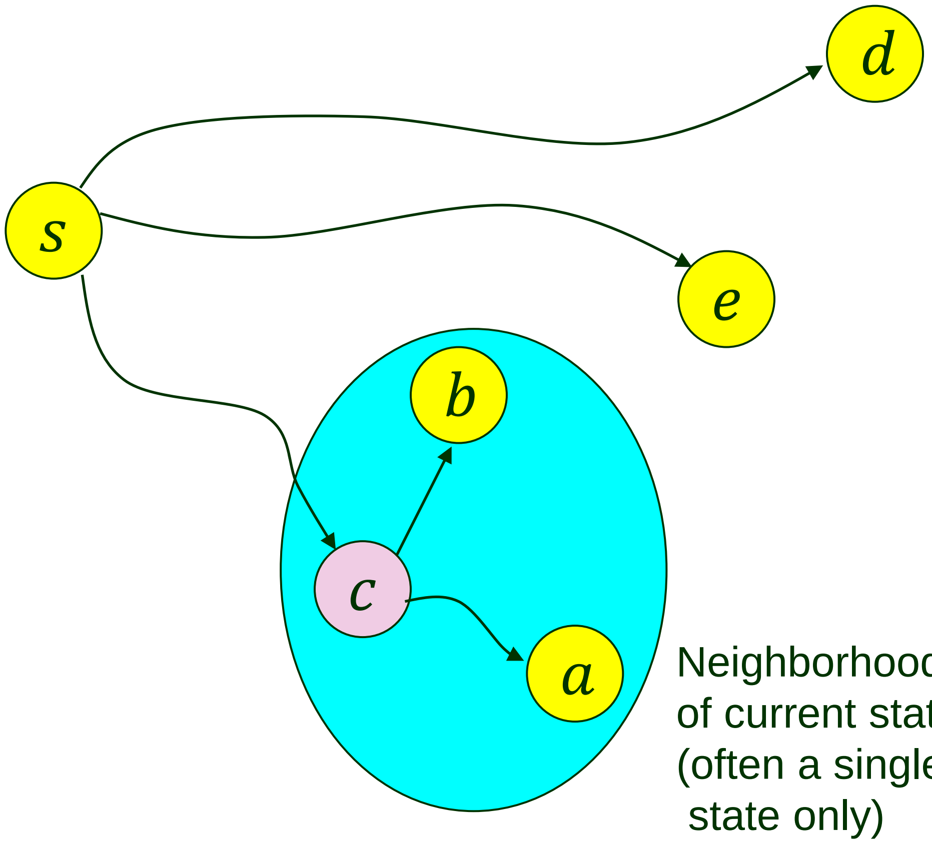
# Local Search

Evaluate and modify one or more *current states* rather than systematically exploring paths from an initial state.




Neighborhood
of current states
(often a single
state only)

* Figures are from the textbook site (or drawn by the instructor).

# Local Search

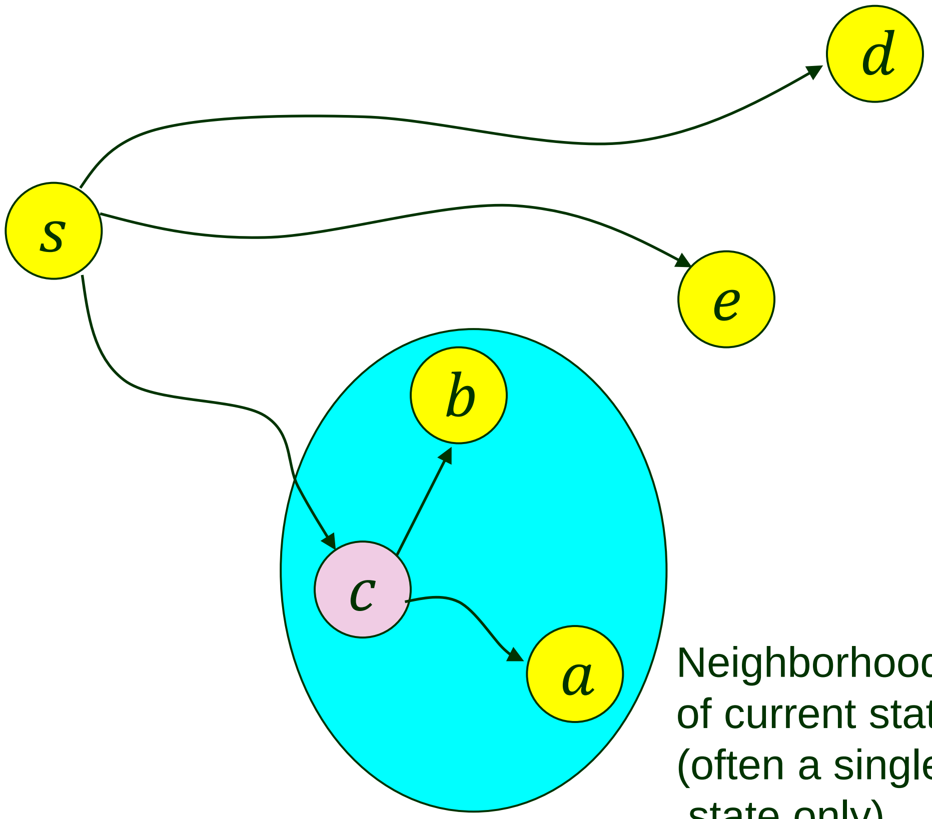
Evaluate and modify one or more *current states* rather than systematically exploring paths from an initial state.

## Outline

I. Hill climbing

II. Simulated annealing

III. Genetic algorithms




Neighborhood of current states (often a single state only)

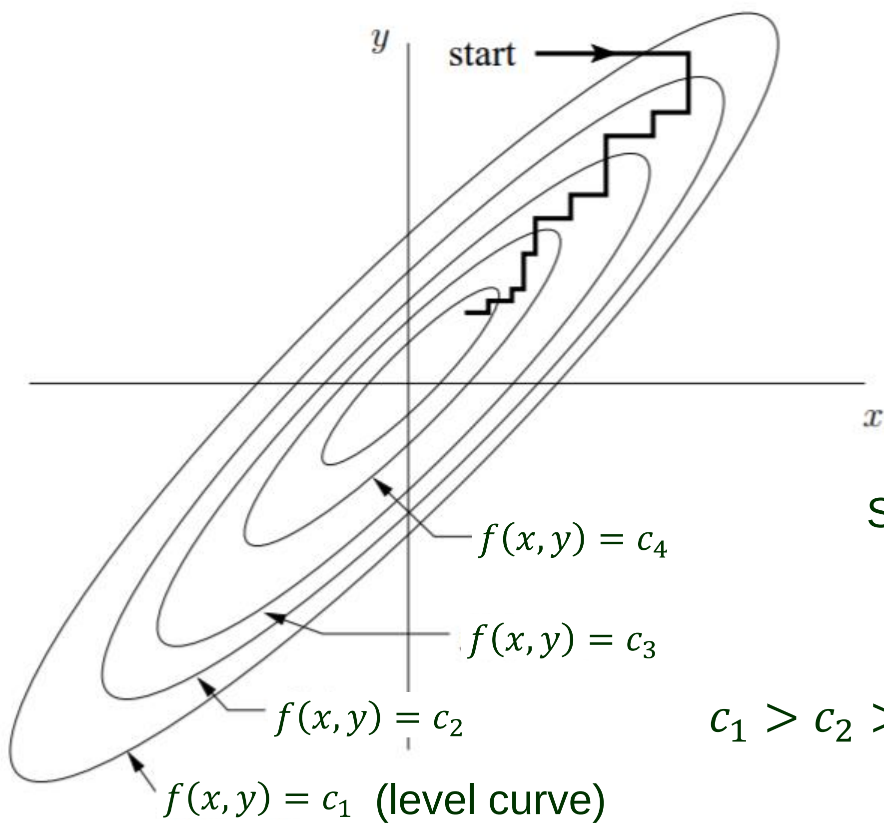
* Figures are from the textbook site (or drawn by the instructor).

