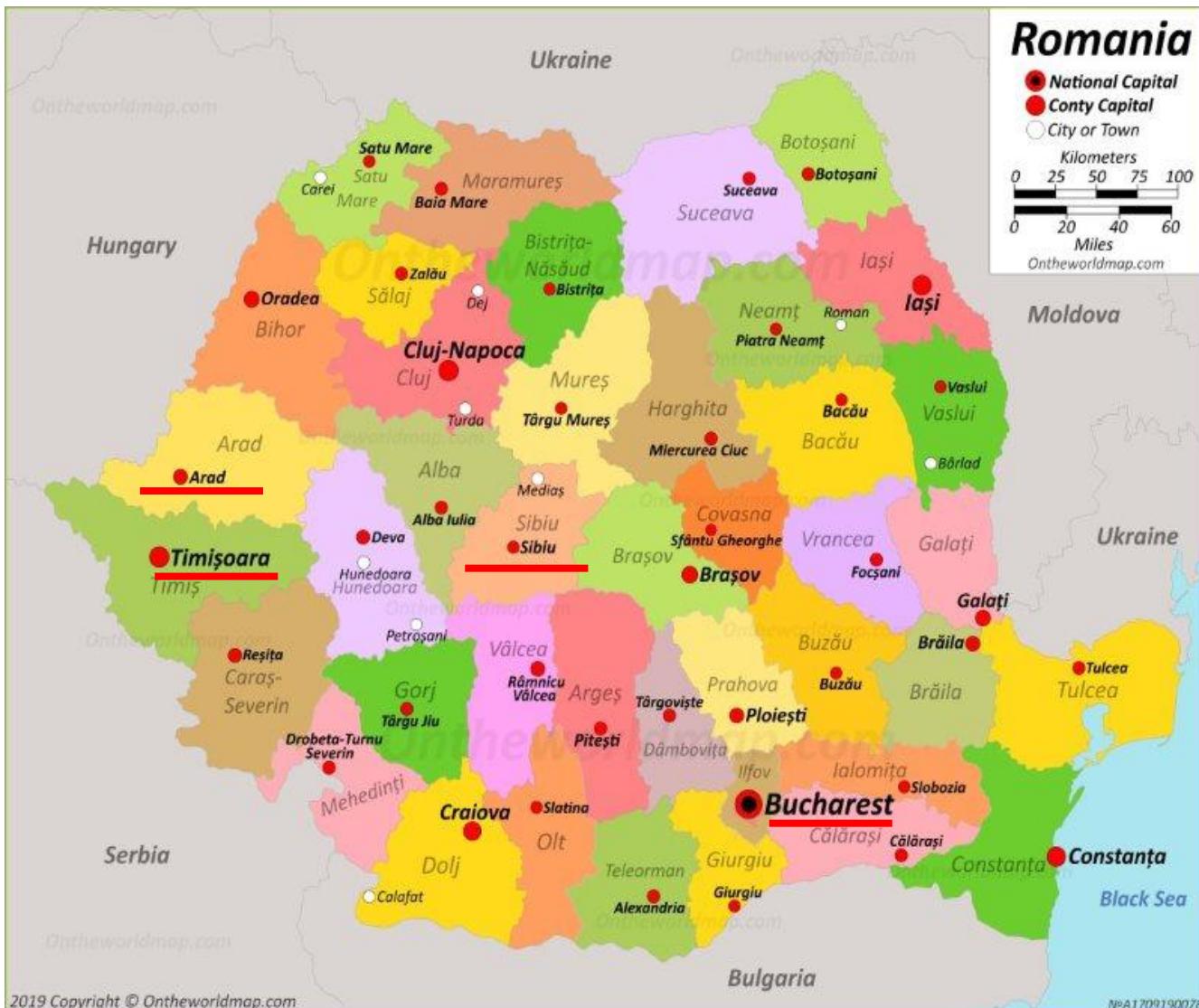


Evaluation Function h calculates the sum of the moves required for each tile to reach its final state.

An 8-puzzle problem solved by a hill climbing method. Worcester Polytechnic Institute

Hill-Climbing on the TSP Problem

- **TSP – Traveling Salesman Problem**
- The salesman must travel to all cities once before **returning the start city**.
- The distance between each city is given and is assumed to be the same in both directions.
- The objective is to **minimize the total distance** to be travelled.
- Actual Map is slightly different than the map, as an example, provided by the reference textbook.



Class Exercise 2

Review the **TSPProblem.py** on Canvas, <https://canvas.wpi.edu/courses/63988/files/6728931?wrap=1>, and do the following with your teammates/classmates

- What is the state defined in this problem? Is it one of the cities on the Romani map or a path solution?
- What is a neighbour state defined in this problem? How to get a neighbour state?
- In the `path_cost(self, c, state1, action, state2)` function, `c`, `state1`, and `action` are NOT used in the `value(self, state)` function. Why?
- In the `value(self, state)` function, why does it multiply -1 and `path_cost(self, c, state1, action, state2)` together? If it is the case, is the following statement correct inside the `hill_climbing(problem)`? Why?

```
if problem.value(neighbor.state) > problem.value(current.state):  
    current.state = neighbor.state
```

- Did you find any potential problem of each possible state solution? If yes, what is it?

Hill-Climbing on the TSP Problem

```
from search import *
import numpy as np

distances = {}
all_cities = []

class TSP_problem(Problem):
    """ subclass of Problem to define various functions """

    def two_opt(self, state):
        """ Neighbour generating function for Traveling Salesman Problem """
        neighbour_state = state[:]
        left = random.randint(0, len(neighbour_state) - 1)
        right = random.randint(0, len(neighbour_state) - 1)
        if left > right:
            left, right = right, left
        neighbour_state[left: right + 1] = reversed(neighbour_state[left: right + 1])
        return neighbour_state

    def actions(self, state):
        """ action that can be executed in given state """
        return [self.two_opt]

    def result(self, state, action):
        """ result after applying the given action on the given state """
        return action(state)

    def path_cost(self, c, state1, action, state2):
        """ total distance for the Traveling Salesman to be covered if in state2 """
        cost = 0
        for i in range(len(state2) - 1):
            cost += distances[state2[i]][state2[i + 1]]
        cost += distances[state2[0]][state2[-1]]
        return cost

    def value(self, state):
        """ value of path cost given negative for the given state """
        return -1 * self.path_cost(None, None, None, state)
```

Return random integers from low (inclusive) to high (exclusive)

```
def hill_climbing(problem):
    """From the initial node, keep choosing the neighbor with highest value,
    stopping when no neighbor is better. [Figure 4.2]"""

    def find_neighbors(state, number_of_neighbors=100):
        """ finds neighbors using two_opt method """

        neighbors = []

        for i in range(number_of_neighbors):
            new_state = problem.two_opt(state)
            neighbors.append(Node(new_state))
            state = new_state

        return neighbors

# as this is a stochastic algorithm, we will set a cap on the number of iterations
iterations = 10000

current = Node(problem.initial)

while iterations:
    neighbors = find_neighbors(current.state)
    if not neighbors:
        break
    neighbor = argmax_random_tie(neighbors,
                                  key=lambda node: problem.value(node.state))
    if problem.value(neighbor.state) > problem.value(current.state):
        current.state = neighbor.state
    iterations -= 1

return current.state
```

Hill-Climbing on the TSP Problem

```
def main():
    for city in romania_map.locations.keys():
        distances[city] = {}
        all_cities.append(city)
```

```
all_cities.sort()
print("All the sorted cities in Romania:")
print(all_cities)
print()
```

We populate the individual lists inside the dictionary with the Manhattan distance between the cities.

```
for name_1, coordinates_1 in romania_map.locations.items():
```

```
    for name_2, coordinates_2 in romania_map.locations.items():
        distances[name_1][name_2] = np.linalg.norm(
            [coordinates_1[0] - coordinates_2[0], coordinates_1[1] - coordinates_2[1]])
        distances[name_2][name_1] = np.linalg.norm(
            [coordinates_1[0] - coordinates_2[0], coordinates_1[1] - coordinates_2[1]])
```

```
tsp = TSP_problem(all_cities)
```

```
print("One shortest possible route that visits each city exactly once and returns to the origin city:")
print(hill_climbing(tsp))
```

```
if __name__ == "__main__":
    main()
```

```
|class Problem:
```

```
|    """The abstract class for a formal problem. You should subclass  
|    this and implement the methods actions and result, and possibly  
|    __init__, goal_test, and path_cost. Then you will create instances  
|    of your subclass and solve them with the various search functions."""
|
```

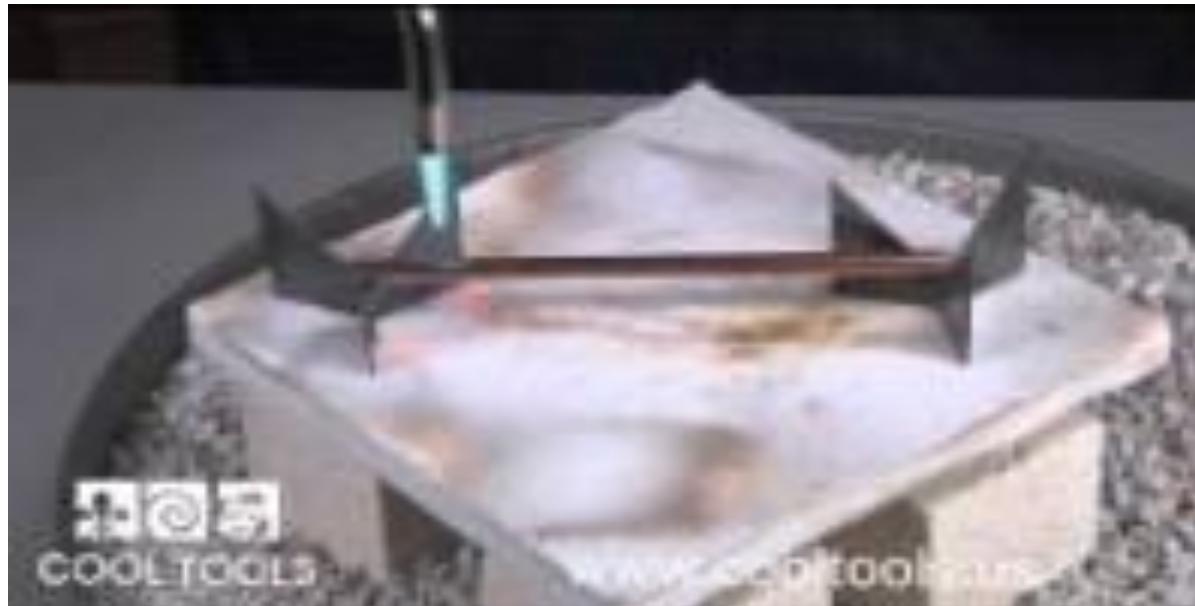
```
| def __init__(self, initial, goal=None):
```

```
|     """The constructor specifies the initial state, and possibly a goal  
|     state, if there is a unique goal. Your subclass's constructor can add  
|     other arguments."""
|     self.initial = initial
|     self.goal = goal
```

$$T = \frac{T_0}{1 + \log(t)}$$

Simulated Annealing

- Due to the **three** problems of Hill Climbing, Simulated Annealing can help us avoid getting **stuck in a Local Maxima/Minima, Ridges, and Plateaus**.
- Annealing is the metallurgical process of heating up a solid and then cooling it slowly until it crystallizes.
- Simulated Annealing (SA) is to mimic the annealing process, where a function $E(S)$ needs to be minimized.
 - This function is analogous to the internal energy in that state S .
 - The goal is to start from **an initial state** to a state having minimum possible energy.



function SIMULATED-ANNEALING(*problem, schedule*) returns a solution state

current \leftarrow *problem.INITIAL*

for $t = 1$ to ∞ do

$T \leftarrow \text{schedule}(t)$

if $T = 0$ then return *current*

next \leftarrow a randomly selected successor of *current*

$\Delta E \leftarrow \text{VALUE}(\text{current}) - \text{VALUE}(\text{next})$

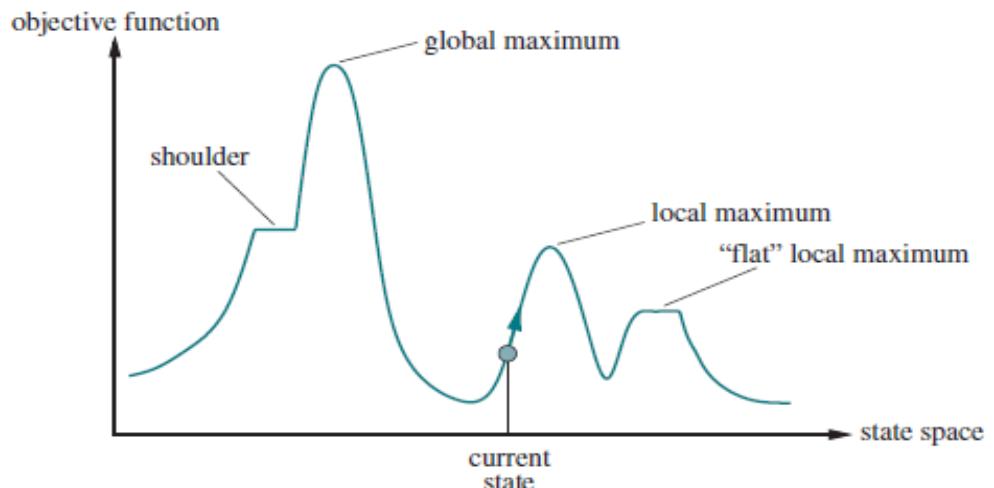
if $\Delta E > 0$ then *current* \leftarrow *next*

else *current* \leftarrow *next* only with probability $e^{-\Delta E/T}$

Temperature $T \geq 0$ in Kelvin absolute temperature scale.

$T = 0$, i.e., Freezing Point

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The schedule input determines the value of the “temperature” T as a function of time.



$$T = \frac{T_0}{1 + \log(t)}$$

Simulated Annealing

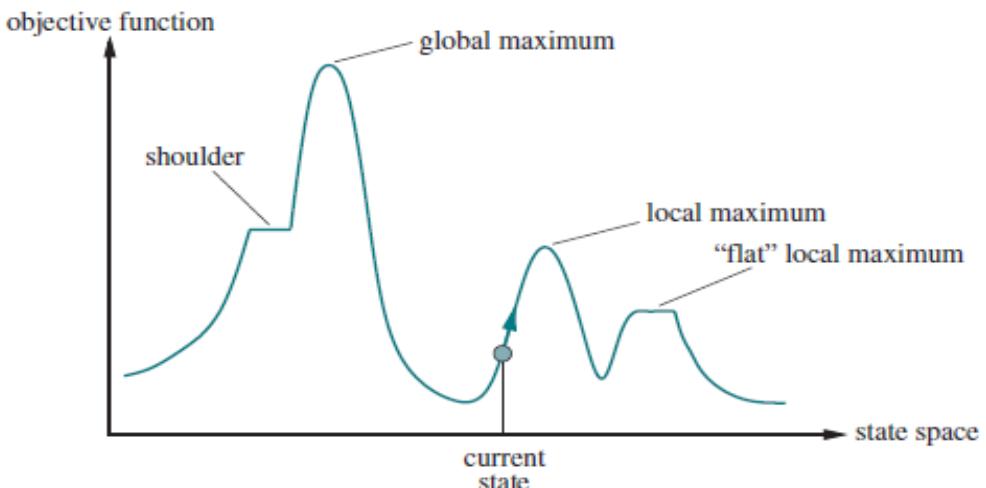
- Basic idea:
 - Allow “random” moves occasionally, depending on “high temperature”
 - High temperature → more random moves allowed, **shake the process out of its local minimum or local maximum**
 - schedule, i.e., annealing schedule, that leads to a better solution.
 - If the new state is **not better**, we make it the current state with **a certain predefined probability** by using a random number generator and deciding based on a threshold. If it is **above the threshold**, we set the current state to the next state.
- The **simulated_annealing(problem, schedule=exp_schedule())** function in search.py.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  current  $\leftarrow$  problem.INITIAL
  for t = 1 to  $\infty$  do
    T  $\leftarrow$  schedule(t)
    if T = 0 then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow \text{VALUE}(\text{current}) - \text{VALUE}(\text{next})$ 
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Temperature $T \geq 0$ in Kelvin absolute temperature scale.

$T = 0$, i.e., Freezing Point

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The schedule input determines the value of the “temperature” T as a function of time.



Class Exercise 3

Review the **Simulated Annealing** pseudocode in Slide #19 and do the following with your teammates/classmates

- If the solution/state value computed is a profit value, is it “if $\Delta E > 0$ then $current \leftarrow next$ ” correct? Do you need to do any modification on the code? Why?
- If the solution/state value computed is a +ve cost value, is it “if $\Delta E > 0$ then $current \leftarrow next$ ” correct? Do you need to do any modification on the code? Why?
- If the solution/state value computed is a -ve cost value, is it “if $\Delta E > 0$ then $current \leftarrow next$ ” correct? Do you need to do any modification on the code? Why?

Simulated Annealing

```


$$20 * e^{-0.005*t}$$

def exp_schedule(k=20, lam=0.005, limit=100):
    """One possible schedule function for simulated annealing"""
    return lambda t: (k * np.exp(-lam * t) if t < limit else 0)

def simulated_annealing(problem, schedule=exp_schedule()):
    """[Figure 4.5] CAUTION: This differs from the pseudocode as it
    returns a state instead of a Node."""
    current = Node(problem.initial)
    for t in range(sys.maxsize):
        T = schedule(t)
        if T == 0:
            return current.state
        neighbors = current.expand(problem)
        if not neighbors:
            return current.state
        next_choice = random.choice(neighbors)
        #delta_e = problem.value(next_choice.state) - problem.value(current.state)
        delta_e = problem.value(current.state) - problem.value(next_choice.state)
        if delta_e > 0 or probability(np.exp(-delta_e / T)):
            current = next_choice
    
```

utils.py

```

def probability(p):
    """Return true with probability p."""
    return p > random.uniform(0.0, 1.0)

```

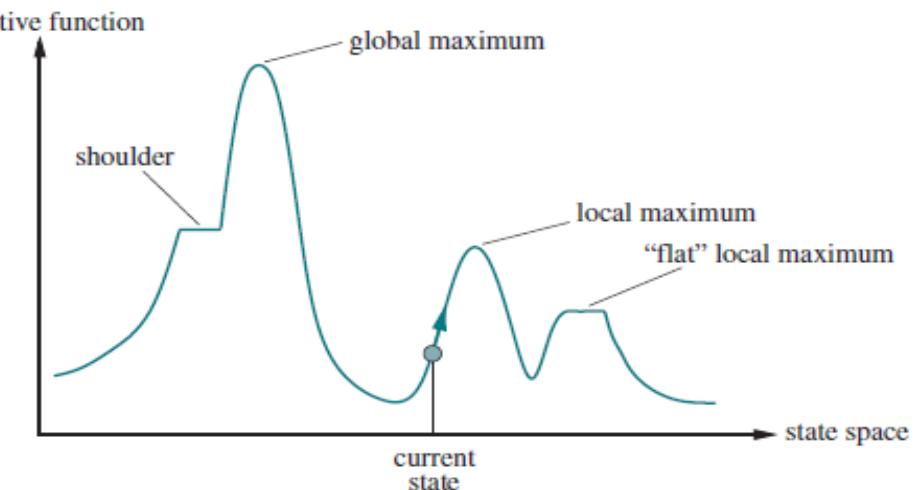
returns a randomly selected element from the specified sequence.

$$T = \frac{T_0}{1 + \log(t)}$$

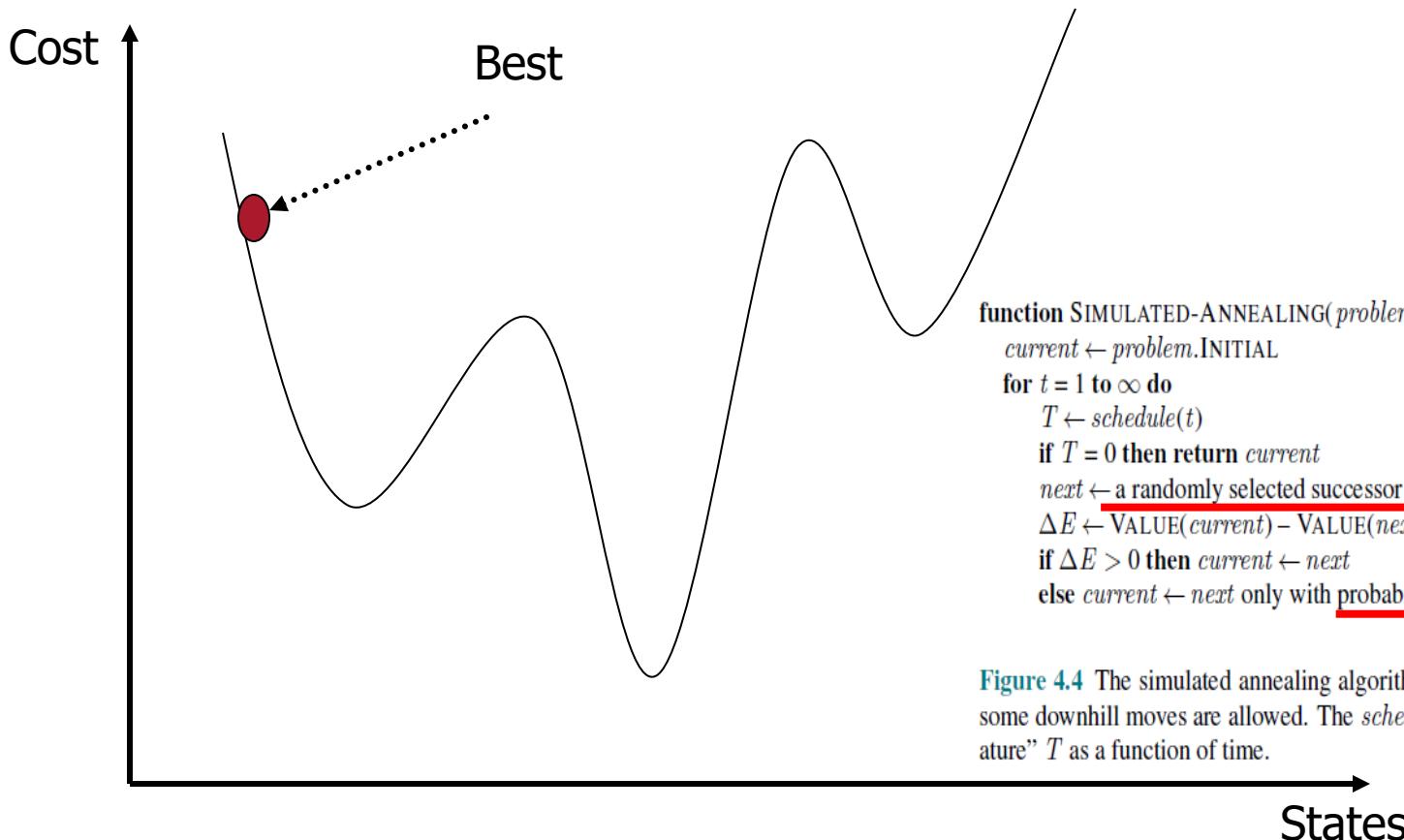
function SIMULATED-ANNEALING(*problem, schedule*) returns a solution state
 $\text{current} \leftarrow \text{problem.INITIAL}$
 $\text{for } t = 1 \text{ to } \infty \text{ do}$
 $\quad T \leftarrow \text{schedule}(t)$
 $\quad \text{if } T = 0 \text{ then return } \text{current}$
 $\quad \text{next} \leftarrow \text{a randomly selected successor of } \text{current}$
 $\quad \Delta E \leftarrow \text{VALUE}(\text{current}) - \text{VALUE}(\text{next})$
 $\quad \text{if } \Delta E > 0 \text{ then } \text{current} \leftarrow \text{next}$
 $\quad \text{else } \text{current} \leftarrow \text{next} \text{ only with probability } e^{-\Delta E/T}$

Temperature $T \geq 0$ in Kelvin absolute temperature scale.
 $T = 0$, i.e., Freezing Point

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” T as a function of time.



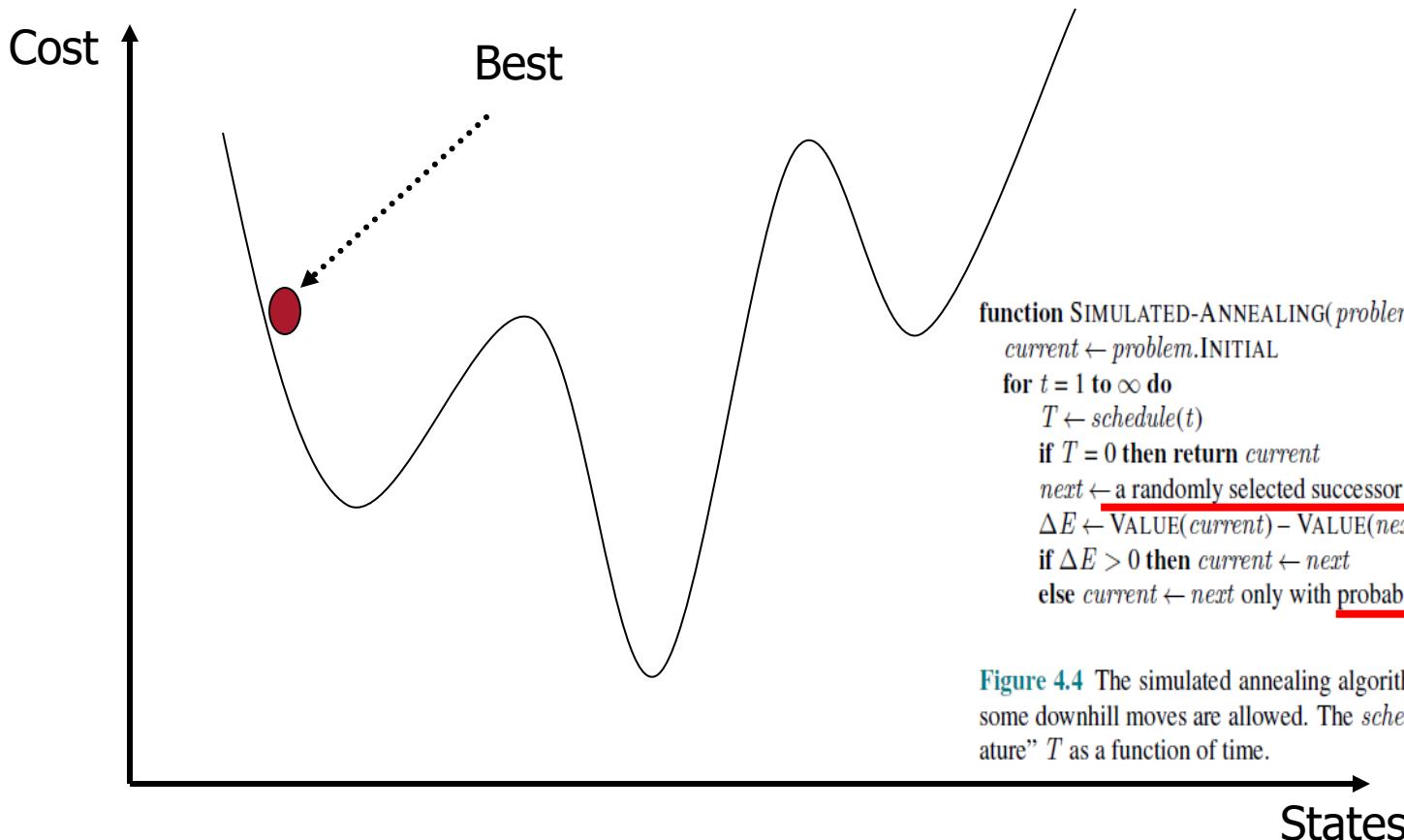
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

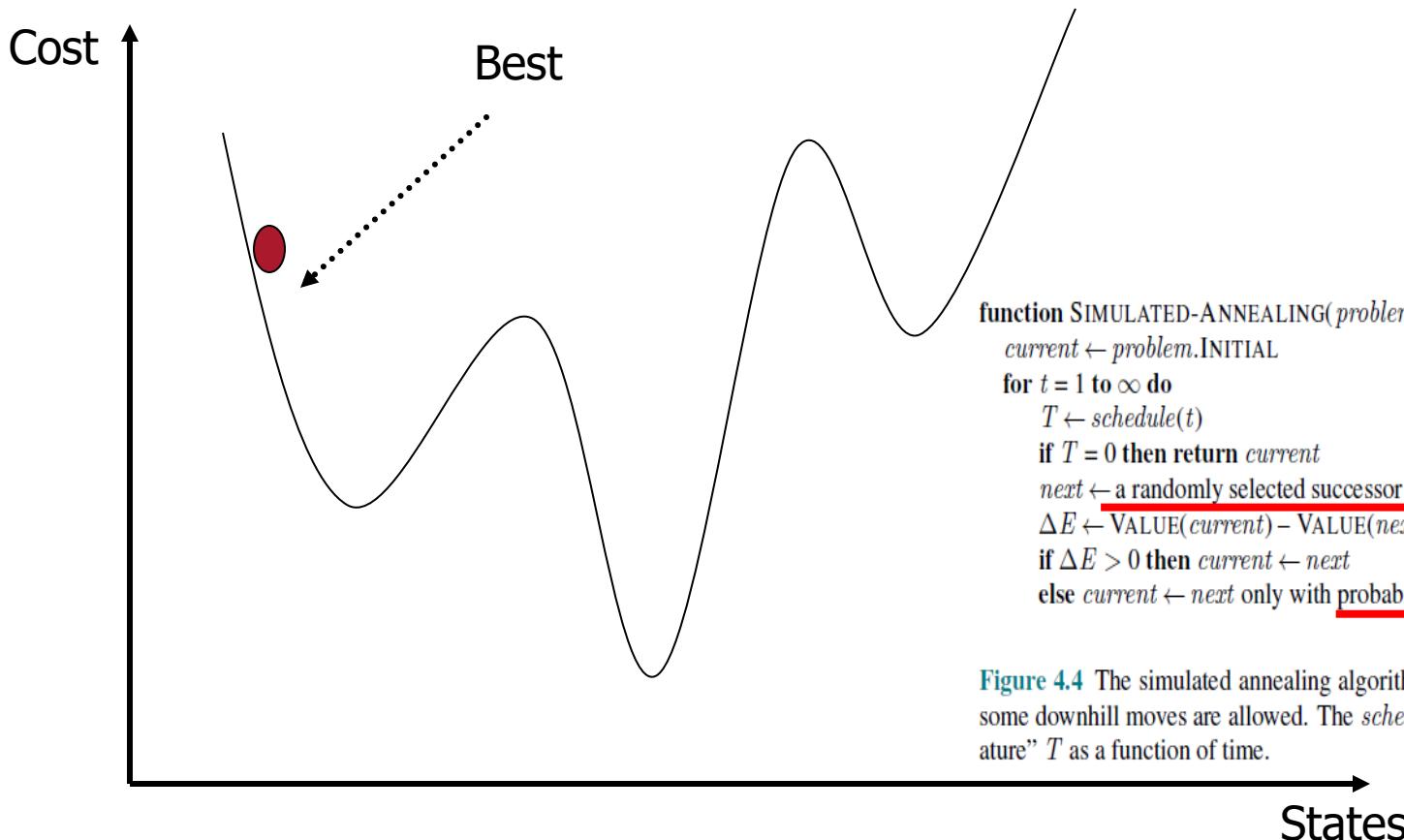
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” T as a function of time.

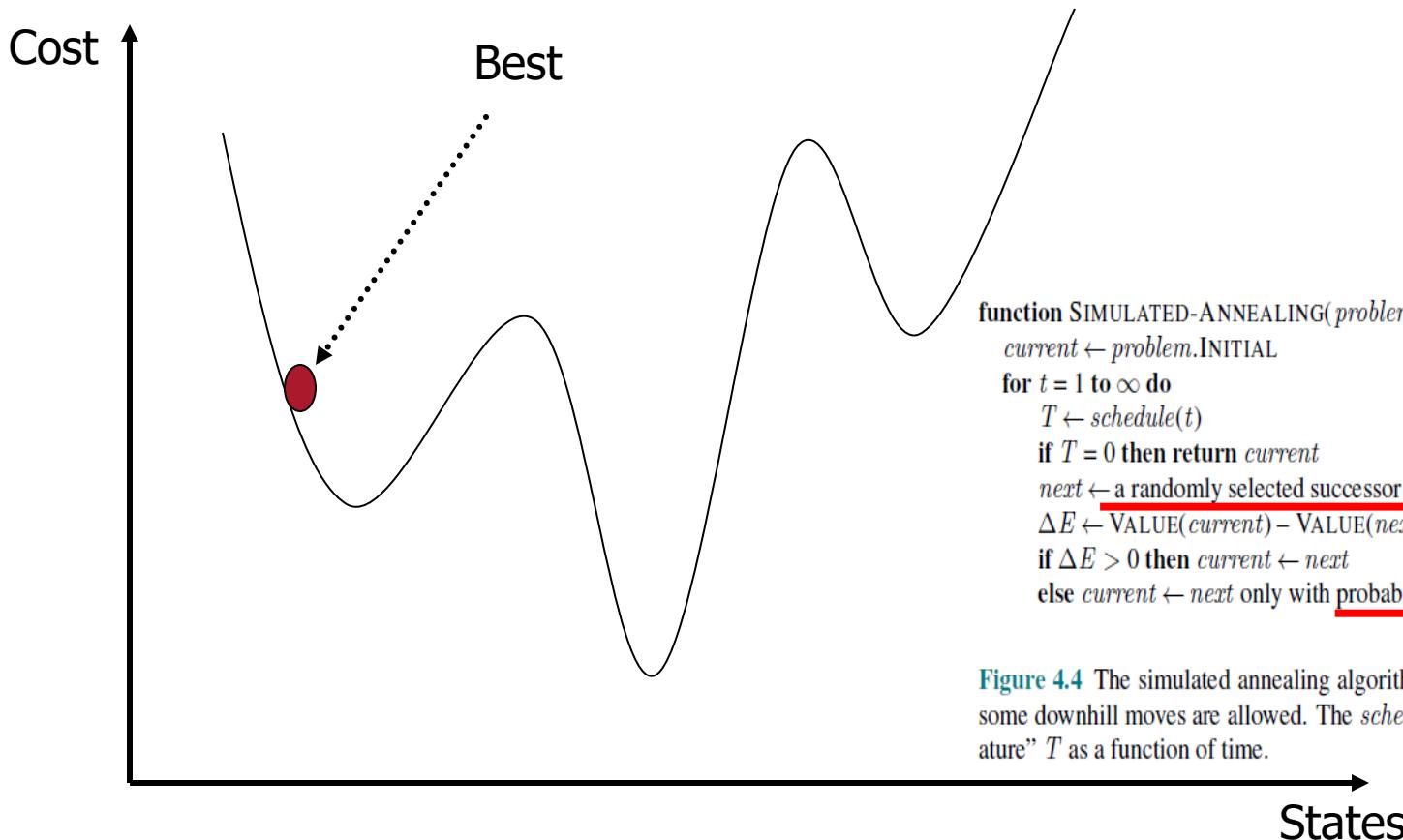
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