# Advantages of Local Search

- Use of very little memory.
- Finding good solutions in state spaces *intractable* for a systematic search.
- Useful in pure optimization (e.g., gradient-based descent methods)

# Advantages of Local Search

- Use of very little memory.
- Finding good solutions in state spaces *intractable* for a systematic search.
- Useful in pure optimization (e.g., gradient-based descent methods)

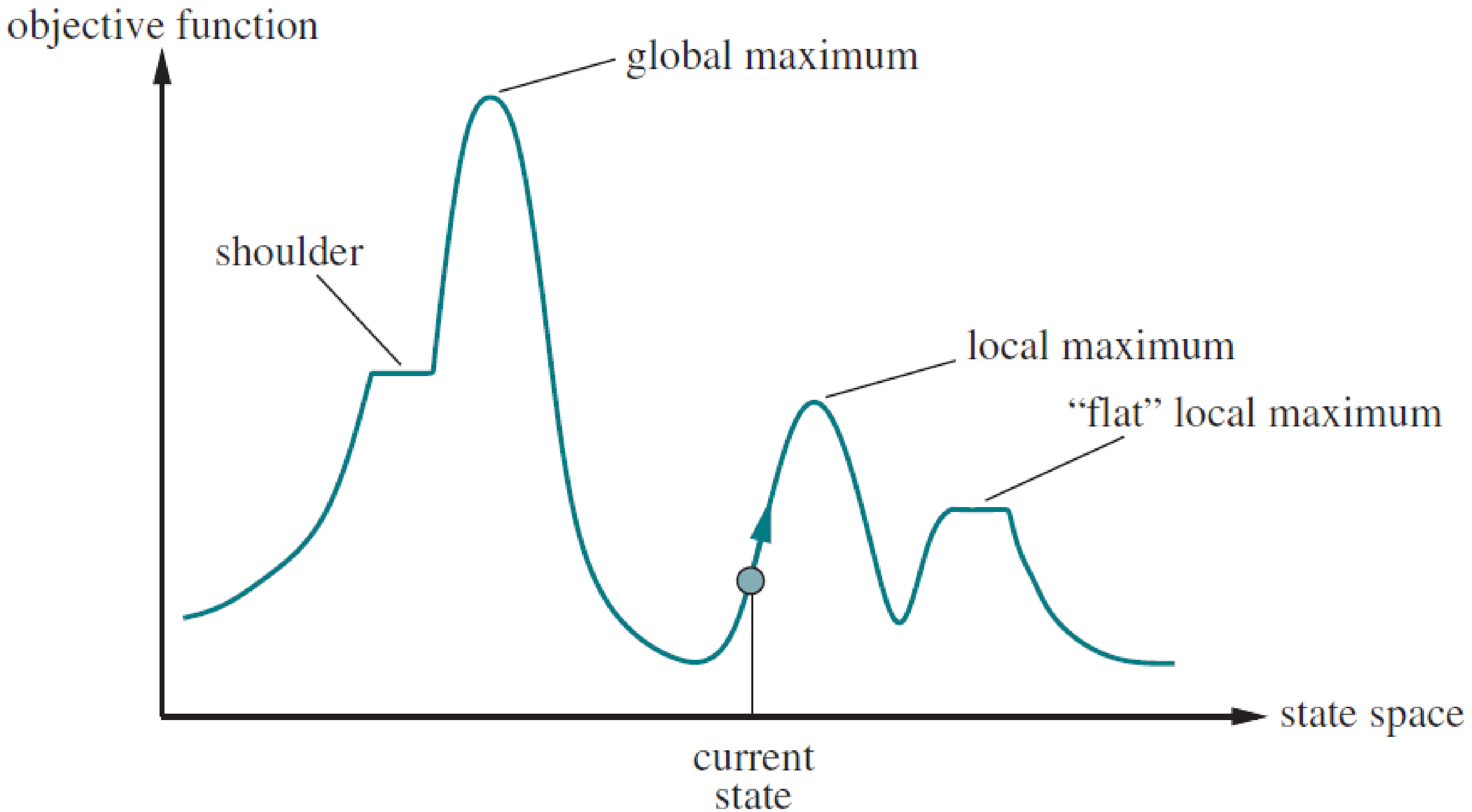



Maximize $f(x, y)$

Steepest ascent
(along $\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$)

$c_1 > c_2 > c_3 > c_4$

# State Space Landscape



# I. Hill Climbing

**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** $current$
    $current \leftarrow neighbor$

# I. Hill Climbing

**function** HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  *current* ← *problem*.INITIAL
  **while** *true* **do**
    *neighbor* ← a highest-valued successor state of *current*  // random choice // to break a tile
    **if** VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
    *current* ← *neighbor*

# I. Hill Climbing

**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 $current$ // random choice // to break a tile
**if** VALUE($neighbor$) $\leq$ VALUE($current$) **then return** $current$
$current \leftarrow neighbor$

8-queens problem

# I. Hill Climbing

**function** HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  *current* ← *problem*.INITIAL
  **while** *true* **do**
    *neighbor* ← a highest-valued successor state of *current*   // random choice // to break a tile
    **if** VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
    *current* ← *neighbor*


## 8-queens problem

- $h(s)$ = # pairs of queens attacking each other, directly or indirectly, in the state $s$.

# I. Hill Climbing

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
   current ← problem.INITIAL
   while true do
      neighbor ← a highest-valued successor state of current   // random choice
                                                               // to break a tile
      if VALUE(neighbor) ≤ VALUE(current) then return current
      current ← neighbor
```


$h = 17$

## 8-queens problem

- $h(s)$ = # pairs of queens attacking each other, directly or indirectly, in the state $s$.

# I. Hill Climbing

**function** HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  *current* ← *problem*.INITIAL
  **while** *true* **do**
    *neighbor* ← a highest-valued successor state of *current*   // random choice // to break a tile
    **if** VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
    *current* ← *neighbor*

## 8-queens problem


$h = 17$

- $h(s) = \#$ pairs of queens attacking each other, directly or indirectly, in the state $s$.
- Successor is a state generated from relocating a queen in the same column.

# I. Hill Climbing

**function** HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  *current* ← *problem*.INITIAL
  **while** *true* **do**
    *neighbor* ← a highest-valued successor state of *current*  // random choice // to break a tile
    **if** VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
    *current* ← *neighbor*


$h = 17$

## 8-queens problem

- $h(s)$ = # pairs of queens attacking each other, directly or indirectly, in the state $s$.
- Successor is a state generated from relocating a queen in the same column.

# I. Hill Climbing

**function** HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
    *current* ← *problem*.INITIAL
    **while** *true* **do**
        *neighbor* ← a highest-valued successor state of *current*   // random choice // to break a tile
        **if** VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
        *current* ← *neighbor*


$h = 17$

## 8-queens problem

- $h(s)$ = # pairs of queens attacking each other, directly or indirectly, in the state $s$.
- Successor is a state generated from relocating a queen in the same column.

# I. Hill Climbing

**function** HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  *current* ← *problem*.INITIAL
  **while** *true* **do**
    *neighbor* ← a highest-valued successor state of *current*   // random choice // to break a tile
    **if** VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
    *current* ← *neighbor*

## 8-queens problem


$h = 17$

- $h(s)$ = # pairs of queens attacking each other, directly or indirectly, in the state $s$.
- Successor is a state generated from relocating a queen in the same column.
- $h(s) = 12$ for the best successor.