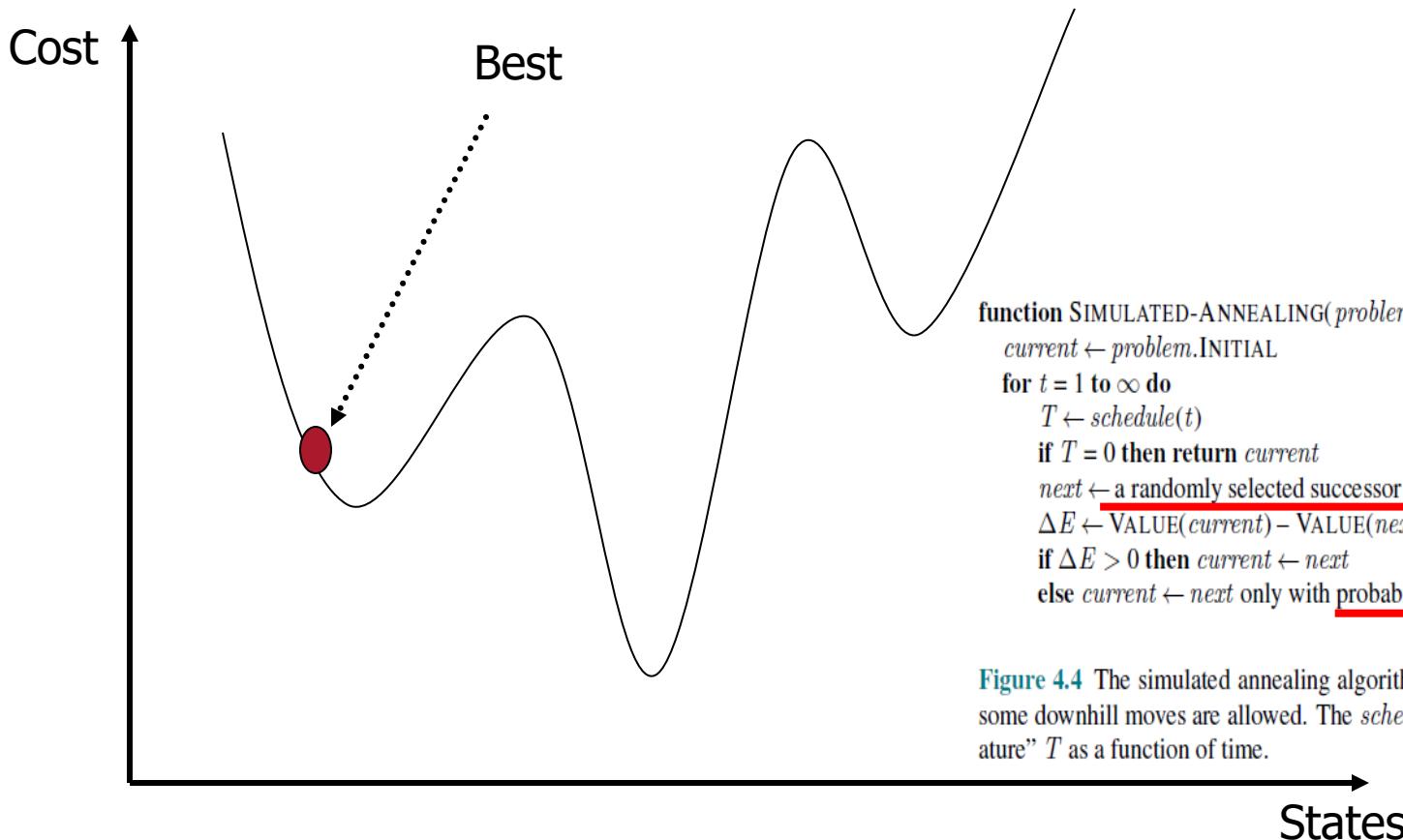
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

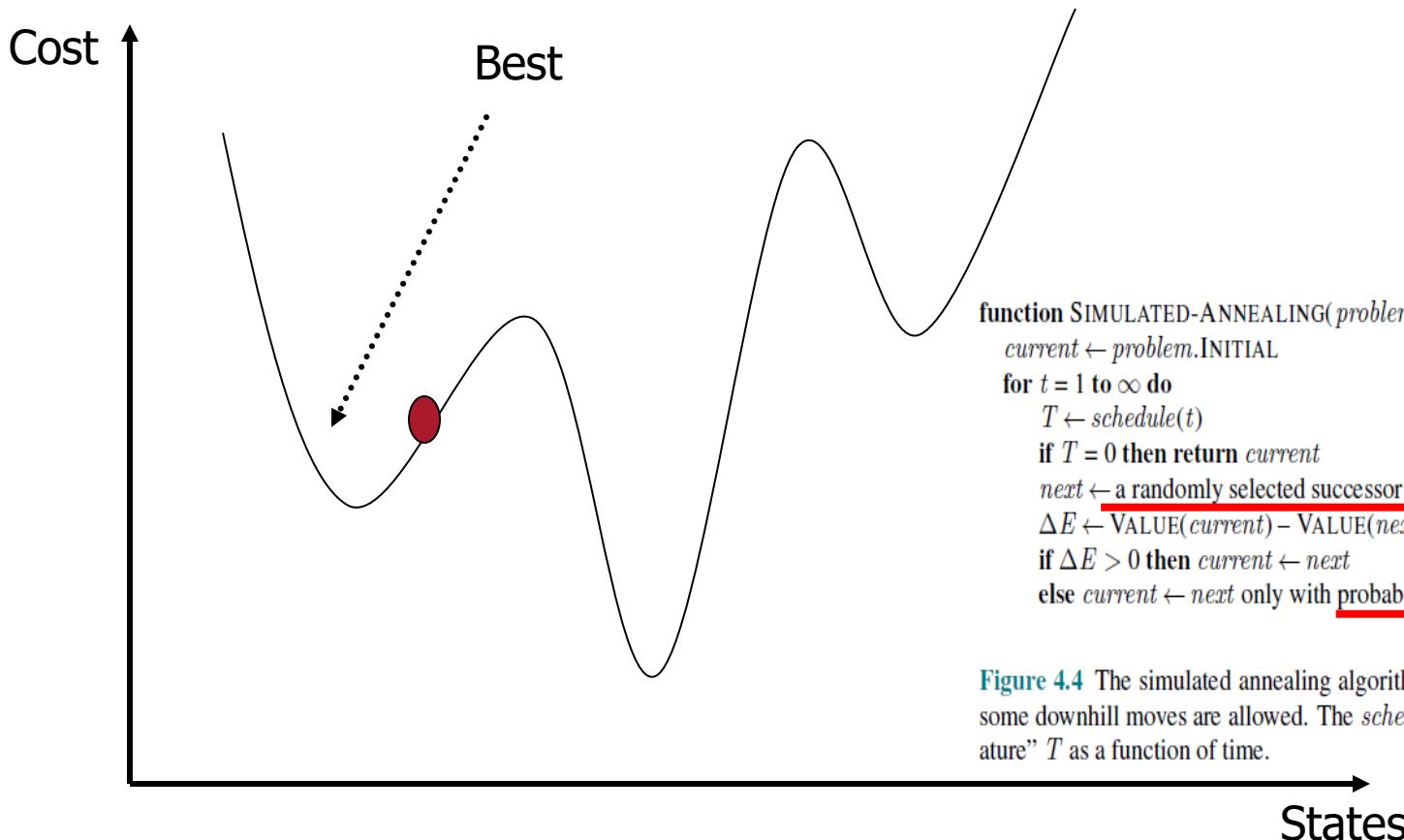
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

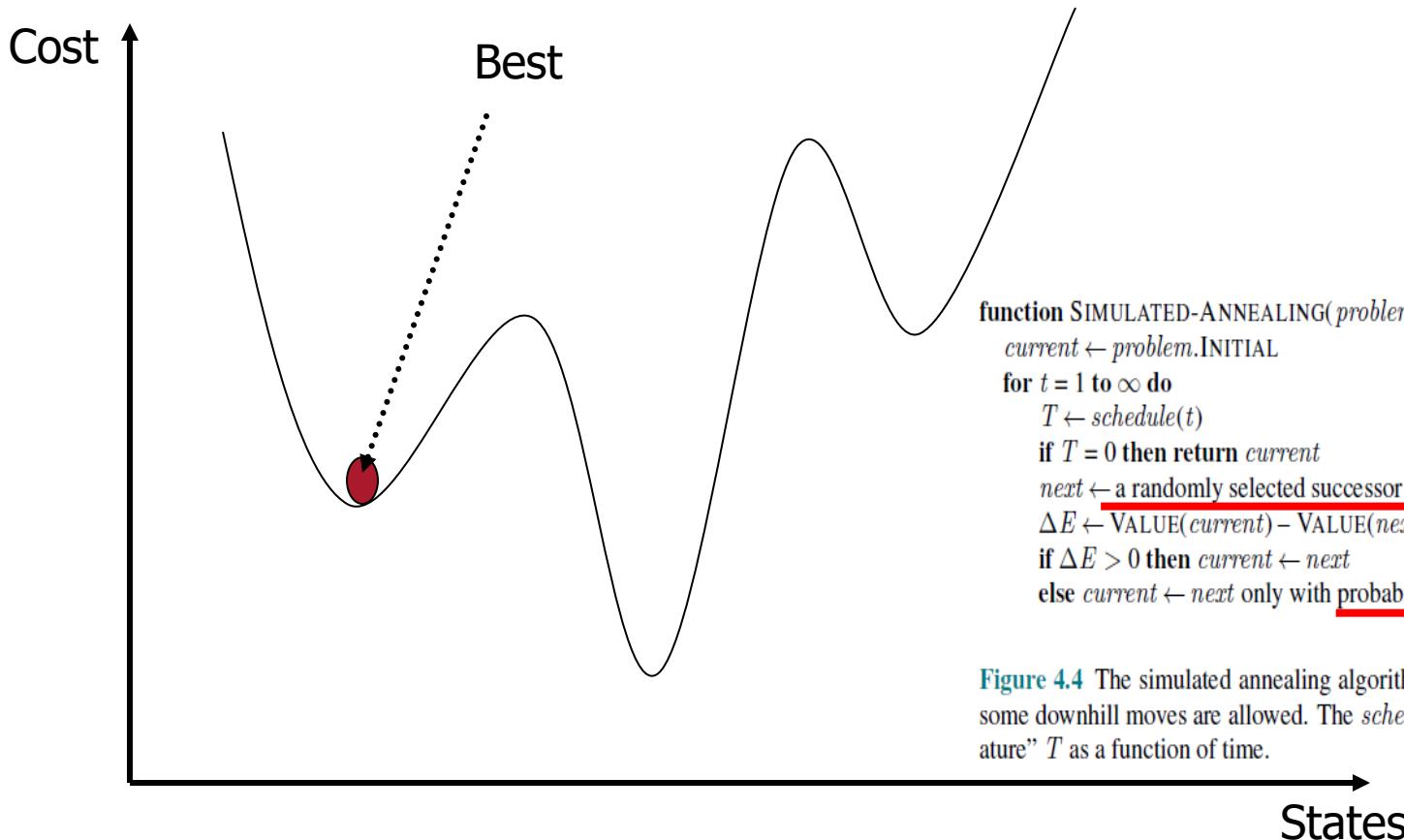
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” T as a function of time.

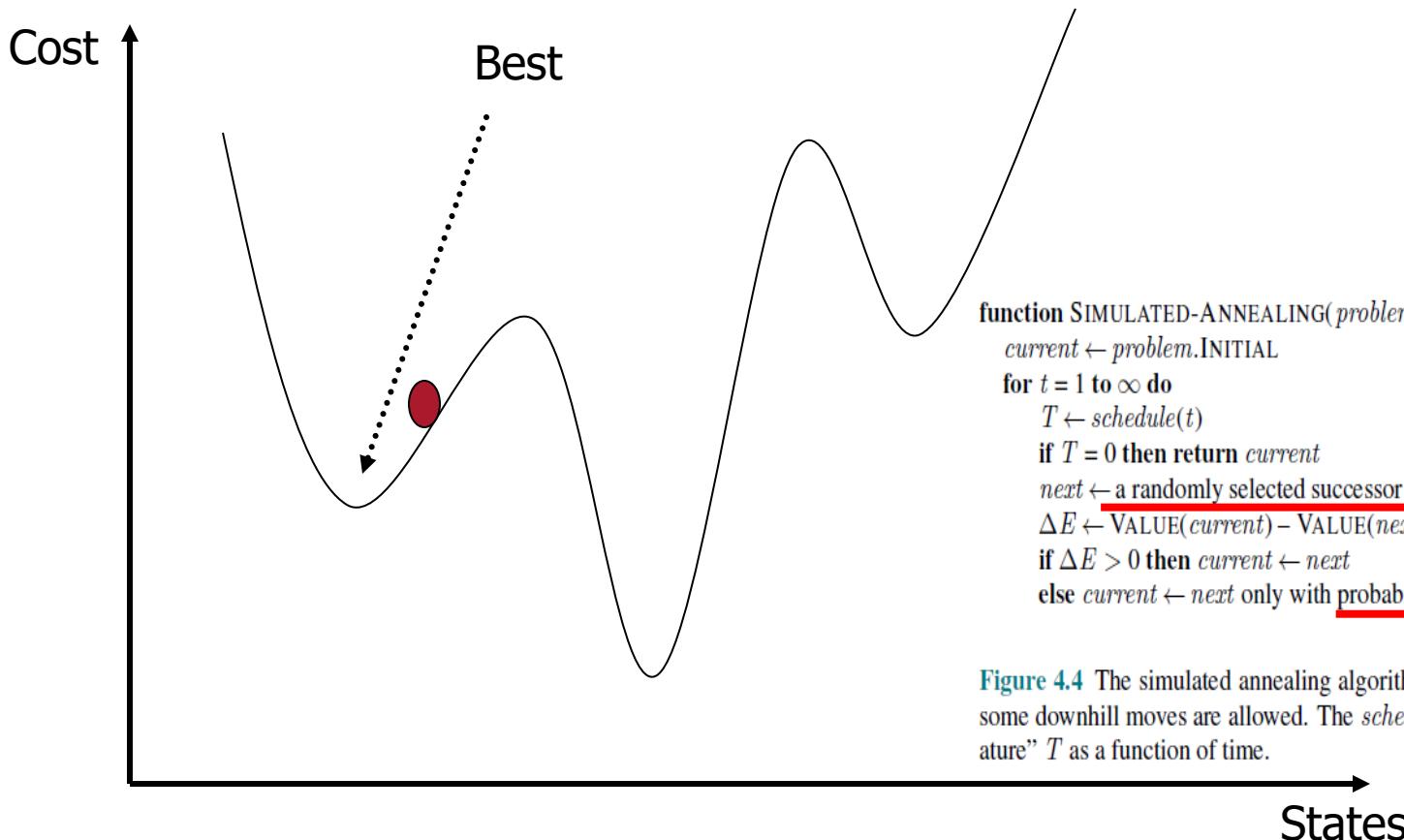
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

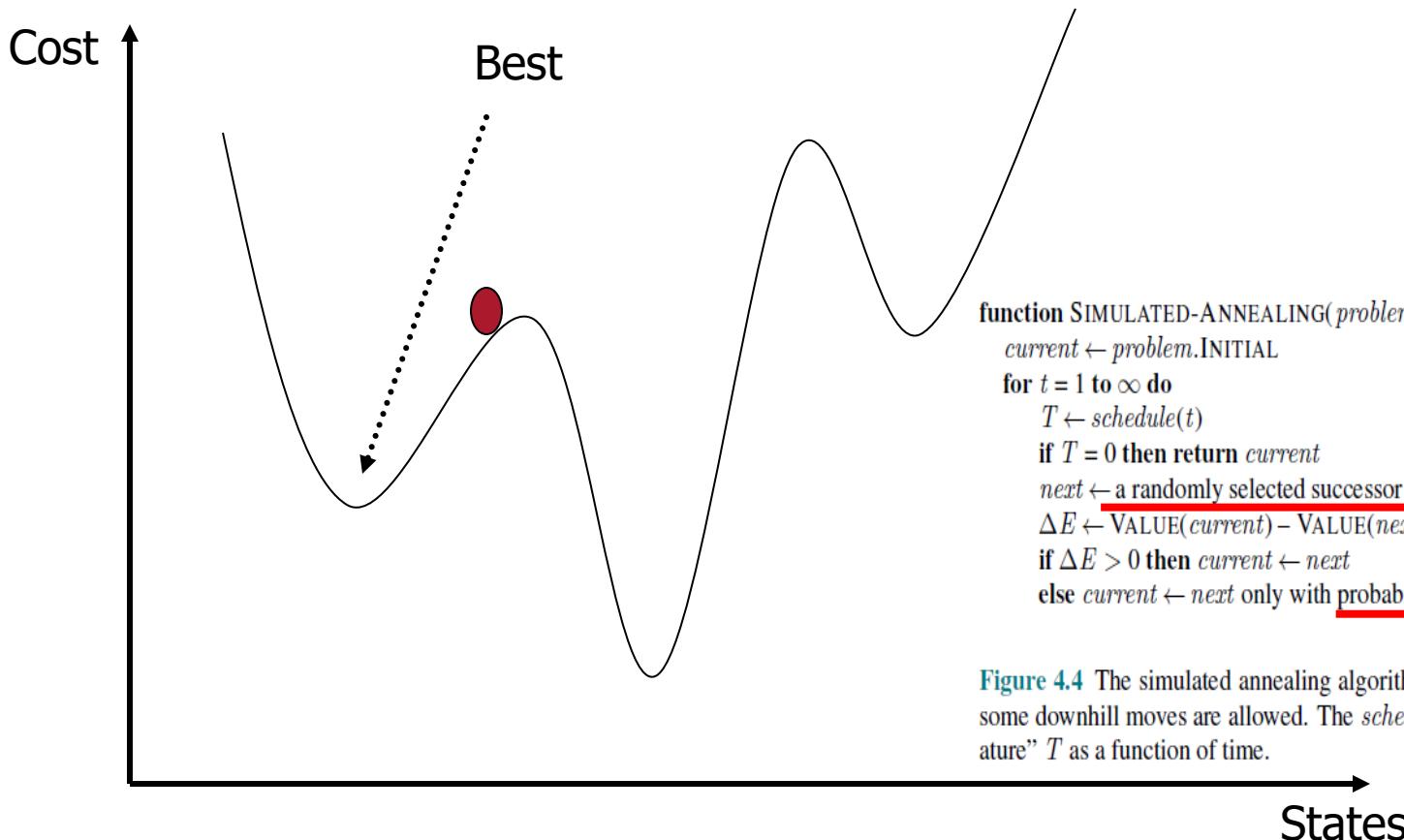
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

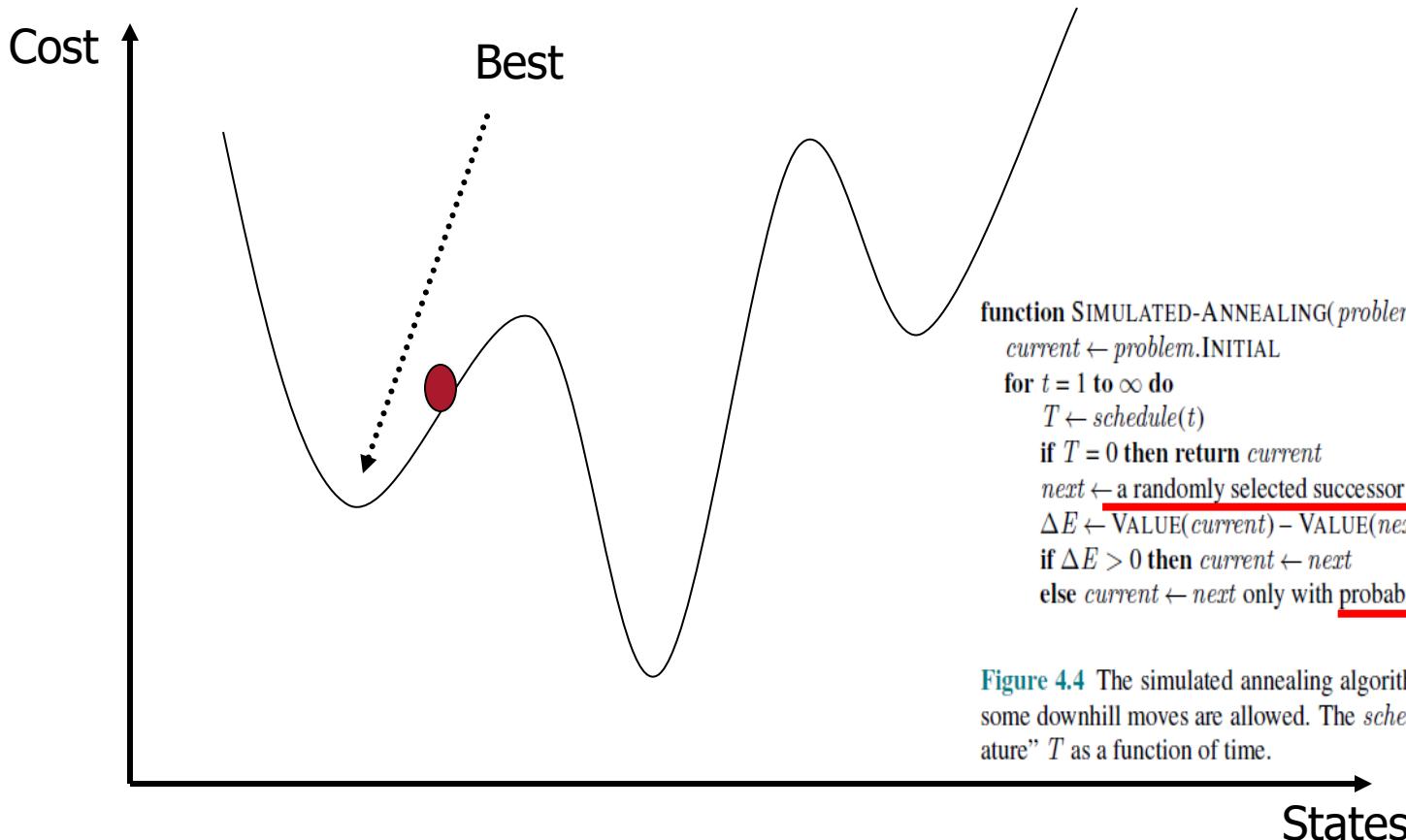
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” T as a function of time.

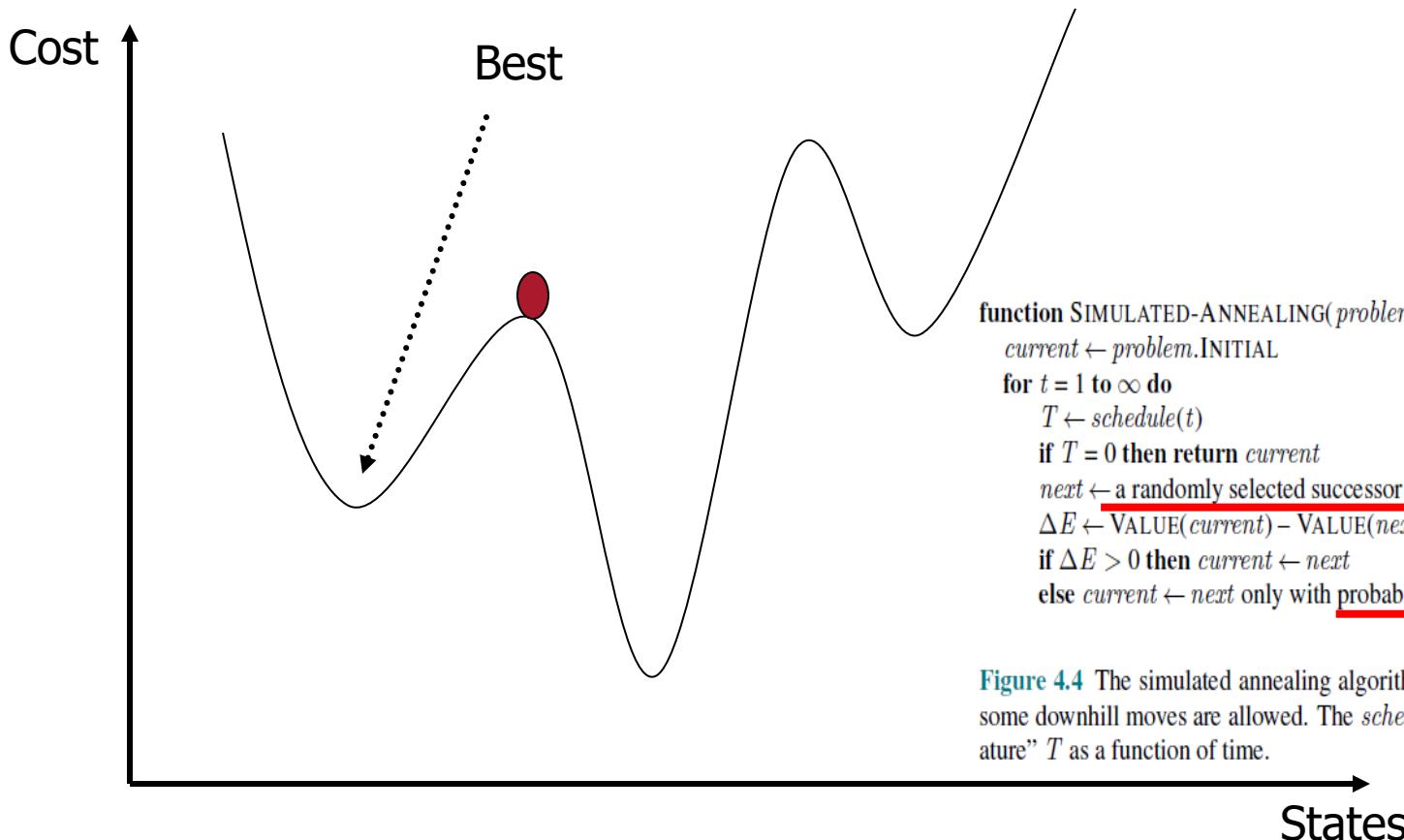
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

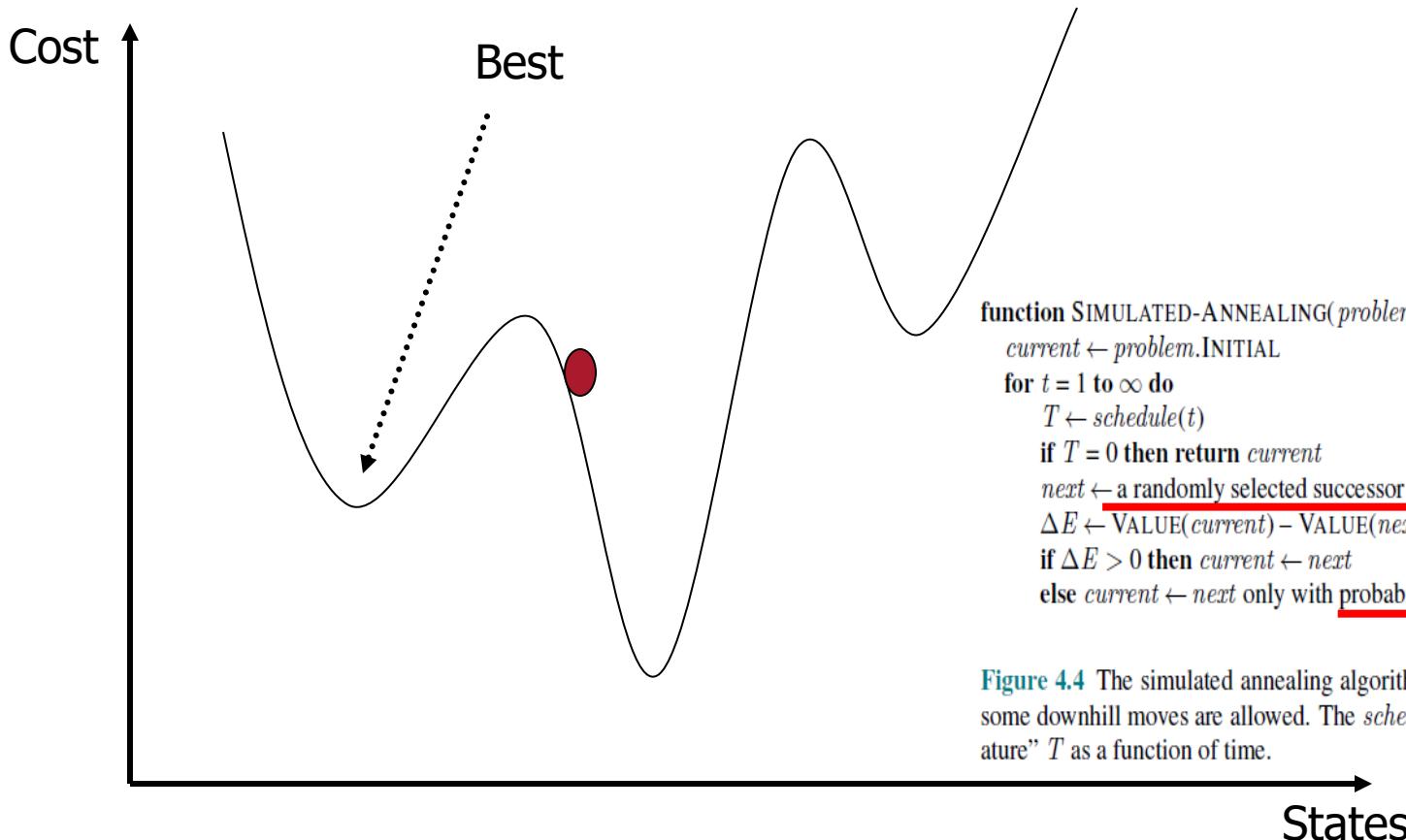
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” T as a function of time.

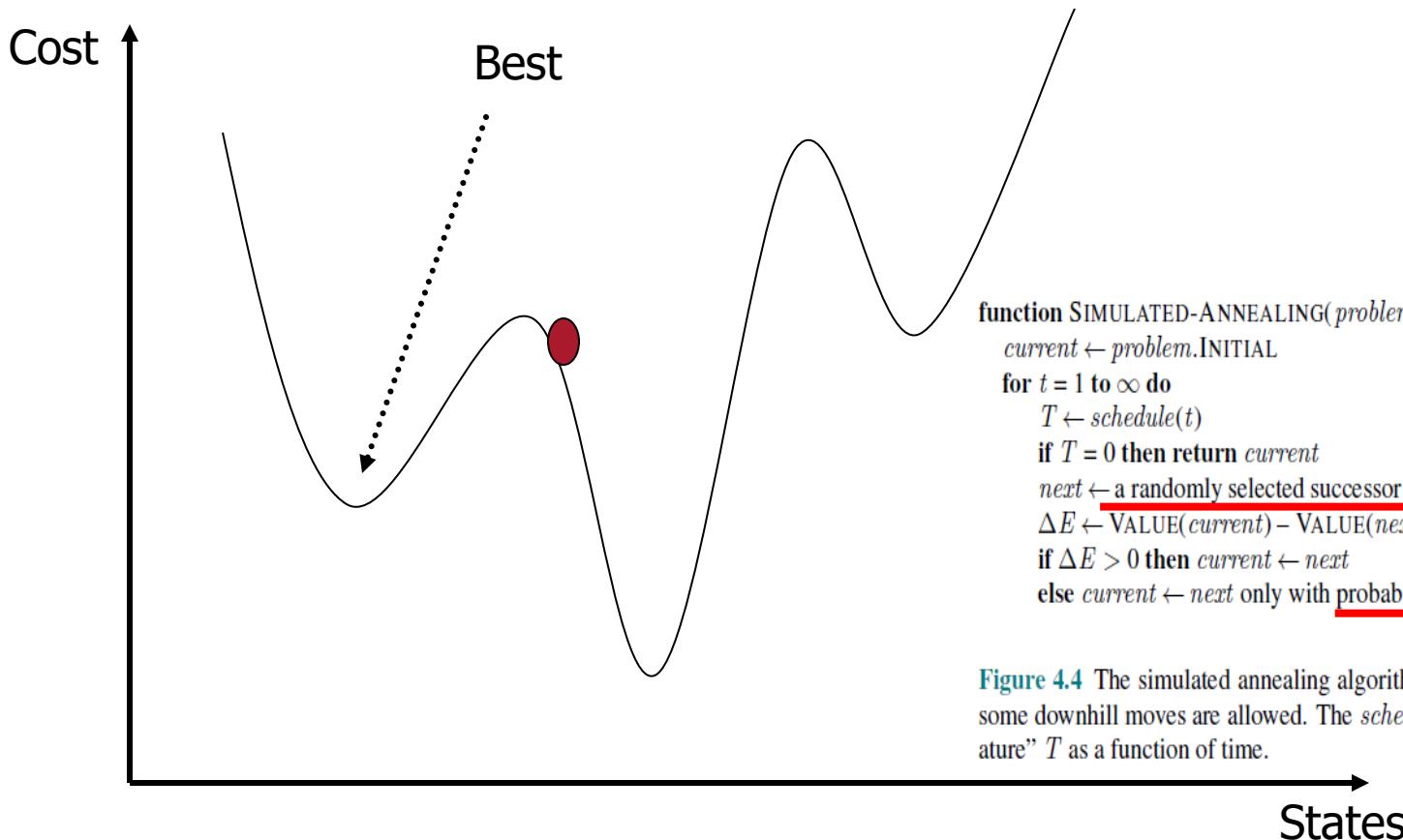
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

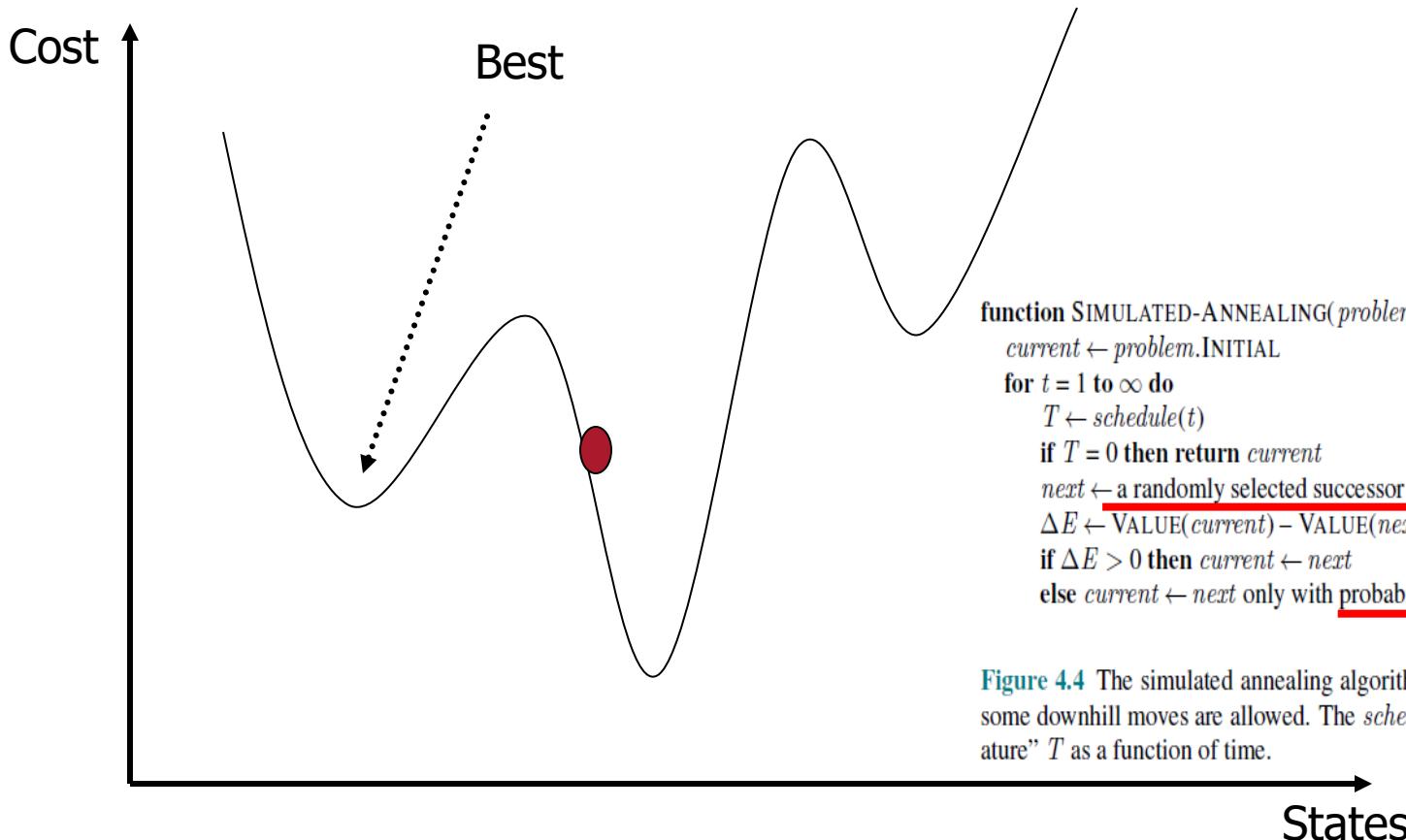
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