# I. Hill Climbing

**function** HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  *current* ← *problem*.INITIAL
  **while** *true* **do**
    *neighbor* ← a highest-valued successor state of *current*  // random choice // to break a tile
    **if** VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
    *current* ← *neighbor*


$h = 17$

## 8-queens problem

- $h(s) =$ # pairs of queens attacking each other, directly or indirectly, in the state $s$.
- Successor is a state generated from relocating a queen in the same column.
- $h(s) = 12$ for the best successor.

# I. Hill Climbing

**function** HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  *current* ← *problem*.INITIAL
  **while** *true* **do**
    *neighbor* ← a highest-valued successor state of *current*   // random choice
                                         // to break a tile
    **if** VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
    *current* ← *neighbor*

## 8-queens problem


$h = 17$

- $h(s)$ = # pairs of queens attacking each other, directly or indirectly, in the state $s$.
- Successor is a state generated from relocating a queen in the same column.
- $h(s) = 12$ for the best successor.

8-way tie!.

# I. Hill Climbing

**function** HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  *current* ← *problem*.INITIAL
  **while** *true* **do**
    *neighbor* ← a highest-valued successor state of *current* // random choice // to break a tile
    **if** VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
    *current* ← *neighbor*

## 8-queens problem


$h = 17$

- $h(s)$ = # pairs of queens attacking each other, directly or indirectly, in the state $s$.
- Successor is a state generated from relocating a queen in the same column.
- $h(s)$ = 12 for the best successor.

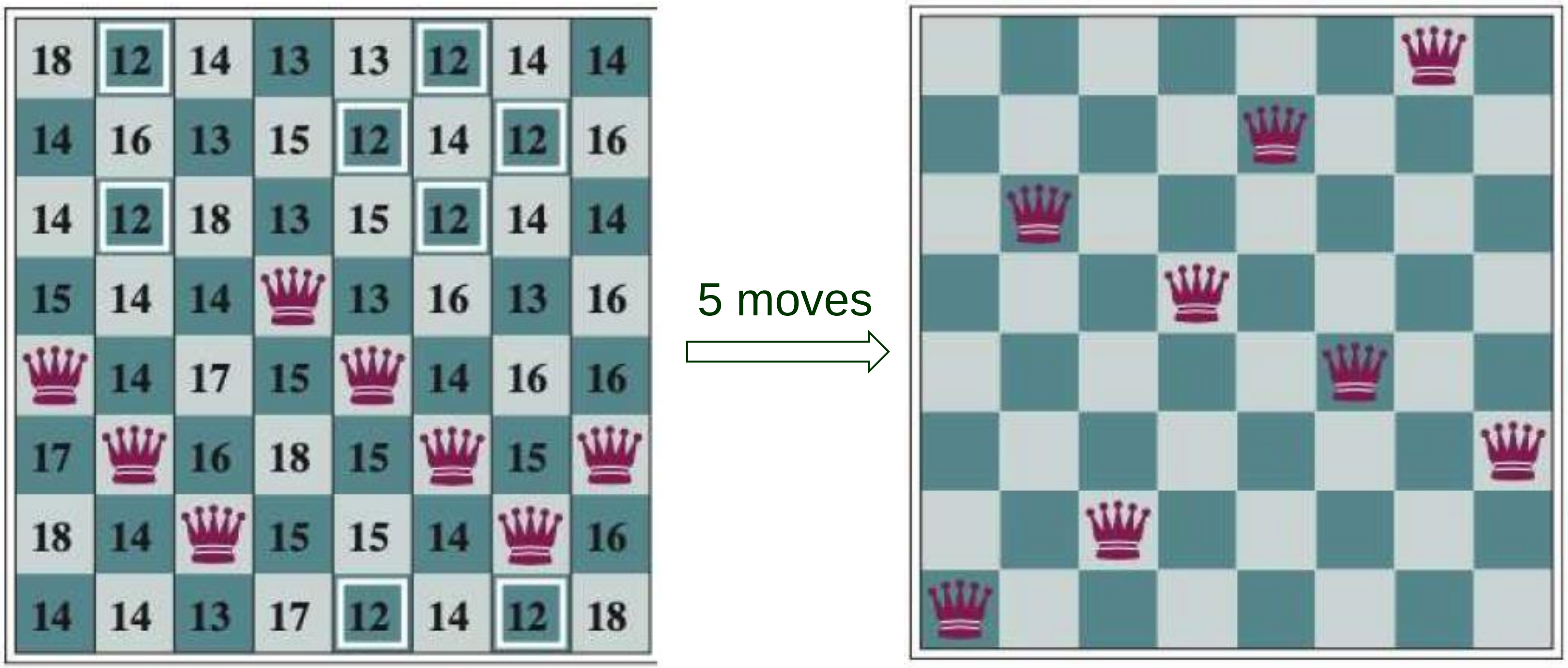
8-way tie!.

Hill climbing randomly picks one.

# Efficiency?

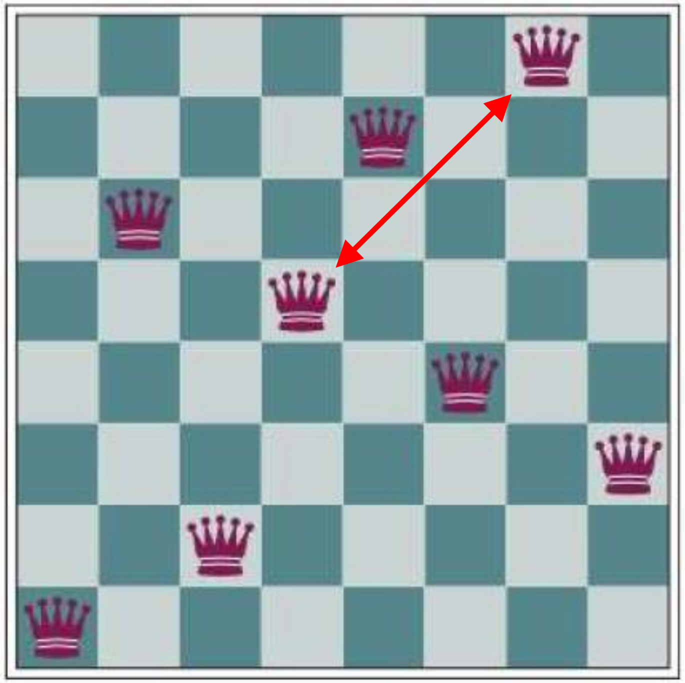

18 12 14 13 13 12 14 14
14 16 13 15 12 14 12 16
14 12 18 13 15 12 14 14
15 14 14 13 16 13 16
14 17 15 14 16 16
17 16 18 15 15
18 14 15 15 14 16
14 14 13 17 12 14 12 18

# Efficiency?


18 12 14 13 13 12 14 14
14 16 13 15 12 14 12 16
14 12 18 13 15 12 14 14
15 14 14 13 16 13 16
14 17 15 14 16 16
17 16 18 15 15
18 14 15 15 14 16
14 14 13 17 12 14 12 18
5 moves

# Efficiency?

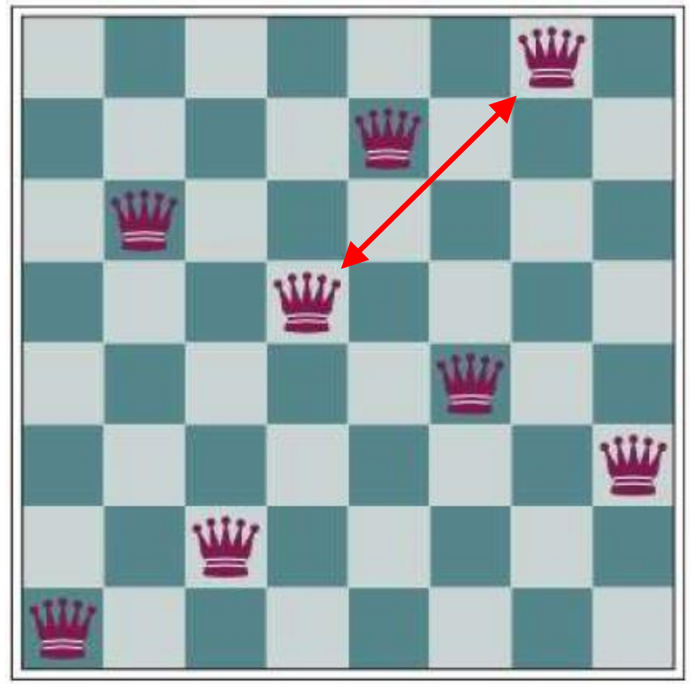

5 moves



$h = 1$

# Efficiency?


5 moves



$h = 1$

- ♦ Hill climbing can make rapid progress toward a solution.

# Drawback of Hill Climbing (1)

Hill climbing terminates when a peak is reached with no neighbor having a higher value.

# Drawback of Hill Climbing (1)

Hill climbing terminates when a peak is reached with no neighbor having a higher value.

- Local maximum

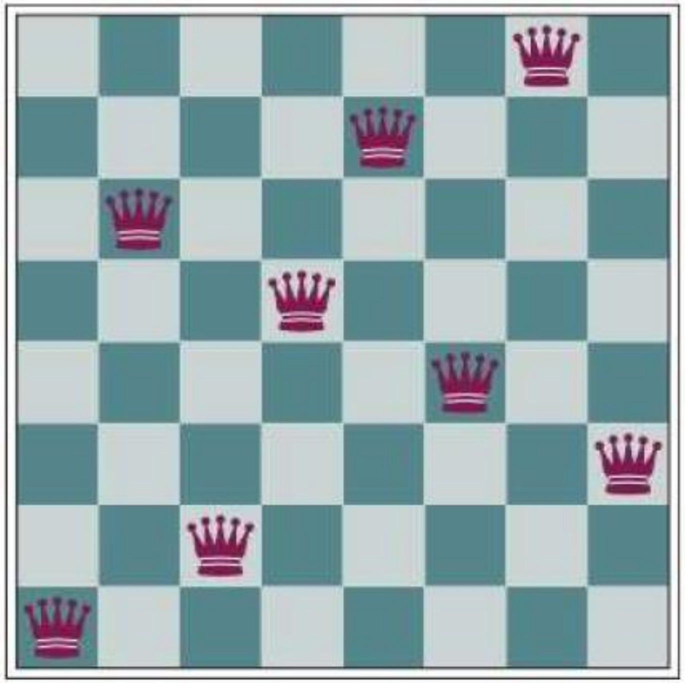
  Not the global maximum.