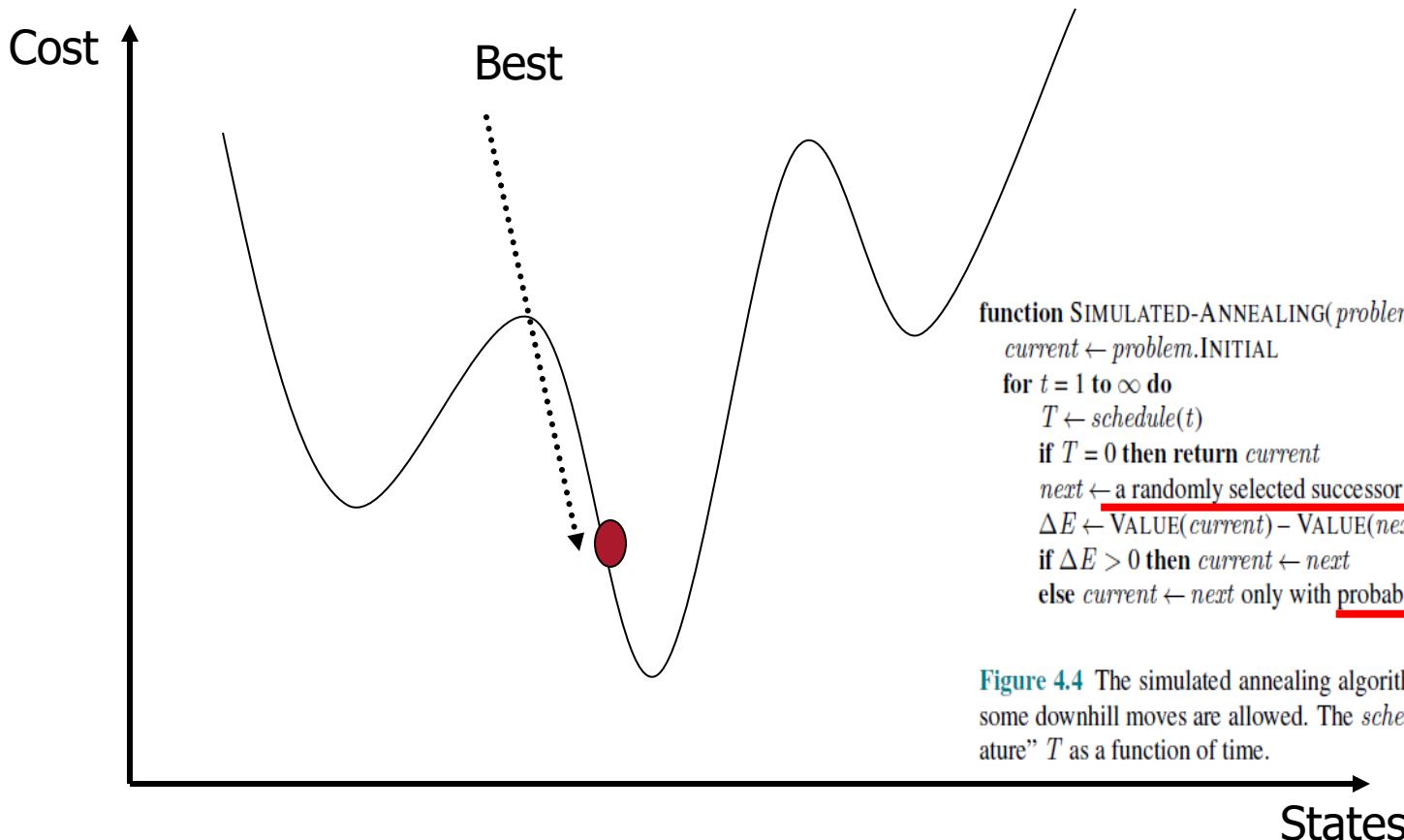
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

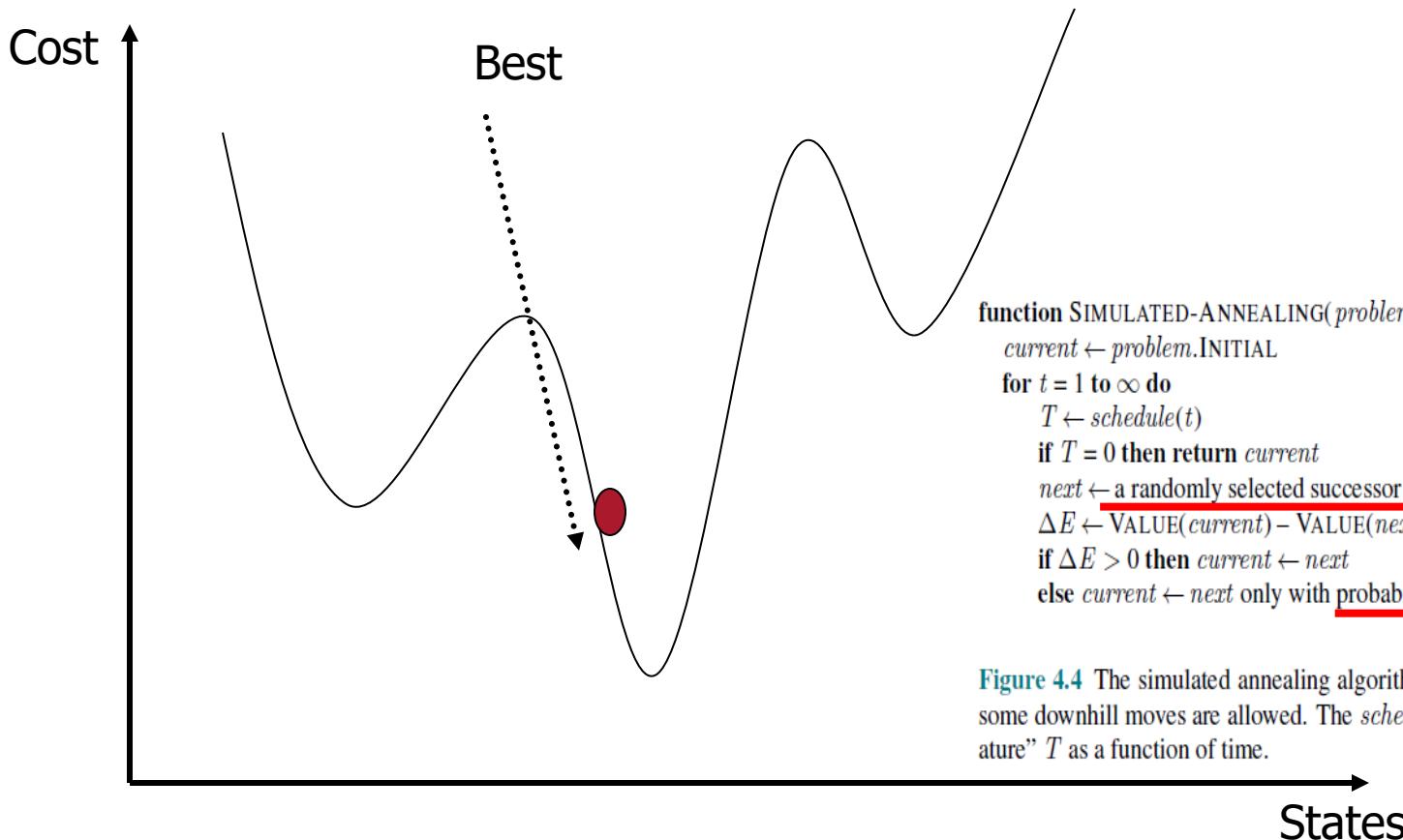
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

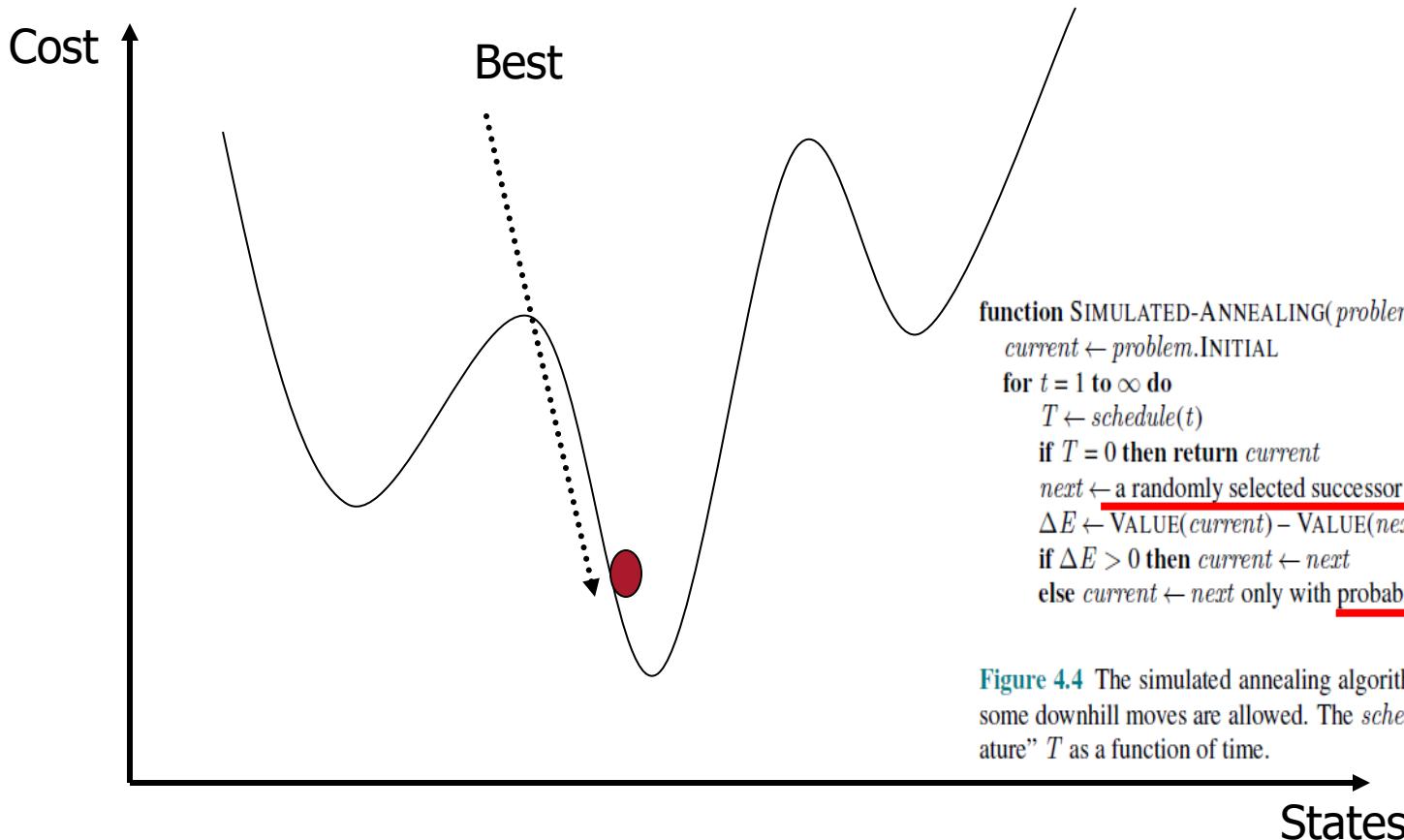
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

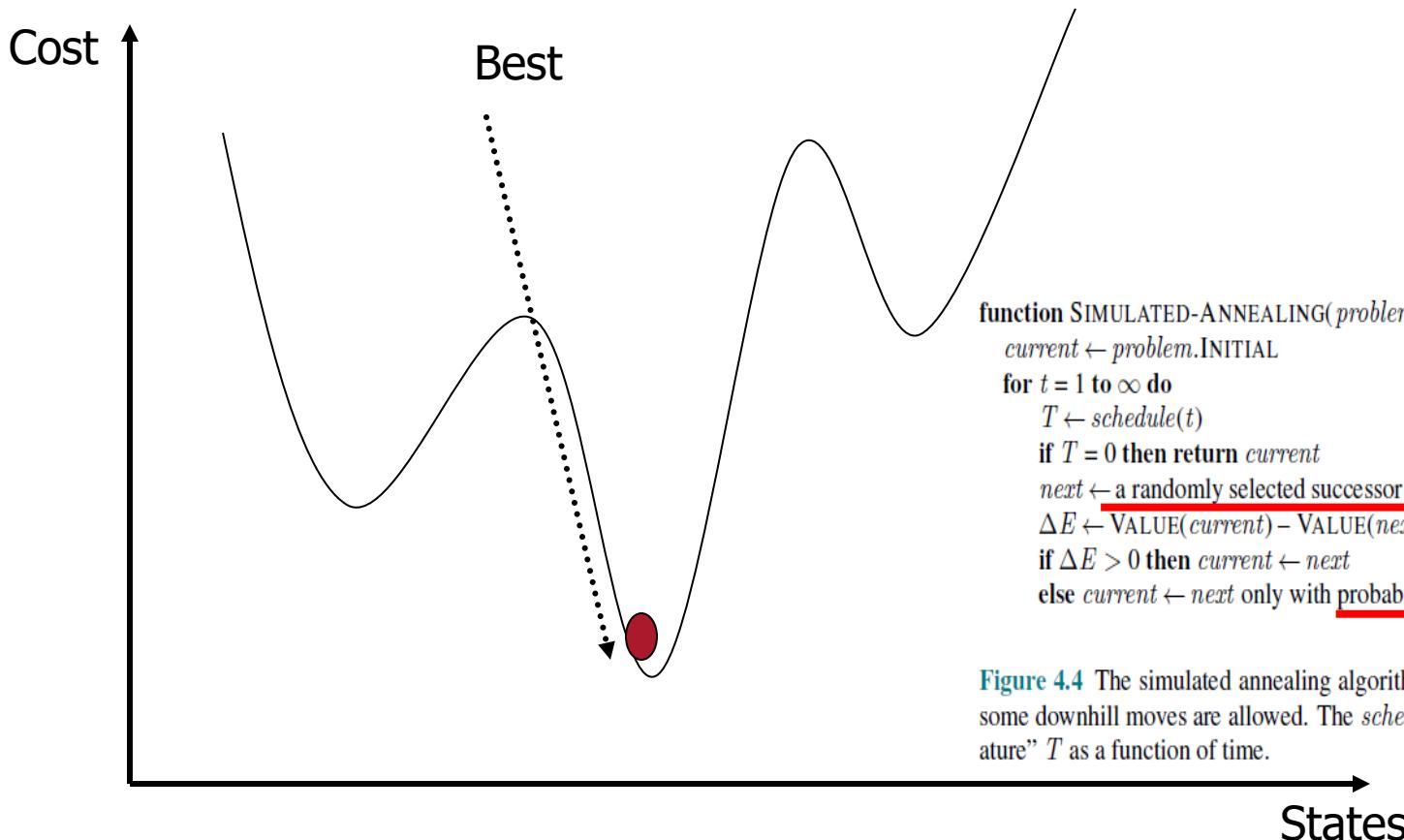
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” T as a function of time.

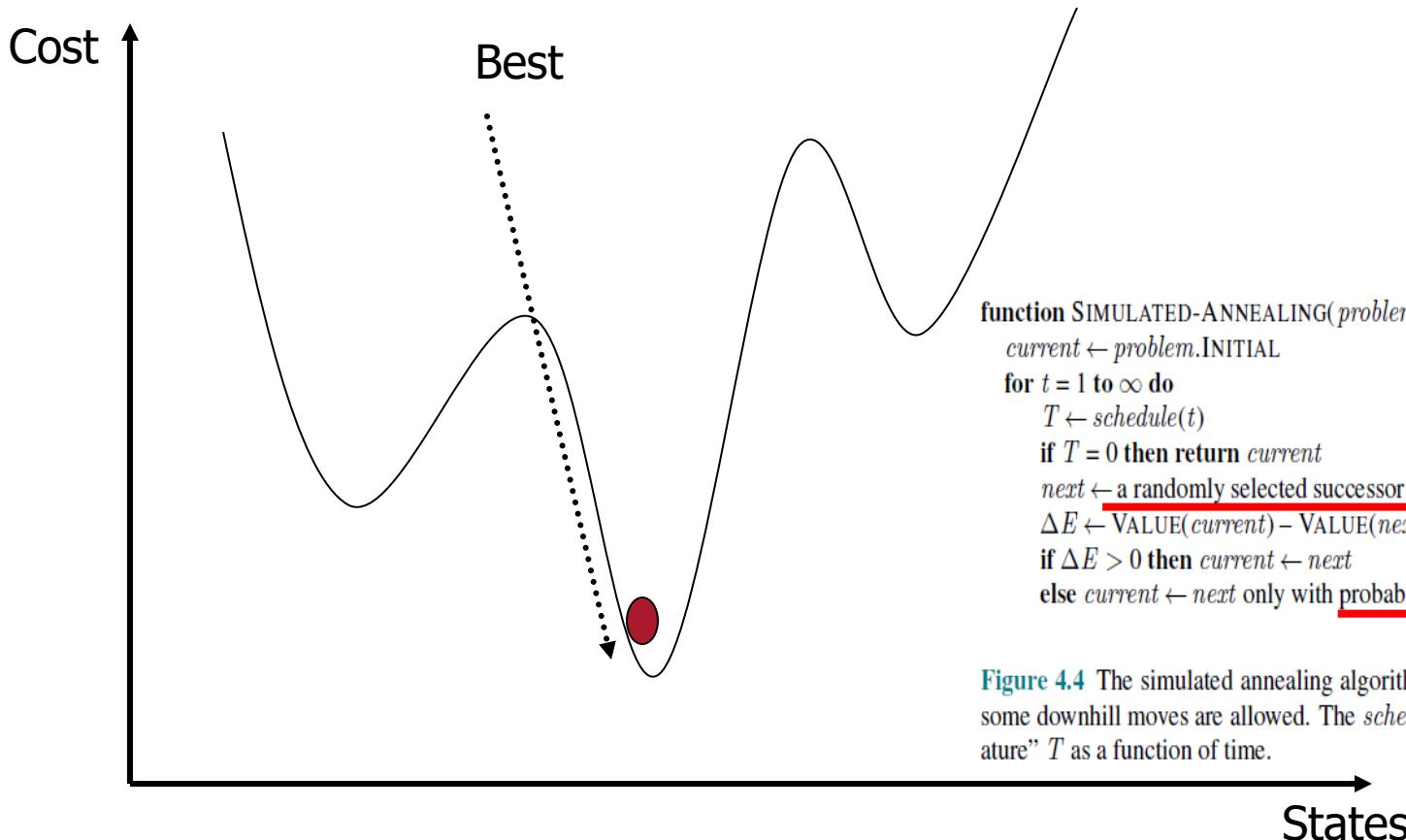
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” T as a function of time.