# Drawback of Hill Climbing (1)

Hill climbing terminates when a peak is reached with no neighbor having a higher value.

- Local maximum

Not the global maximum.

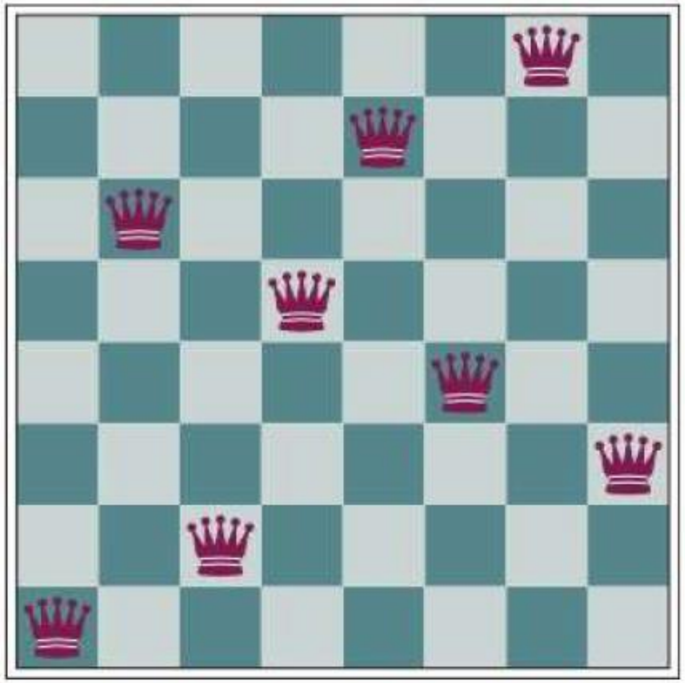


# Drawback of Hill Climbing (1)

Hill climbing terminates when a peak is reached with no neighbor having a higher value.

- Local maximum

Not the global maximum.



Every move of one queen introduces more conflicts.

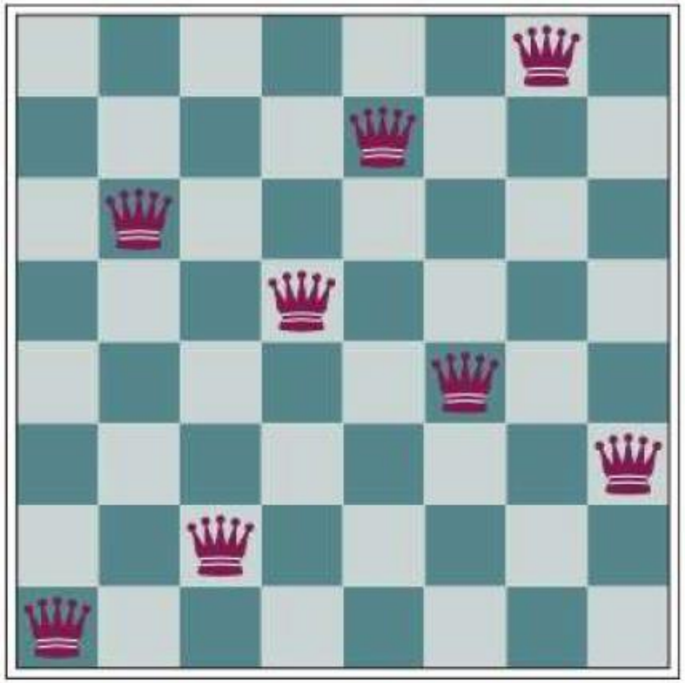
# Drawback of Hill Climbing (1)

Hill climbing terminates when a peak is reached with no neighbor having a higher value.

- Local maximum

  Not the global maximum.

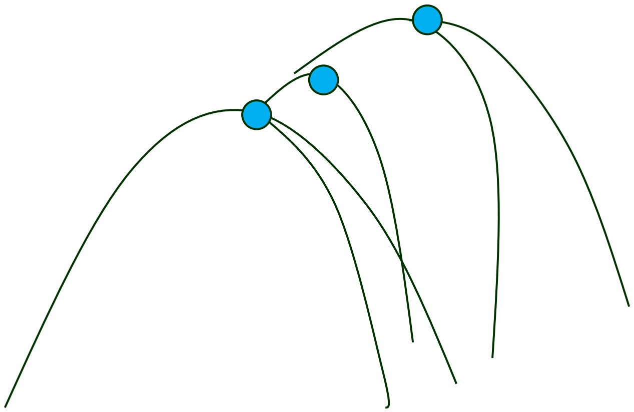
  Hill climbing in the vicinity of a local maximum will be drawn toward it and then get stuck there.



Every move of one queen introduces more conflicts.

# Drawback of Hill Climbing (2)

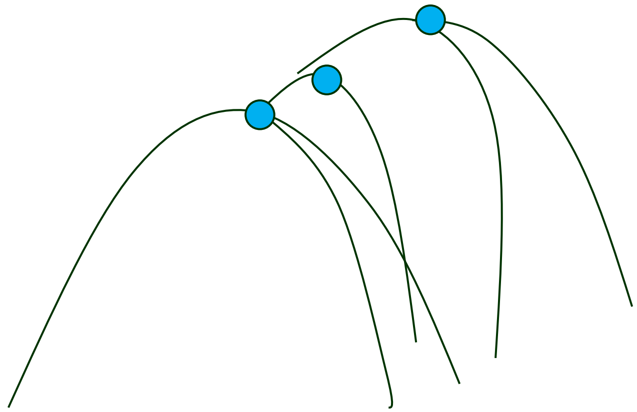
- Ridge: A sequence of local maxima difficult to navigate.



At each local maximum, all available actions are downhill.

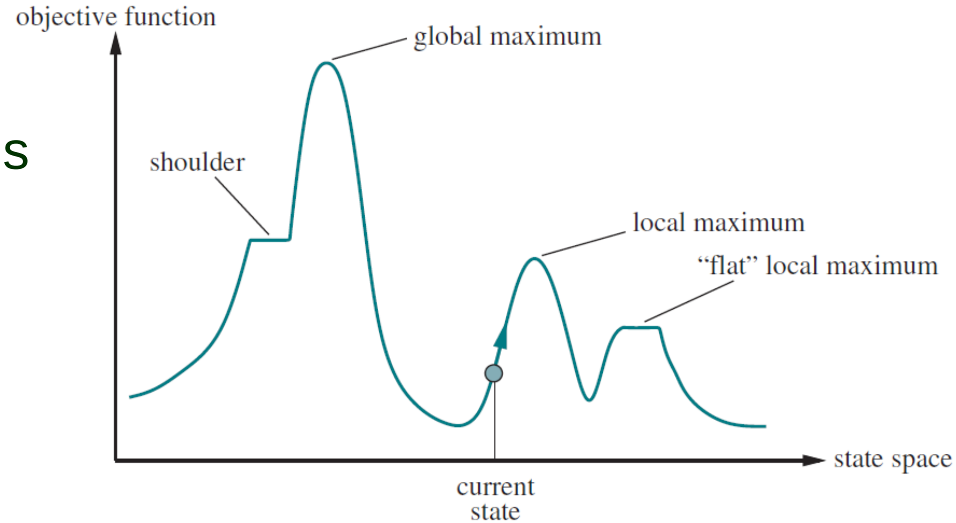
# Drawback of Hill Climbing (2)

♠ Ridge:  A sequence of local maxima difficult to navigate.



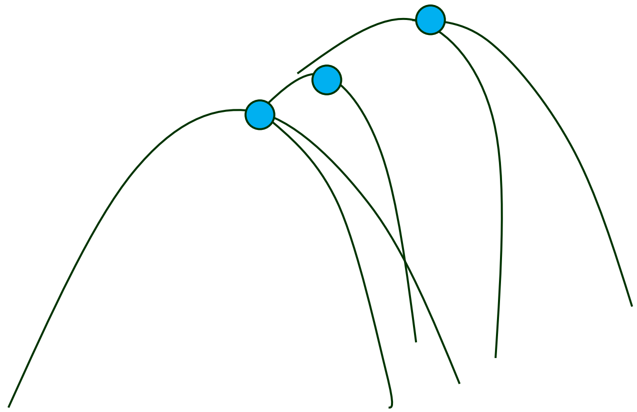
At each local maximum, all available actions are downhill.

♠ Plateaus:  No uphill actions exist.



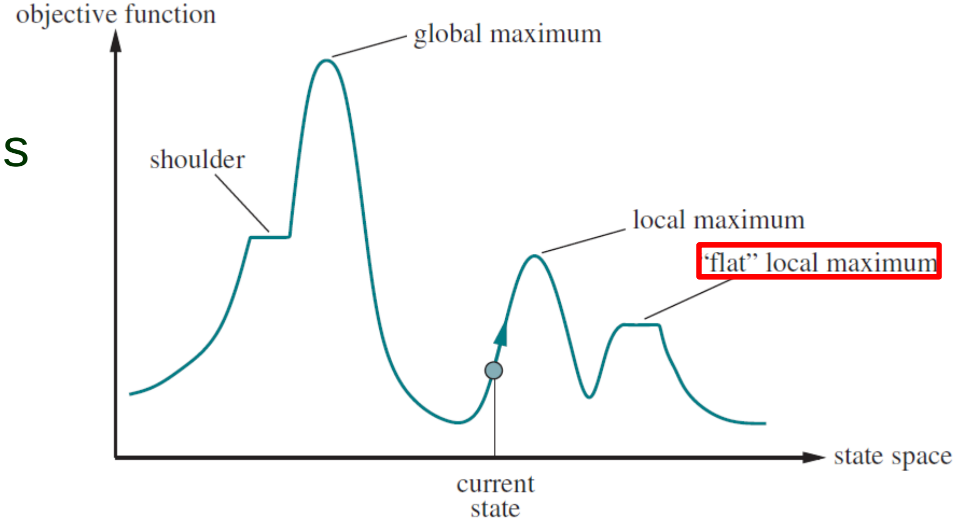
# Drawback of Hill Climbing (2)

♠ Ridge: A sequence of local maxima difficult to navigate.



At each local maximum, all available actions are downhill.

♠ Plateaus: No uphill actions exist.



# Variations of Hill Climbing

- Stochastic hill climbing
  - Random selection among the uphill moves.
  - Probability of selection varying with steepness..

# Variations of Hill Climbing

- Stochastic hill climbing
  - Random selection among the uphill moves.
  - Probability of selection varying with steepness..
- First-choice hill climbing
  - Random generation of successors until a better (than the current) one is found.
  - Useful when many successors exist and/or the objective function is costly to evaluate.

# Variations of Hill Climbing

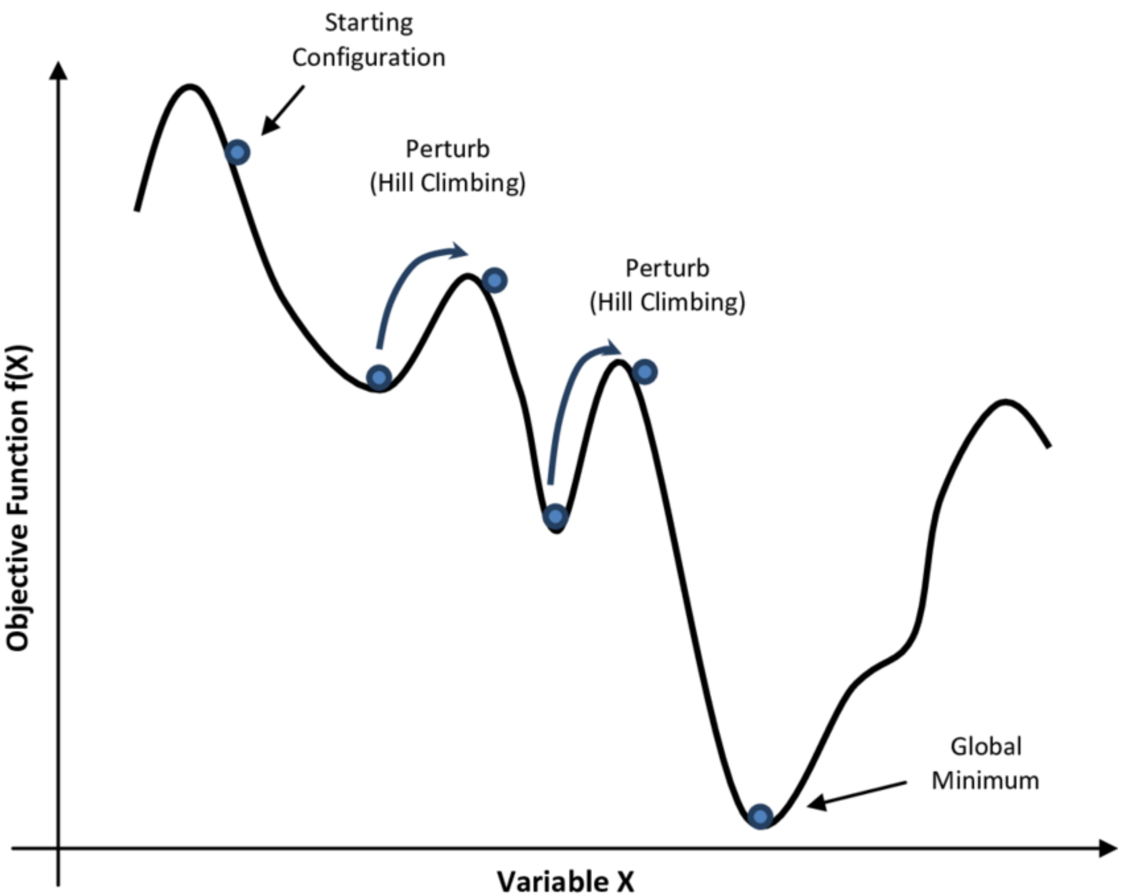
- Stochastic hill climbing
  - Random selection among the uphill moves.
  - Probability of selection varying with steepness..
- First-choice hill climbing
  - Random generation of successors until a better (than the current) one is found.
  - Useful when many successors exist and/or the objective function is costly to evaluate.
- Random restart hill climbing
  - Restart search from random initial state.

# II. Simulated Annealing

Annealing: Heat a metal to a high temperature and then gradually cool it, allowing the material to reach a low-energy crystalline state so it is hardened.

# II. Simulated Annealing

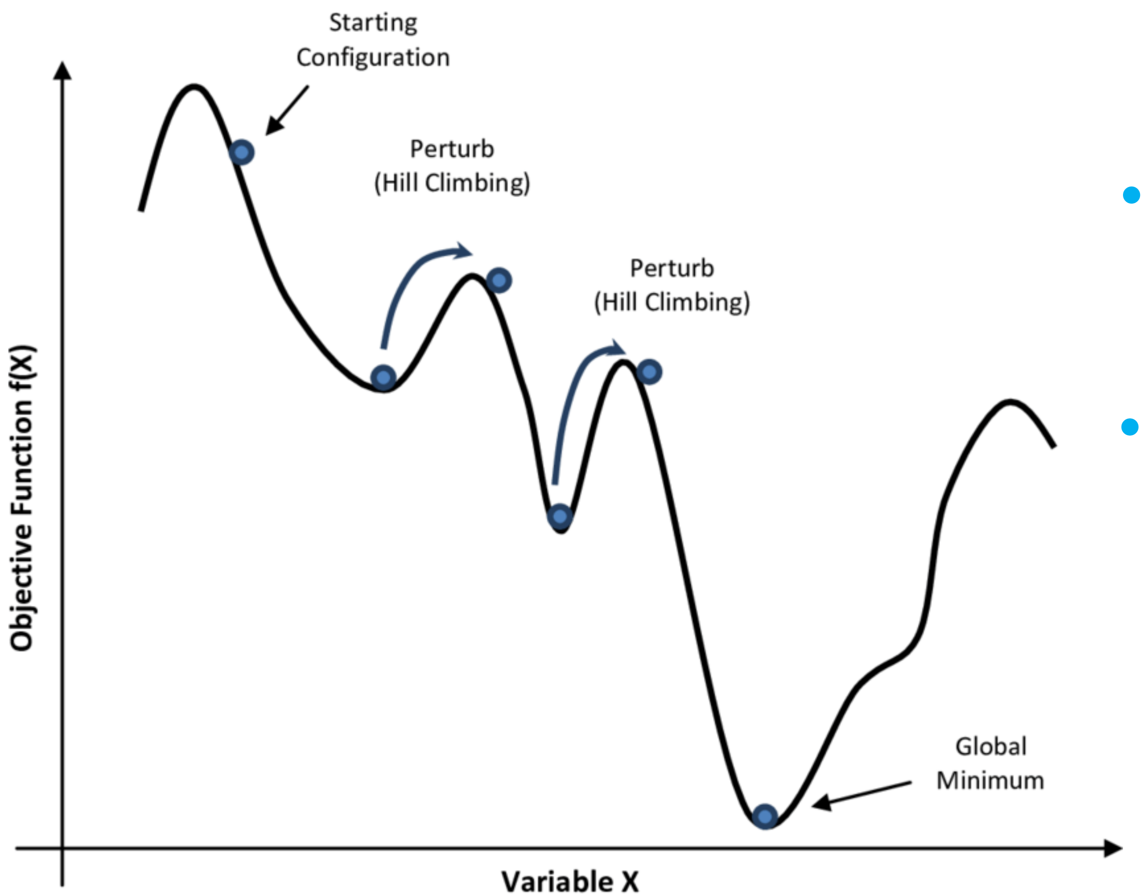
Annealing: Heat a metal to a high temperature and then gradually cool it, allowing the material to reach a low-energy crystalline state so it is hardened.




* from http://www.cs.us.es/~fsancho/?e=206

# II. Simulated Annealing

Annealing: Heat a metal to a high temperature and then gradually cool it, allowing the material to reach a low-energy crystalline state so it is hardened.




- Start by shaking hard (i.e., at high temperature).
- Gradually reduce the intensity of shaking (i.e., lower the temperature).

* from http://www.cs.us.es/~fsancho/?e=206

# Simulated Annealing Algorithm

Minimization

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

# Simulated Annealing Algorithm

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

*Minimization*

# Simulated Annealing Algorithm

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

- Accept the next state if it is an improvement ($\Delta E > 0$).

# Simulated Annealing Algorithm

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

*Minimization*

- Accept the next state if it is an improvement ($\Delta E > 0$).
- Otherwise, accept it with a probability that decreases exponentially
  - as the badness $-\Delta E$ of the move increases, and
  - as the "temperature" goes down.

# Simulated Annealing Algorithm

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

*Minimization*

- Accept the next state if it is an improvement ($\Delta E > 0$).
- Otherwise, accept it with a probability that decreases exponentially
  - as the badness $-\Delta E$ of the move increases, and
  - as the "temperature" goes down.

Bad moves are more tolerated at the start when $T$ is high, and become less likely as $T$ decreases.

# Simulated Annealing Algorithm

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

*Minimization*

- Accept the next state if it is an improvement ($\Delta E > 0$).
- Otherwise, accept it with a probability that decreases exponentially
  - as the badness $-\Delta E$ of the move increases, and
  - as the "temperature" goes down.

Bad moves are more tolerated at the start when $T$ is high, and become less likely as $T$ decreases.

- Escape local minima by allowing bad moves.

# More About SA

- $T \to 0$ slowly enough

# More About SA

- $T \to 0$ slowly enough

$$\Downarrow$$

A property of Boltzmann distribution $e^{\Delta E/T}$ guarantees the global minimum with probability $\to 1$.

# More About SA

- $T \to 0$ slowly enough

$$\Downarrow$$

A property of Boltzmann distribution $e^{\Delta E/T}$ guarantees the global minimum with probability $\to 1$.

- Commonly used $T \leftarrow cT$ with constant $c < 1$ and close to 1 at each step.

# More About SA

- $T \to 0$ slowly enough

$$\Downarrow$$

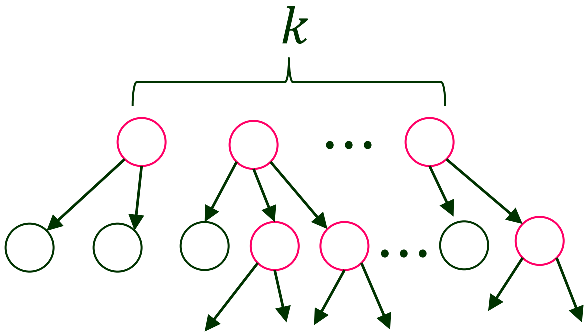
A property of Boltzmann distribution $e^{\Delta E/T}$ guarantees the global minimum with probability $\to 1$.