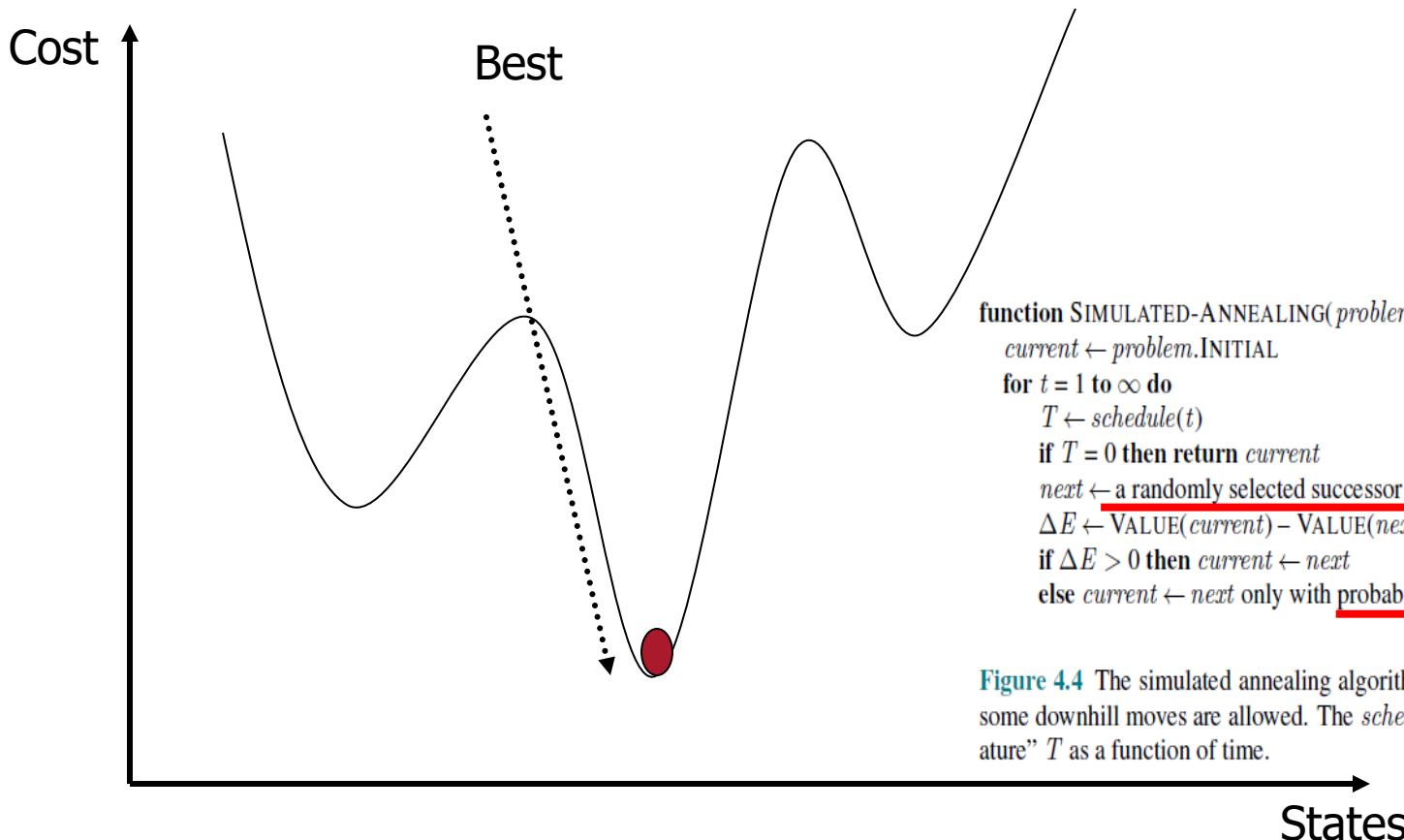
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

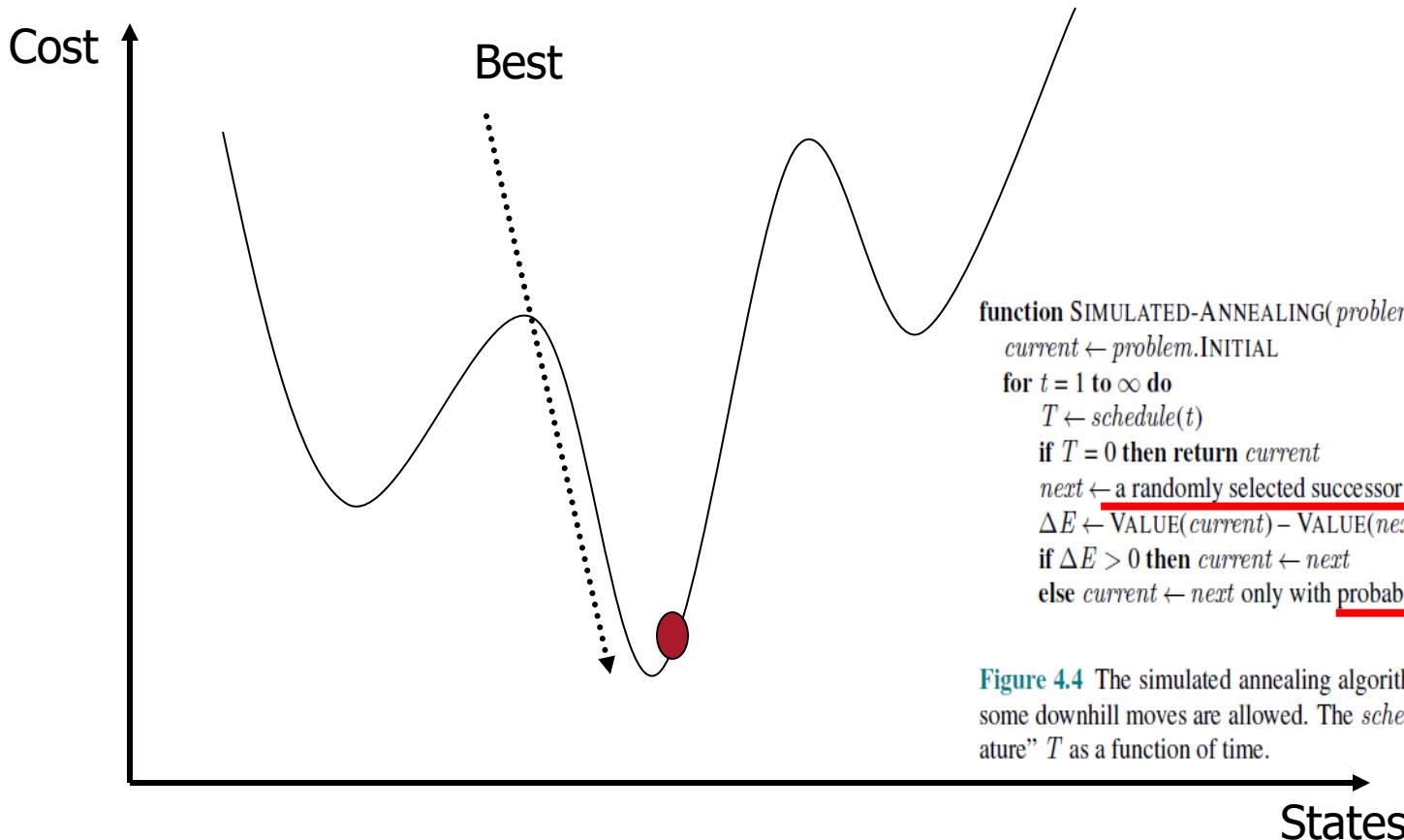
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

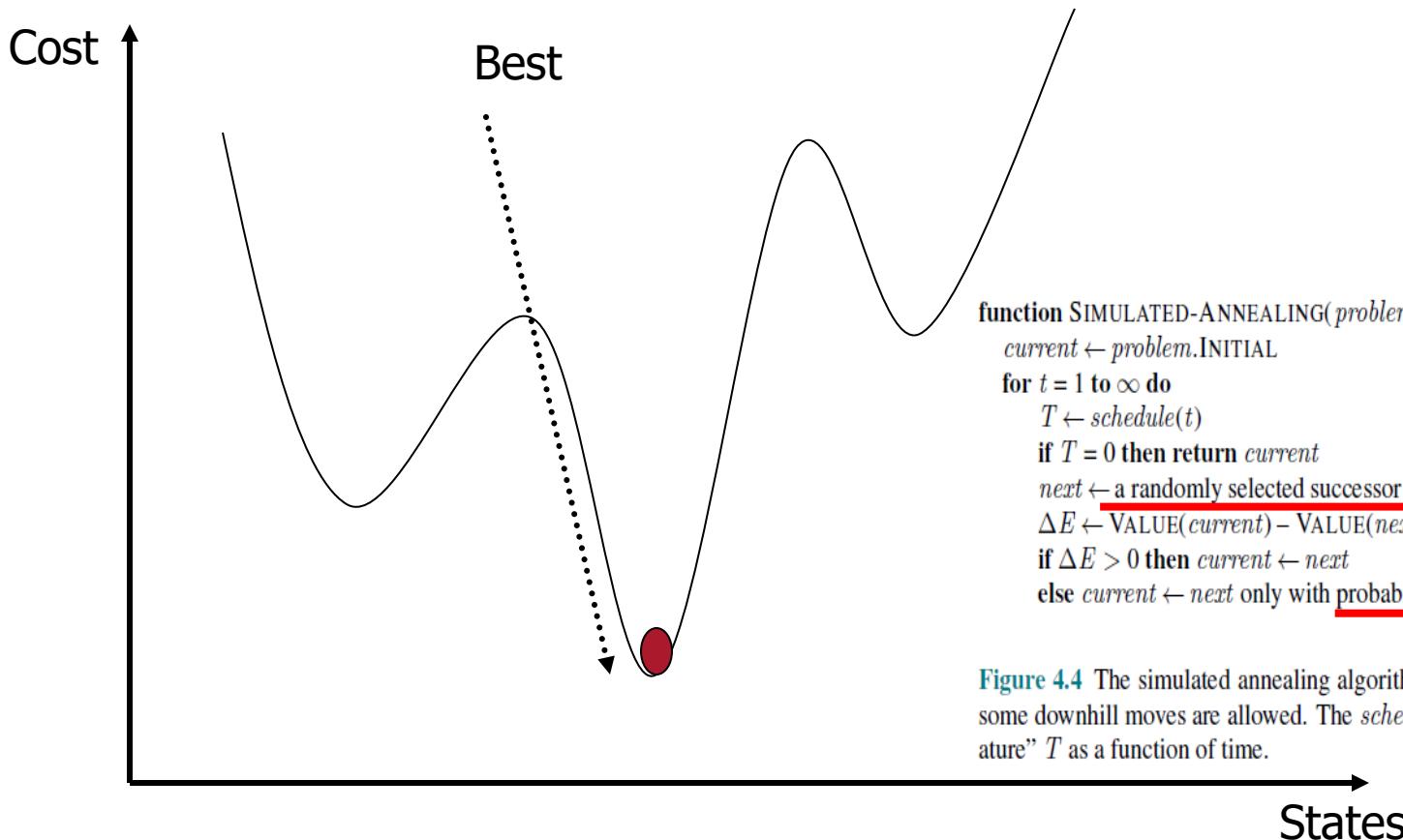
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

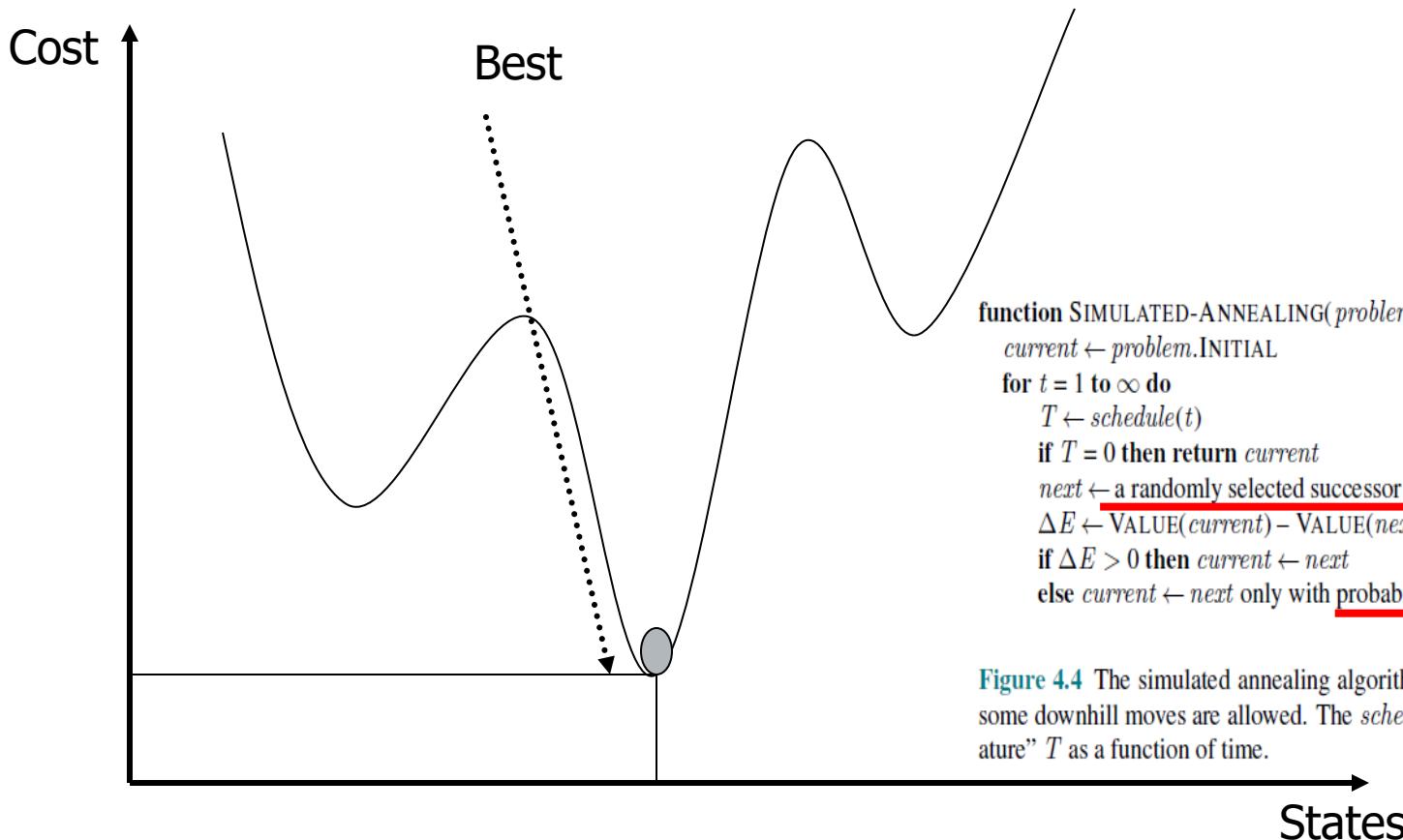
Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current  $\leftarrow$  problem.INITIAL
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  current  $\leftarrow$  problem.INITIAL
  for t = 1 to  $\infty$  do
    T  $\leftarrow$  schedule(t)
    if T = 0 then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

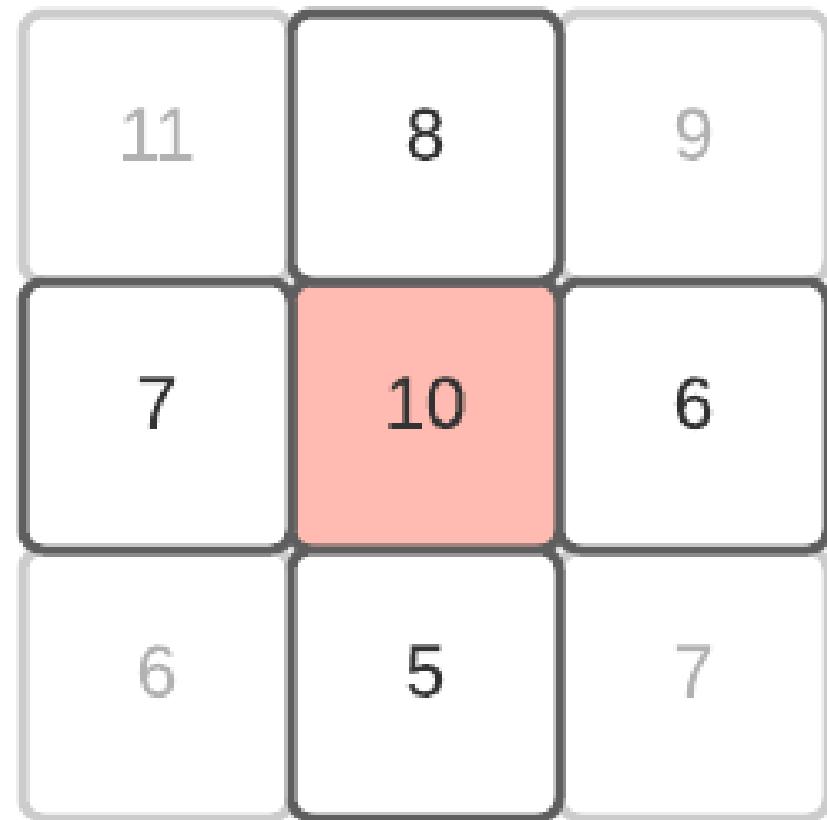
Figure 4.4 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

Simulated Annealing on the Peak Finding Problem

Given a 2D Array/Matrix, the task is to find the Peak element.