- Commonly used $T \leftarrow cT$ with constant $c < 1$ and close to 1 at each step.

- Applied to many problems:
  - VLSL layout
  - factory scheduling
  - aircraft trajectory planning
  - NP-hard optimization (i.e., the traveling salesman problem)
  - large-scale stochastic optimization tasks

# Local Beam Search

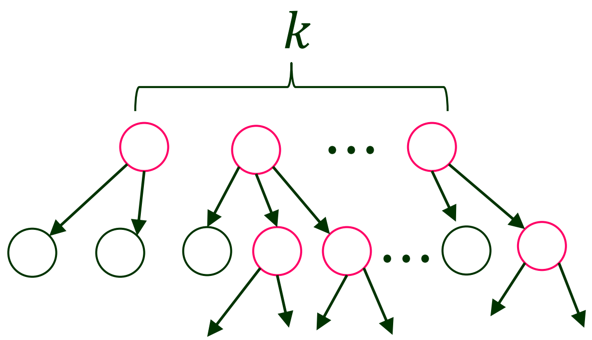
Keep track of $k$ states rather than one.




1. Start with $k$ randomly generated states.
2. Generate all their successors.
3. Stop if any successor is a goal.
4. Otherwise, keep the $k$ best successors and go back to step 2.

# Local Beam Search

Keep track of $k$ states rather than one.

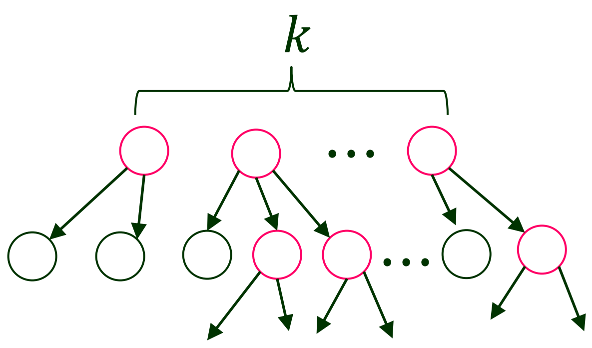



1. Start with $k$ randomly generated states.
2. Generate all their successors.
3. Stop if any successor is a goal.
4. Otherwise, keep the $k$ best successors and go back to step 2.

♣ May suffer from a lack of diversity among the $k$ states.

# Local Beam Search

Keep track of $k$ states rather than one.




1. Start with $k$ randomly generated states.
2. Generate all their successors.
3. Stop if any successor is a goal.
4. Otherwise, keep the $k$ best successors and go back to step 2.

♣ May suffer from a lack of diversity among the $k$ states.

**Solution**: stochastic beam search which chooses successors with probabilities proportional to their values.

# III. Evolutionary Algorithms

- Also called *genetic algorithms*.
- Inspired by natural selection in biology.

# III. Evolutionary Algorithms

- Also called *genetic algorithms*.
- Inspired by natural selection in biology.

1. Start with a population of $k$ randomly generated states (individuals).

# III. Evolutionary Algorithms

- ♣ Also called *genetic algorithms*.
- ♣ Inspired by natural selection in biology.

1. Start with a population of $k$ randomly generated states (individuals).

2. Select the *most fit* individuals to become parents of the next generation

# III. Evolutionary Algorithms

- ♣ Also called *genetic algorithms*.
- ♣ Inspired by natural selection in biology.

1. Start with a population of $k$ randomly generated states (individuals).
2. Select the *most fit* individuals to become parents of the next generation
3. Combine every $\rho$ parents to form an offspring (typically $\rho = 2$).

# III. Evolutionary Algorithms

- Also called *genetic algorithms*.
- Inspired by natural selection in biology.

1. Start with a population of $k$ randomly generated states (individuals).
2. Select the *most fit* individuals to become parents of the next generation
3. Combine every $\rho$ parents to form an offspring (typically $\rho = 2$).

Crossover: Split each of the parent strings and recombine the parts to form two children.

# III. Evolutionary Algorithms

- ♣ Also called *genetic algorithms*.