An element is a peak element if it is **greater than or equal to** its four neighbors, left, right, top and bottom.

- A Diagonal adjacent is **NOT** considered a neighbor, as we focus on **4 directions only**.
- A peak element is not necessarily the maximal element.
- More than one such element can exist.
- There is always a peak element.
- For **corner elements**, missing neighbors are considered of **negative infinite value**.



Simulated Annealing on the Peak Finding Problem

Given a 2D Array/Matrix, the task is to find the Peak element.

An element is a peak element if it is **greater than or equal to** its four neighbors, left, right, top and bottom.

- A Diagonal adjacent is **NOT** considered a neighbor, as we focus on 4 directions only.
- A peak element is not necessarily the maximal element.
- More than one such element can exist.
- There is always a peak element.
- For **corner elements**, missing neighbors are considered of **negative infinite value**.

Test if a condition returns True:

#if condition returns **True**, then nothing happens

#if condition returns **False**, AssertionError is raised.

```
# Pre-defined actions for PeakFindingProblem
directions4 = {'W': (0, -1), 'N': (-1, 0), 'E': (0, 1), 'S': (1, 0)}
directions8 = dict(directions4)
directions8.update({'NW': (-1, -1), 'NE': (-1, 1), 'SE': (1, 1), 'SW': (1, -1)})

class PeakFindingProblem(Problem):
    """Problem of finding the highest peak in a limited grid"""

    def __init__(self, initial, grid, defined_actions=directions4):
        """The grid is a 2 dimensional array/list whose state is specified by tuple of indices"""
        super().__init__(initial)
        self.grid = grid
        self.defined_actions = defined_actions
        self.n = len(grid)
        assert self.n > 0
        self.m = len(grid[0])
        assert self.m > 0

    def actions(self, state):
        """Returns the list of actions which are allowed to be taken from the given state"""
        allowed_actions = []
        for action in self.defined_actions:
            next_state = vector_add(state, self.defined_actions[action])
            if 0 <= next_state[0] <= self.n - 1 and 0 <= next_state[1] <= self.m - 1:
                allowed_actions.append(action)

        return allowed_actions

    def result(self, state, action):
        """Moves in the direction specified by action"""
        return vector_add(state, self.defined_actions[action])

    def value(self, state):
        """Value of a state is the value it is the index to"""
        x, y = state
        assert 0 <= x < self.n
        assert 0 <= y < self.m
        return self.grid[x][y]
```

utils.py

Simulated Annealing on the Peak Finding Problem

Given a 2D Array/Matrix, the task is to find the Peak element.

An element is a peak element if it is greater than or equal to its four neighbors, left, right, top and bottom.

- A Diagonal adjacent is not considered a neighbor.
- A peak element is not necessarily the maximal element.
- More than one such element can exist.
- There is always a peak element.
- For corner elements, missing neighbors are considered of **negative infinite value**.

```
from search import *

# directions4 = {'W': (0, -1), 'N': (-1, 0), 'E': (0, 1), 'S': (1, 0)}
# directions8 = dict(directions4)
# directions8.update({'NW': (-1, -1), 'NE': (-1, 1), 'SE': (1, 1), 'SW': (1, -1)})

def main():
    initial = (0, 0)
    grid = [[10, 20, 15], [21, 30, 14], [7, 16, 32]]

    problem = PeakFindingProblem(initial, grid, directions4)

    solutions = [problem.value(simulated_annealing(problem)) for i in range(3)] #Change to 100
    print(solutions)

    solutions = set(solutions)
    print(solutions)

    print(max(solutions))

if __name__ == "__main__":
    main()
```

Local Beam Search

- Basic idea:
 - K states instead of one of a local search algorithm, initialized **k randomly generated states.**
Or, K chosen randomly with a bias towards good ones
 - For each iteration
 - Generate ALL successors from K current states
 - If any one reaches a goal state, the algorithm stops.
 - Or else, the algorithm select the **best K successors** from the above complete list to be the new current K states.
 - This process repeats until it finds a successor reaches a goal, the algorithm stops.

```
function BEAM-SEARCH(problem, k) returns a solution state
```

 start with k randomly generated states

 loop

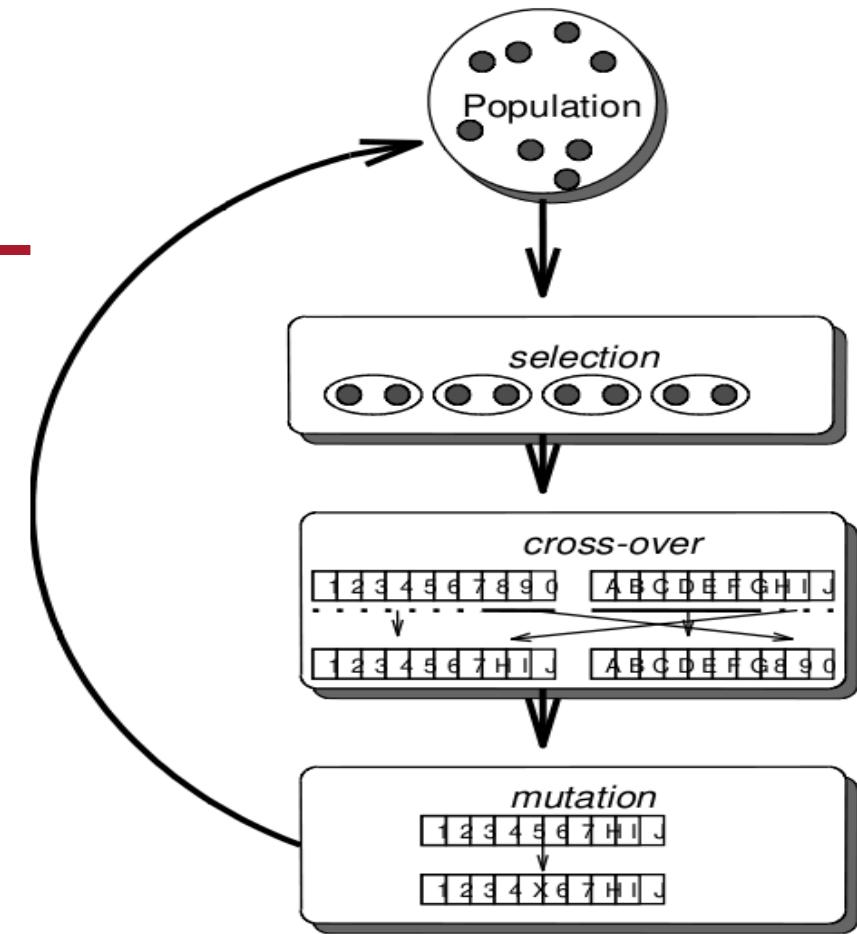
 generate all successors of all k states

 if any of them is a solution then return it

 else select the k best successors

Genetic Algorithms

- A population of individual solutions (states), in which the fittest (highest value) individuals **produce offspring** (successor states) that populate the next generation, a process called **recombination (mating/crossover)**.
- **Each state or individual** is represented as a string over a finite alphabet. It is also called **chromosome which contains genes**.
- Genetic Algorithms are one of these algorithms.
 - Start with ***k* randomly generated solutions/chromosomes (states) (population)**
 - An individual solution (state) is a sequence of **real numbers** or **a computer program** or **a CNN architecture**,
 - Evaluation function (**fitness function**). Higher values for better solution (state).
 - Produce the next generation of solutions (states) by **selection, crossover, and mutation**
- The **genetic_algorithm()** function in search.py.

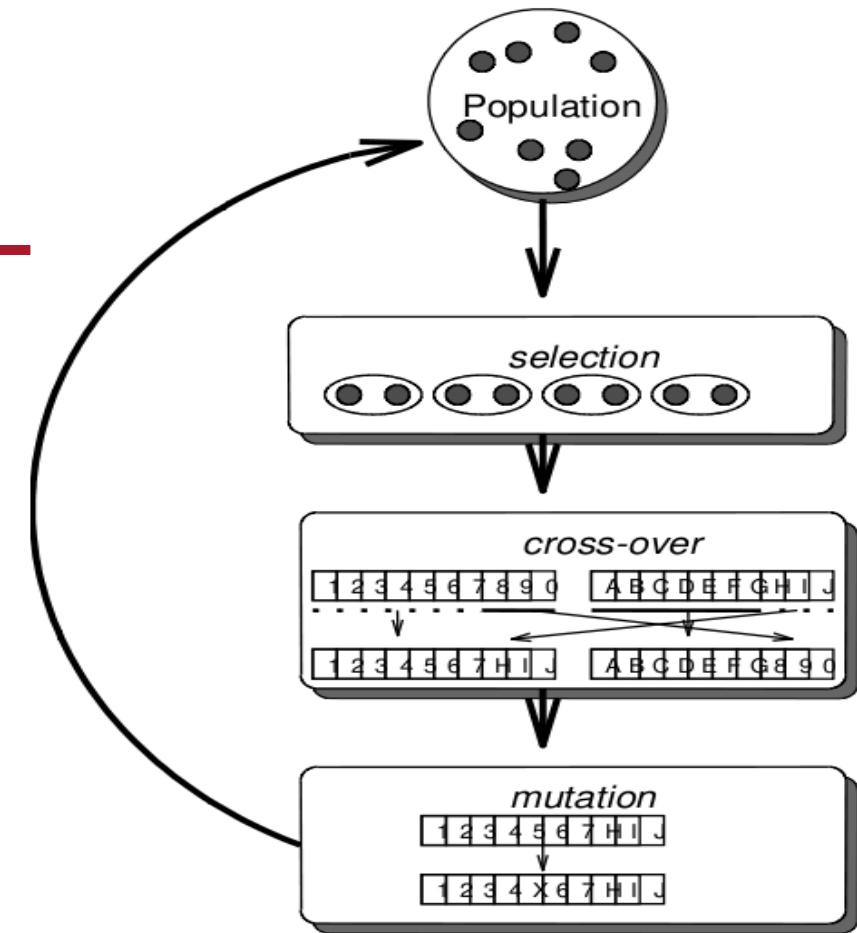


Algorithm 9.5 Genetic Algorithm(Input: Initial Population, fitness function, percent for mutation, selection threshold)

- 1: Initialize the population with random candidate solutions
- 2: Apply fitness function to Evaluate each candidate's fitness value
- 3: repeat
- 4: Select parents based on fitness value
- 5: Recombine pairs of parents (crossover)
- 6: Mutate resulting offspring
- 7: Apply fitness function to Evaluate new candidates' fitness value
- 8: until termination condition/goal is reached

termination_condition
= the max number of generations reached

Genetic Algorithms

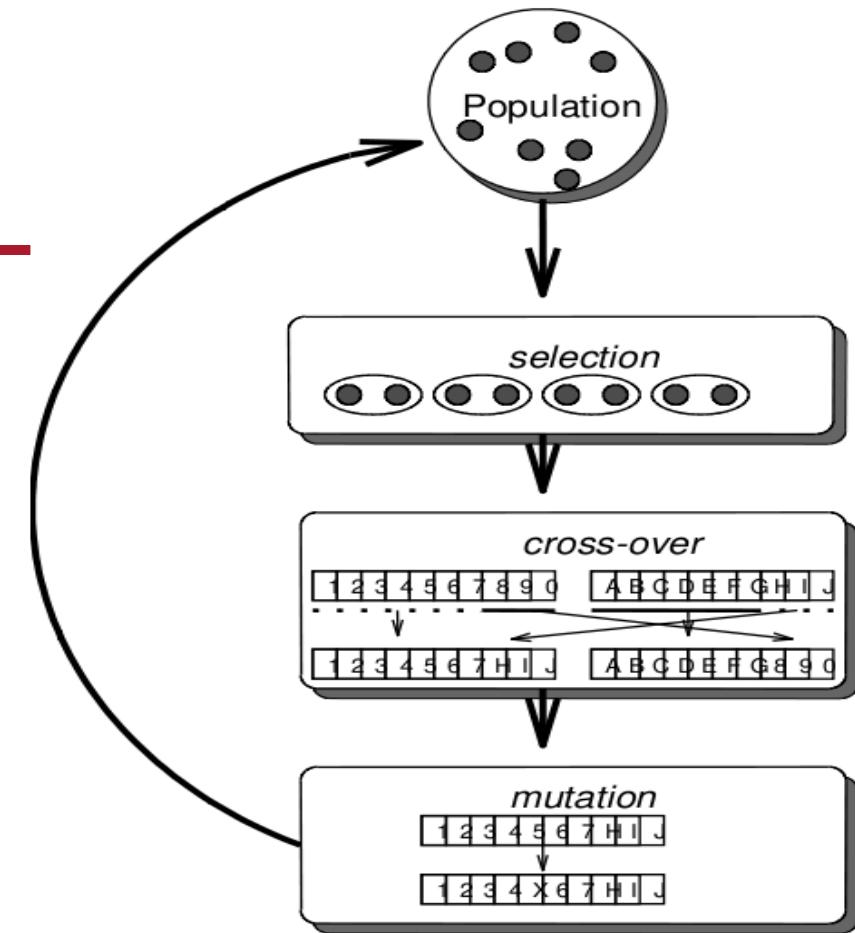
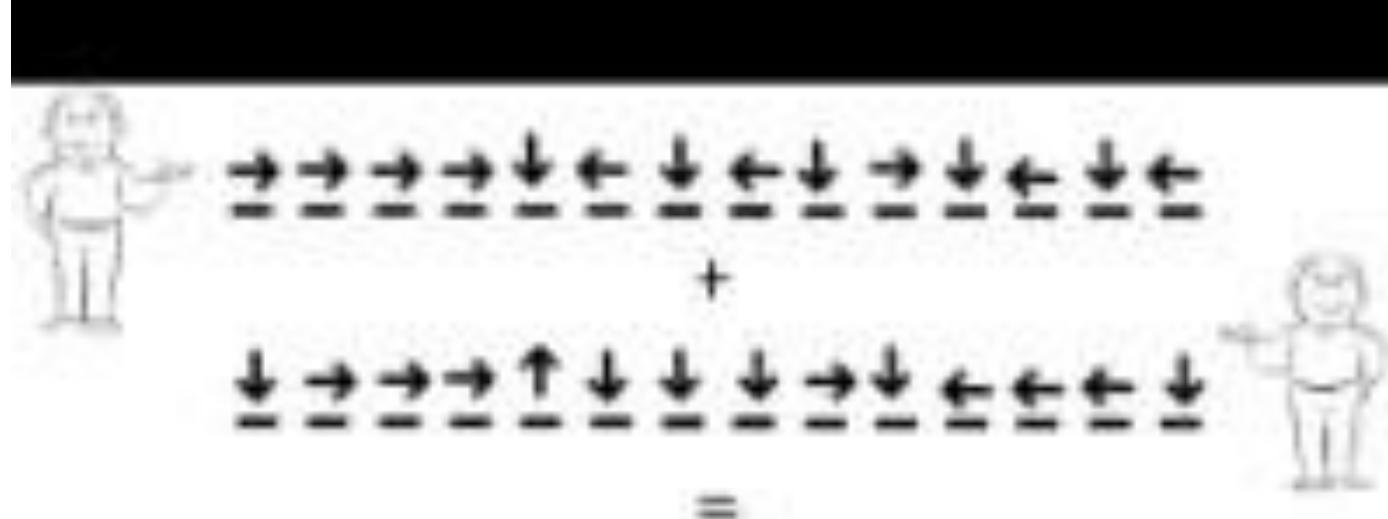


Algorithm 9.5 Genetic Algorithm(**Input:** Initial Population, fitness function, percent for mutation, selection threshold)

- 1: Initialize the population with random candidate solutions
- 2: Apply fitness function to Evaluate each candidate's fitness value
- 3: repeat
- 4: Select parents based on fitness value
- 5: Recombine pairs of parents (crossover)
- 6: Mutate resulting offspring
- 7: Apply fitness function to Evaluate new candidates' fitness value
- 8: until termination condition/goal is reached

termination_condition
= the max number of generations reached

Genetic Algorithms



Algorithm 9.5 Genetic Algorithm (Input: Initial Population, fitness function, percent for mutation, selection threshold)

- 1: Initialize the population with random candidate solutions
 - 2: Apply fitness function to Evaluate each candidate's fitness value
 - 3: repeat
 - 4: Select parents based on fitness value
 - 5: Recombine pairs of parents (crossover)
 - 6: Mutate resulting offspring
 - 7: Apply fitness function to Evaluate new candidates' fitness value
 - 8: until termination condition/goal is reached

**terminat
= the ma
generati**

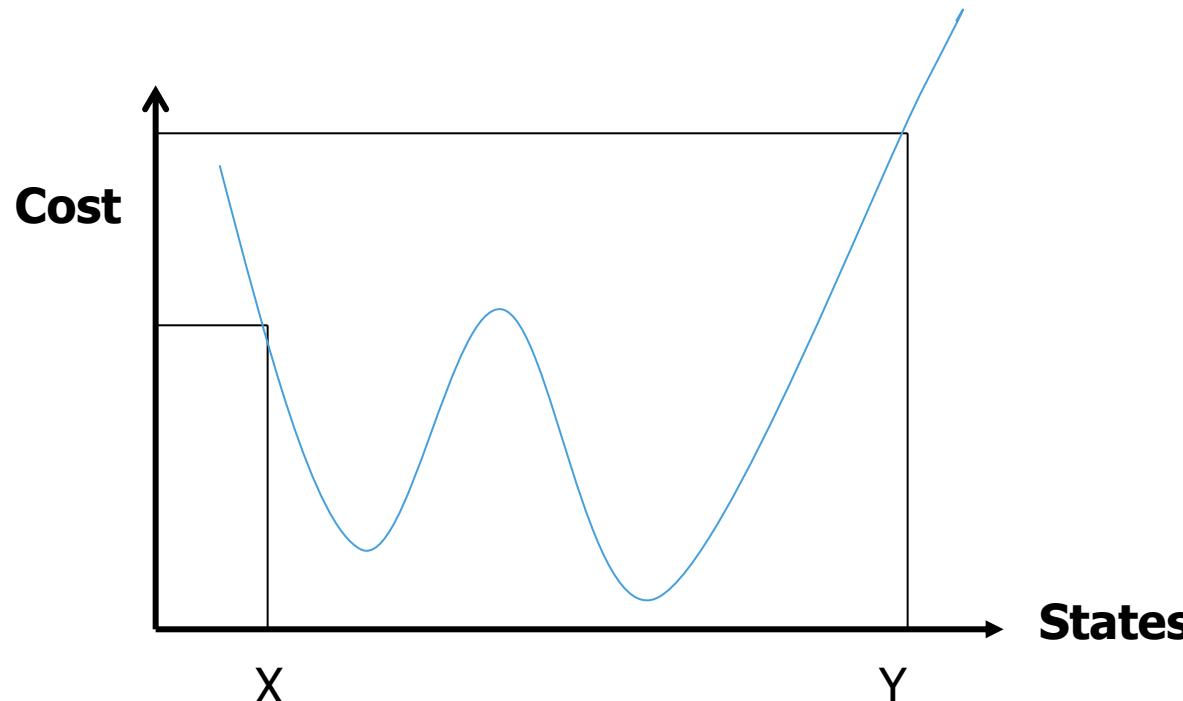
termination_condition
= the max number of
generations reached

Genetic Algorithms

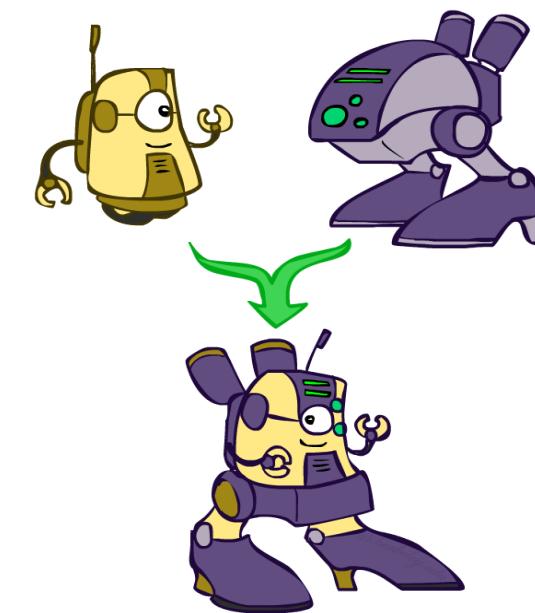
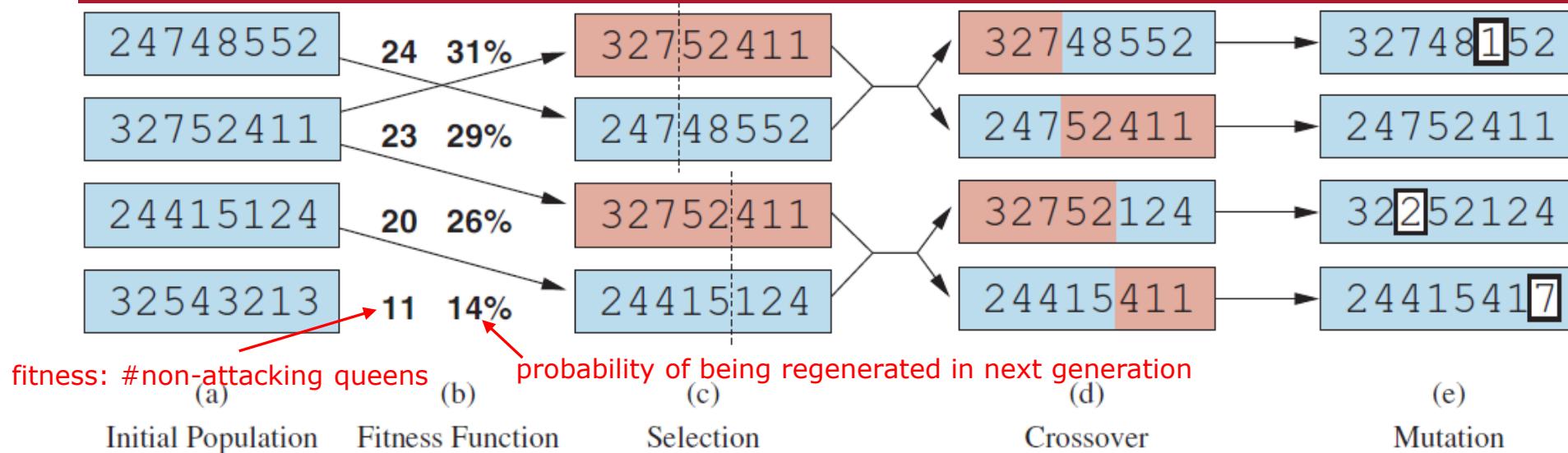
- Each state is rated by the evaluation function called fitness function. Fitness function should return higher values for better states:

Fitness(X) should be greater than Fitness(Y) !!

[Fitness(x) = 1/Cost(x)]

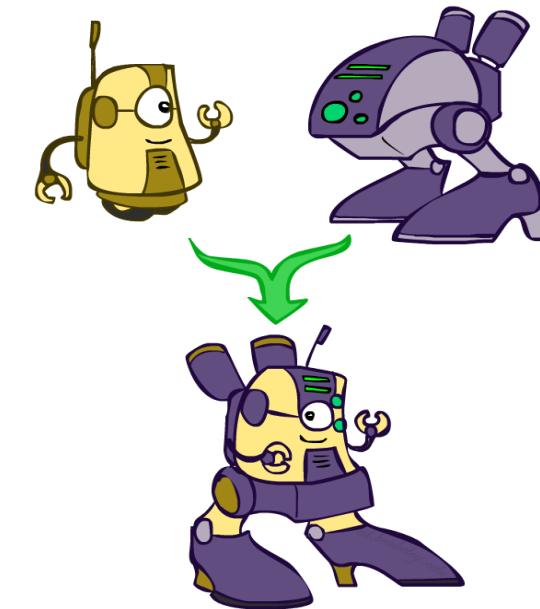
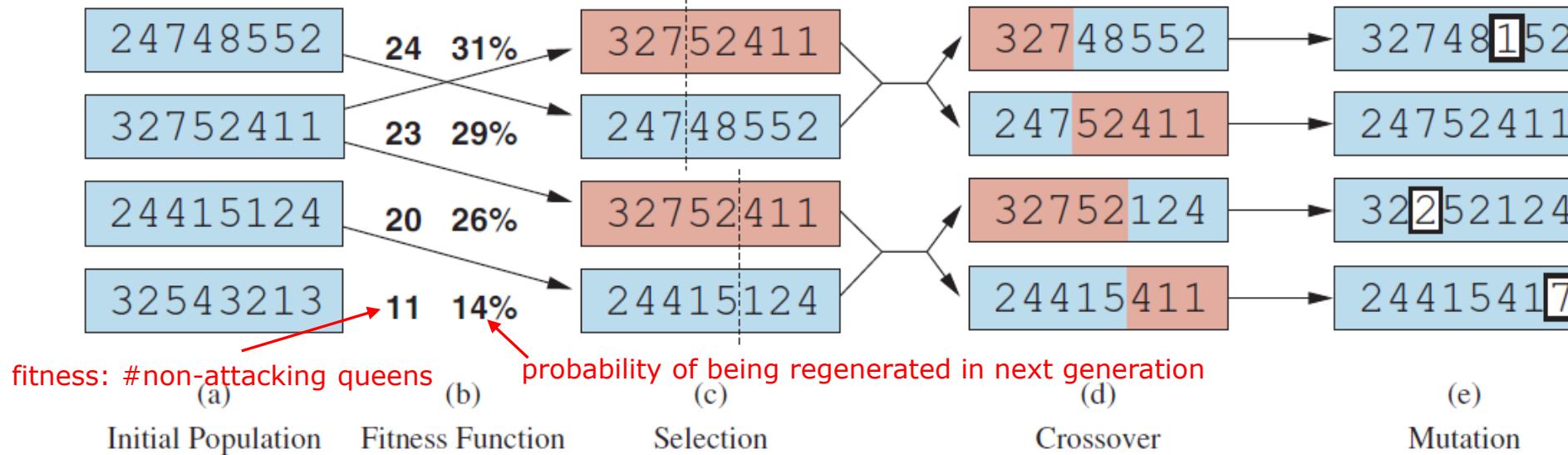


Genetic Algorithms on 8-Queens Problem



- **Initial Population:** Start with K randomly individuals
- **Fitness Function:** Find the number of non-attacking pairs of queens for each individual. Higher fitness values are better. Min = 0; Max = 28. **Why?**
- A possible fitness function is the number of non-attacking pairs of queens that we are interested to maximize which has the maximum value $C_2^8 = \frac{8 \cdot 7}{2} = 28$.
- We have 8 ways to choose the first queen in the pair, and 7 ways to choose the second queen, different from the first. But then we divide by 2, because each pair of queens $\{X, Y\}$ was counted twice: taking X as the first queen and Y as the second, and taking Y as the first queen and X as the second.

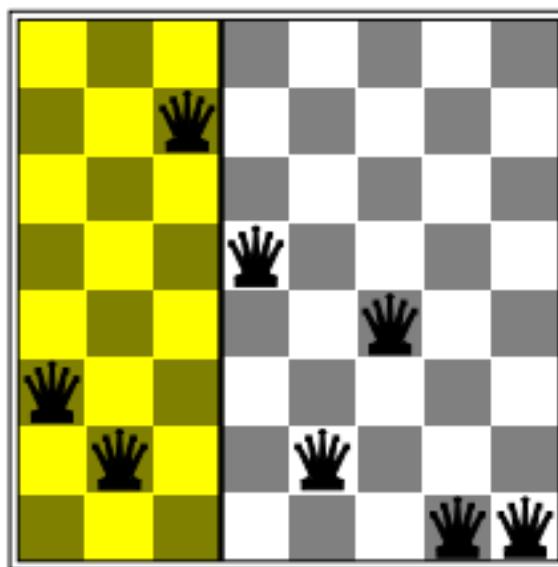
Genetic Algorithms on 8-Queens Problem



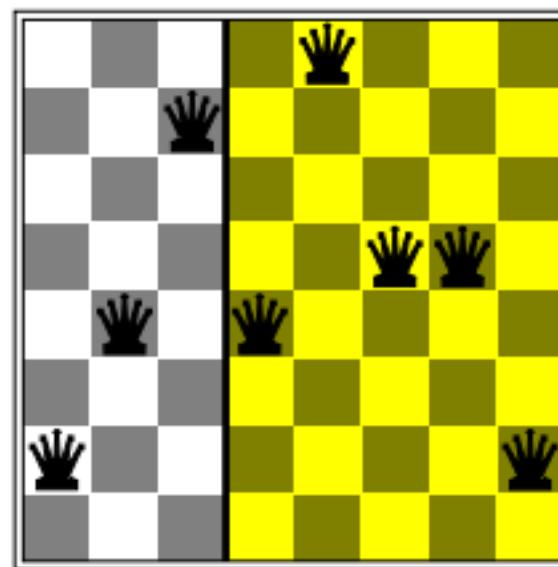
- **Initial Population:** Start with K randomly individuals
- **Fitness Function:** Find the number of non-attacking pairs of queens for each individual. Higher fitness values are better. Min = 0; Max = 28
- **Selection:** Compute the probability of each individual, e.g., $24 / (24 + 23 + 20 + 11) = 31\%$
 - Select from p individuals with higher probabilities **OR**
 - Randomly select n individuals and then select the p most fit ones as parents with higher probabilities
- **Crossover:** Split each of the parent individuals and recombine the parts to form two children.
- **Mutation:** Determine the mutation rate that every bit in its composition is flipped with probability equal to the mutation rate.

Genetic Algorithms on 8-Queens Problem

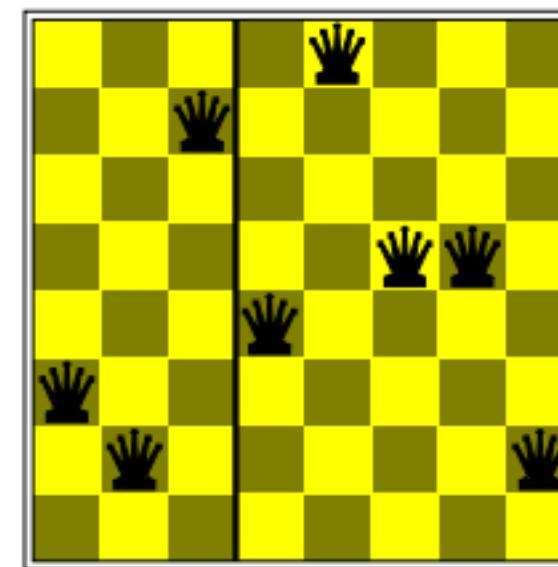
i'th character = row where i'th queen is located



+



=



672 47588

752 51447

672 51447

GAs are suitable to generate a variety of good solutions, but not in finding the optimal solution

Genetic Algorithms on Learning How to Jump Over the Ball



Class Exercise 4

Review the **EightQueen** by **GA.py** in <https://canvas.wpi.edu/courses/63988/files/6728933?wrap=1> and the below functions in <https://github.com/aimacode/aima-python/blob/master/search.py>, do the following with your teammates/classmates to figure out how to determine an individual in the next generation population?

- `init_population(pop_number, gene_pool, state_length)`
- `genetic_algorithm(population, fitness_fn, gene_pool=[0, 1], f_thres=None, ngen=1000, pmut=0.1)`
- `select(r, population, fitness_fn)`
- `recombine(x, y)`
- `mutate(x, gene_pool, pmut)`
- `fitness_threshold(fitness_fn, f_thres, population)`

8-Queens Problem

`map(fun, iter)` function returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.)

```
def init_population(pop_number, gene_pool, state_length):
    """Initializes population for genetic algorithm init_population(100, range(8), 8)
pop_number : Number of individuals in population
gene_pool   : List of possible values for individuals
state_length: The length of each individual"""\n    [0, 0, 5, 1, 7, 2, 2, 5]
g = len(gene_pool)
population = []
for i in range(pop_number):
    new_individual = [gene_pool[random.randrange(0, g)] for j in range(state_length)]
    population.append(new_individual)
return population

def genetic_algorithm(population, fitness_fn, gene_pool=[0, 1], f_thres=None, ngen=1000, pmut=0.1):
    """[Figure 4.8]"""\n    genetic_algorithm(population, fitness, gene_pool=range(8), f_thres=2)
for i in range(ngen):
    population = [mutate(recombine(*select(2, population, fitness_fn)), gene_pool, pmut)
                  for i in range(len(population))]

    fittest_individual = fitness_threshold(fitness_fn, f_thres, population)
    if fittest_individual:
        return fittest_individual
return max(population, key=fitness_fn)

def fitness_threshold(fitness_fn, f_thres, population):
    if not f_thres:
        return None
```

number of arguments to a function

number of arguments to a function

```
def fitness_threshold(fitness_fn, f_thres, population):
    if not f_thres:
        return None

    fittest_individual = max(population, key=fitness_fn)
    if fitness_fn(fittest_individual) >= f_thres:
        return fittest_individual

    return None
```

```
def select(r, population, fitness_fn):  
    fitnesses = map(fitness_fn, population)  
    sampler = weighted_sampler(population, fitnesses)  
    return [sampler() for i in range(r)]
```

```
def recombine(x, y):
```

n = len(x)

```
c = random.randrange(0, n)
```

```
return x[:c] + y[c:]
```

pmut = percent for non-mutation

```
def mutate(x, gene_pool, pmut):
```

```
if random.un  
    return x
```

```
n = len(x)
```

```
g = len(gene_pool)
```

```
c = random.randrange(0, n)
```

```
r = random.randrange(0, g)
```

```
new_gene = gene_pool[r]
```

```
return x[ :c] + [new gene] + x[c + 1:]
```

The `random.randrange(start, stop, step)` method returns a randomly selected element from the specified range. `stop` is not included.

Worcester Polytechnic Institute

8-Queens Problem

```
from search import *

def fitness(q):
    non_attacking = 0
    for row1 in range(len(q)):
        for row2 in range(row1+1, len(q)):
            col1 = int(q[row1])
            col2 = int(q[row2])
            row_diff = row1 - row2
            col_diff = col1 - col2

            if col1 != col2 and row_diff != col_diff and row_diff != -col_diff:
                non_attacking += 1

    return non_attacking

def main():
    """Initializes population for genetic algorithm
    pop_number : Number of individuals in population
    gene_pool   : List of possible values for individuals
    state_length: The length of each individual
    """
    population = init_population(100, range(8), 8)
    print(population[:5]) #Display the first 5 individuals. Each individual is a solution state for the 8-Queens Problem

    solution = genetic_algorithm(population, fitness, gene_pool=range(8), f_thres=25)
    print(solution)

    print(fitness(solution))

if __name__ == "__main__":
    main()
```

Genetic Algorithms – Library and Resources

PyGAD - Python Genetic Algorithm: <https://pygad.readthedocs.io/en/latest/>

<https://www.youtube.com/watch?v=uQj5UNhCPuo>

<https://www.youtube.com/watch?v=nhT56blfRpE>



Summary

- Many configuration and optimization problems can be formulated as local search
- General families of algorithms:
 - Hill-climbing, continuous optimization
 - Simulated annealing (and other stochastic methods)
 - Local beam search: multiple interaction searches
 - Genetic algorithms: break and recombine states

Many machine learning algorithms are local searches