- ♣ Inspired by natural selection in biology.

1. Start with a population of $k$ randomly generated states (individuals).

2. Select the *most fit* individuals to become parents of the next generation

3. Combine every $\rho$ parents to form an offspring (typically $\rho = 2$).

Crossover: Split each of the parent strings and recombine the parts to form two children.

Mutation: Randomly change the bits of an offspring.

# III. Evolutionary Algorithms

- ♣ Also called *genetic algorithms*.
- ♣ Inspired by natural selection in biology.

1. Start with a population of $k$ randomly generated states (individuals).
2. Select the *most fit* individuals to become parents of the next generation
3. Combine every $\rho$ parents to form an offspring (typically $\rho = 2$).

Crossover: Split each of the parent strings and recombine the parts to form two children.

Mutation: Randomly change the bits of an offspring.

Culling: All individuals below a threshold are discarded from the population.

# III. Evolutionary Algorithms

- Also called *genetic algorithms*.
- Inspired by natural selection in biology.

1. Start with a population of $k$ randomly generated states (individuals).
2. Select the *most fit* individuals to become parents of the next generation
3. Combine every $\rho$ parents to form an offspring (typically $\rho = 2$).

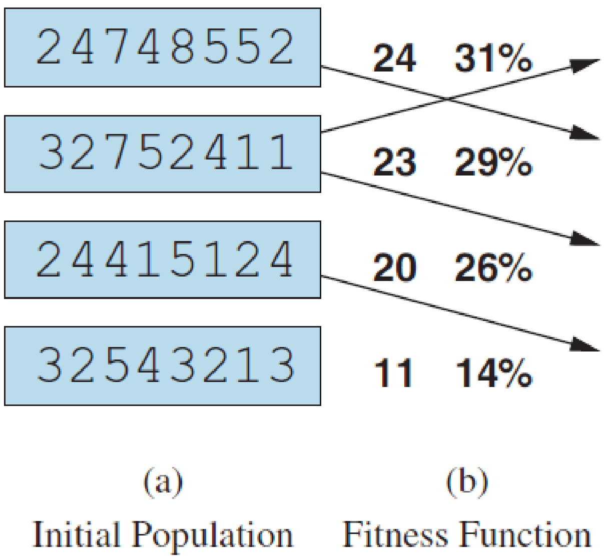
   Crossover: Split each of the parent strings and recombine the parts to form two children.

   Mutation: Randomly change the bits of an offspring.

   Culling: All individuals below a threshold are discarded from the population.

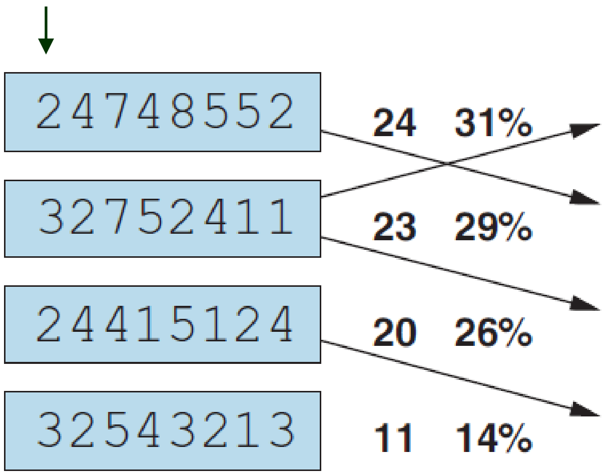
4. Go back to step 2 and repeat until *sufficiently fit* states are discovered (in which case the best one is chosen as a solution).

# Genetic Algorithm on 8-Queen



# Genetic Algorithm on 8-Queen

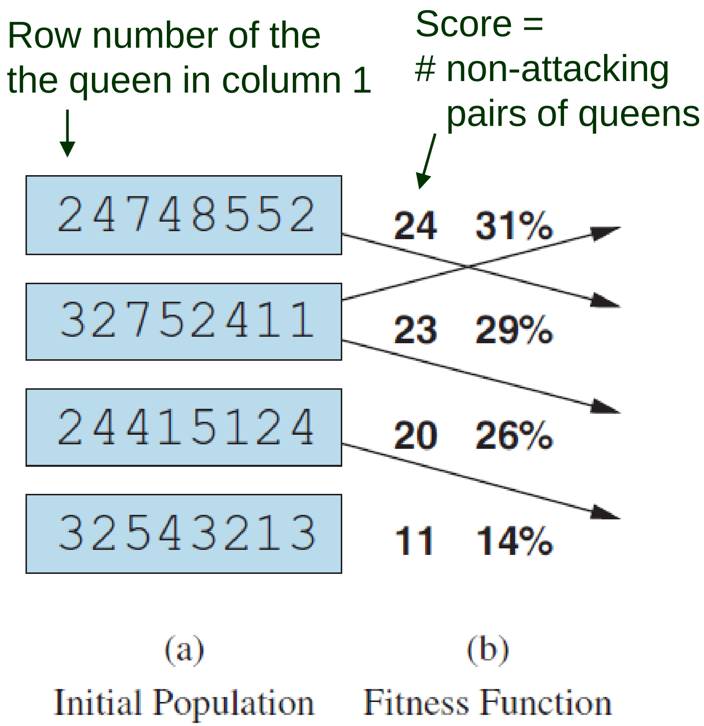
Row number of the
the queen in column 1




(a) Initial Population

(b) Fitness Function

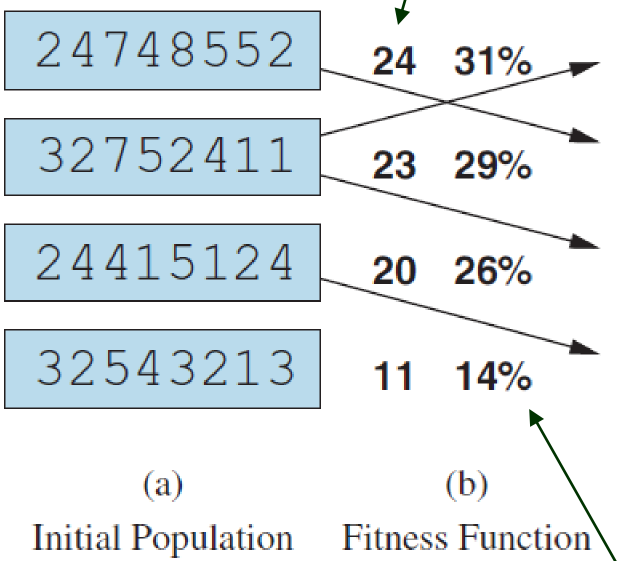
# Genetic Algorithm on 8-Queen



# Genetic Algorithm on 8-Queen

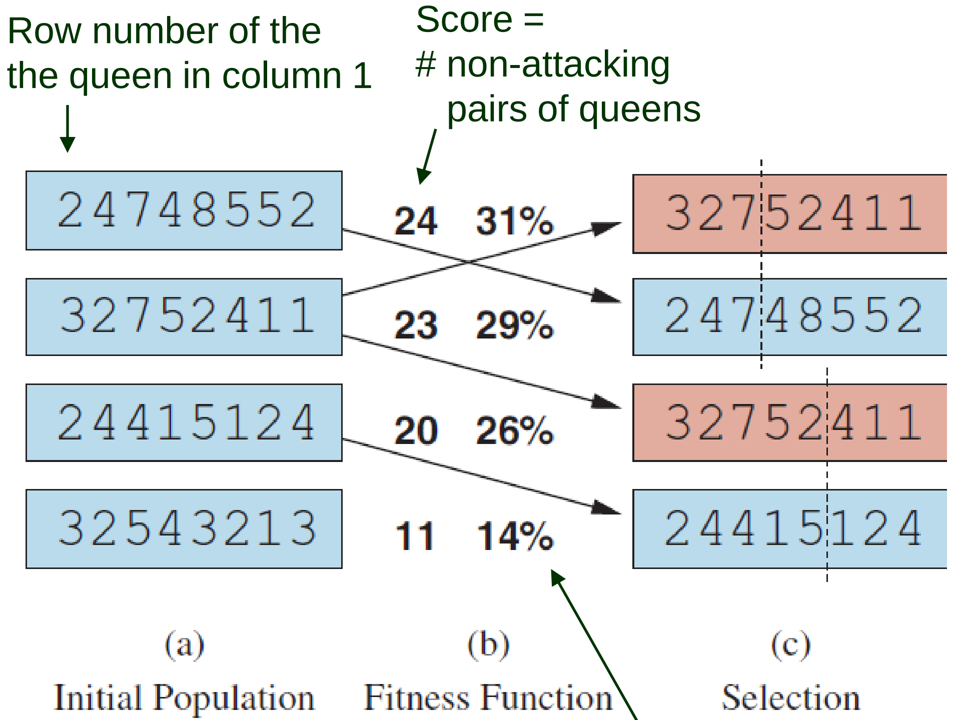
Row number of the
the queen in column 1

Score =
# non-attacking
pairs of queens



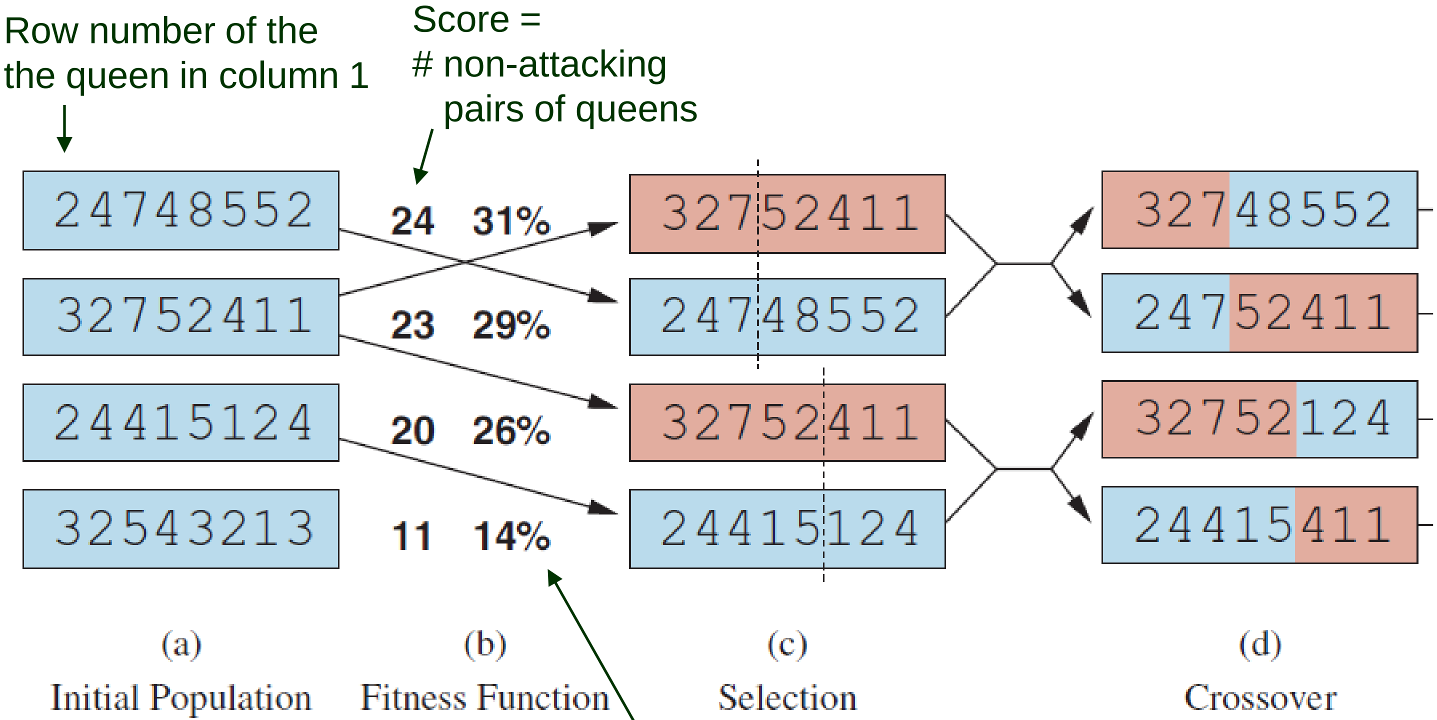

$$14\% = \frac{11}{24 + 23 + 20 + 11}$$

# Genetic Algorithm on 8-Queen



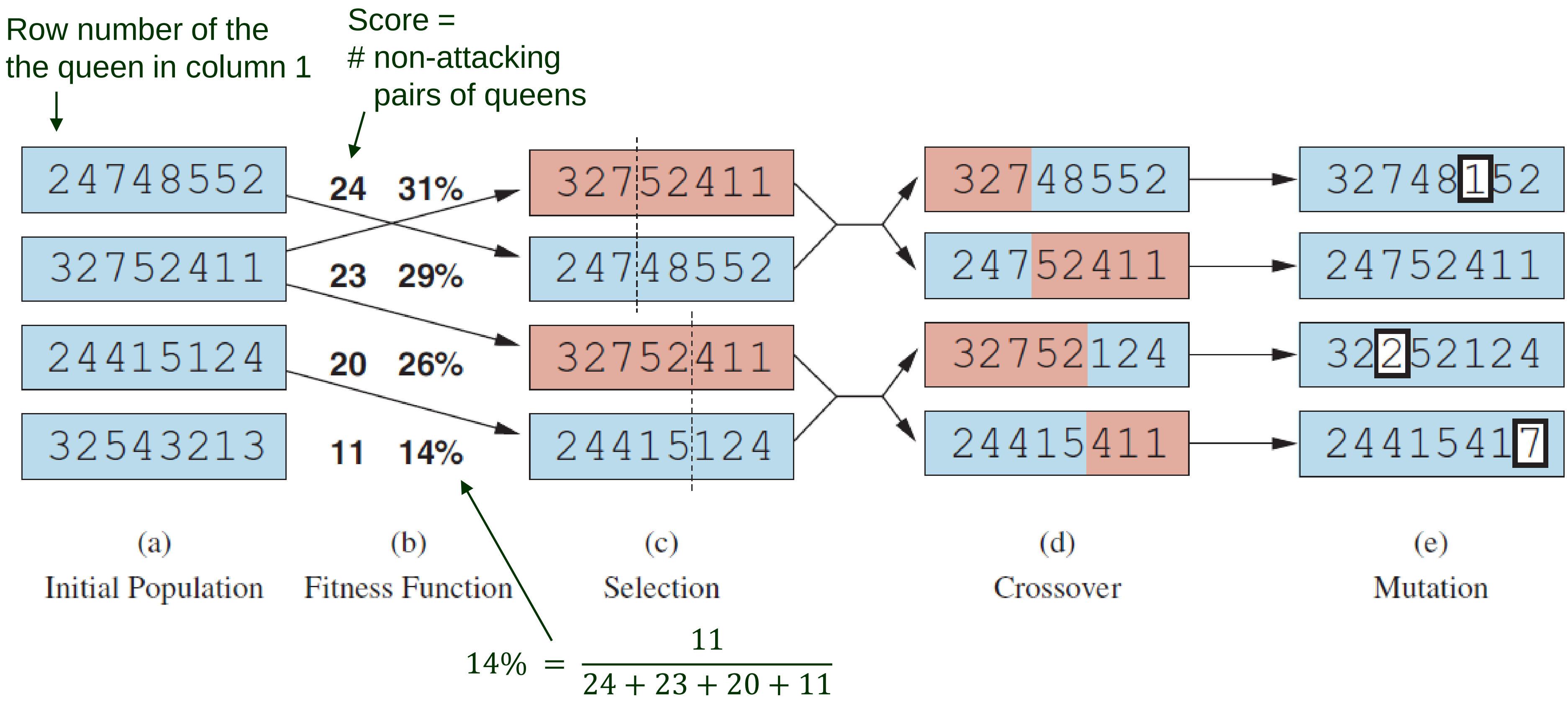

$$14\% = \frac{11}{24 + 23 + 20 + 11}$$

# Genetic Algorithm on 8-Queen

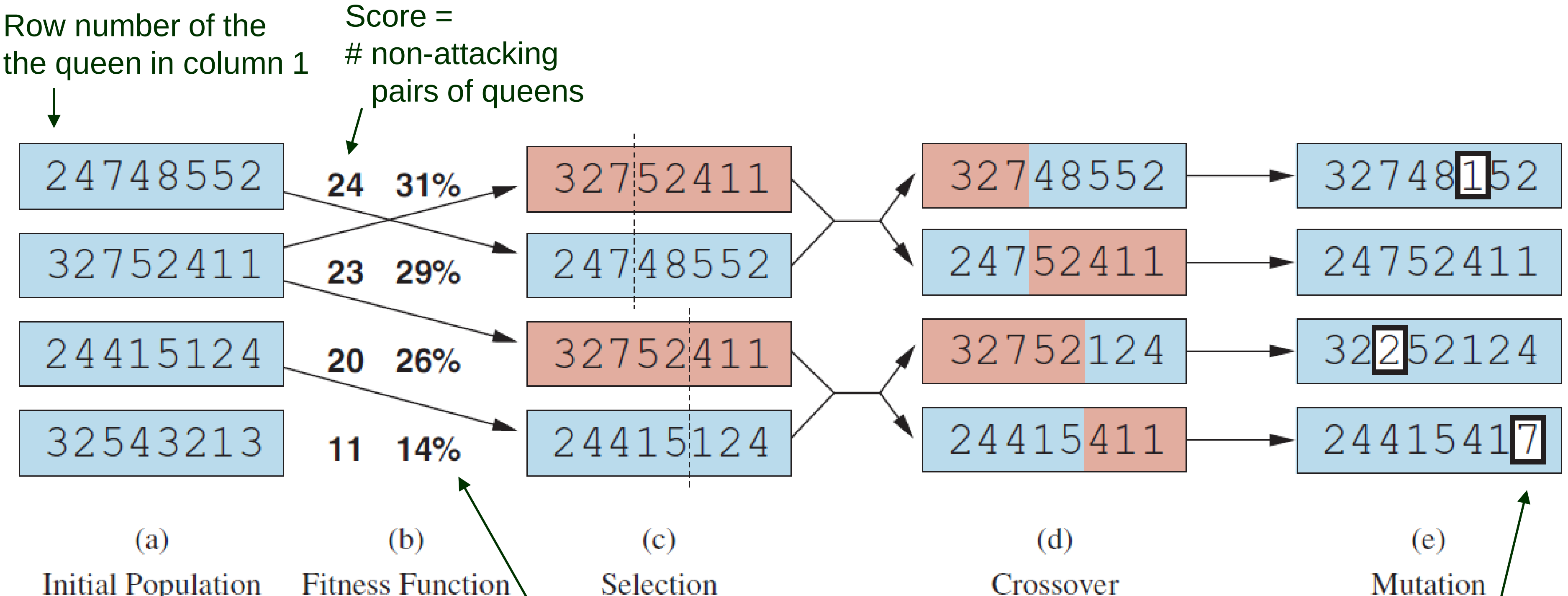



$$14\% = \frac{11}{24 + 23 + 20 + 11}$$

# Genetic Algorithm on 8-Queen



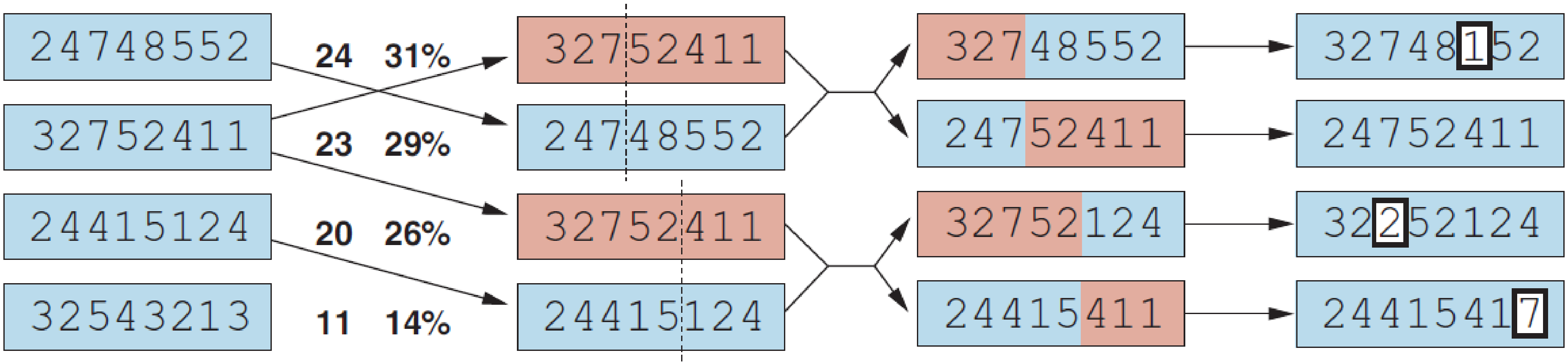
# Genetic Algorithm on 8-Queen




$$14\% = \frac{11}{24 + 23 + 20 + 11}$$

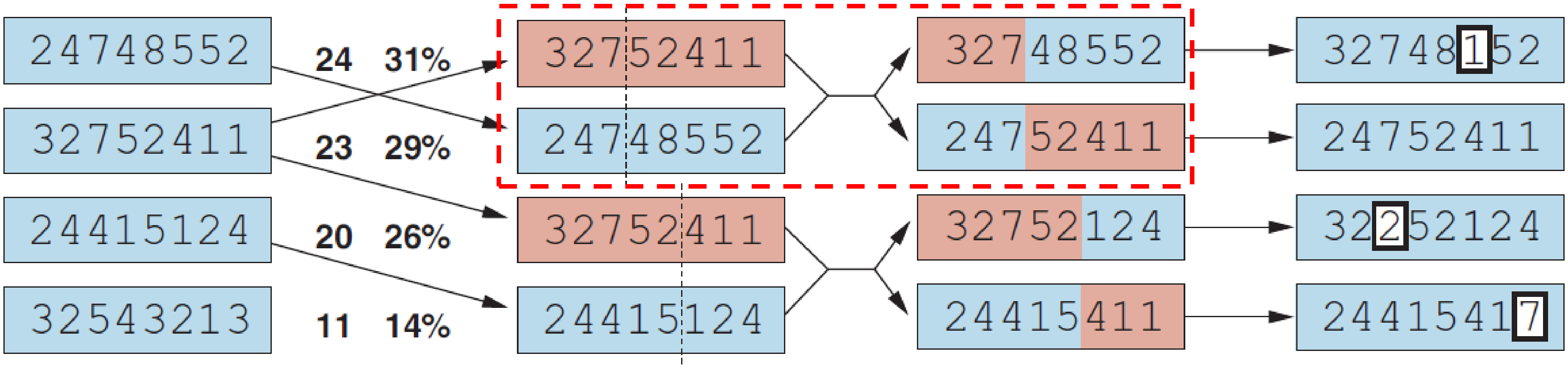
Move the queen in column 8 to a random square (7).

# Crossover



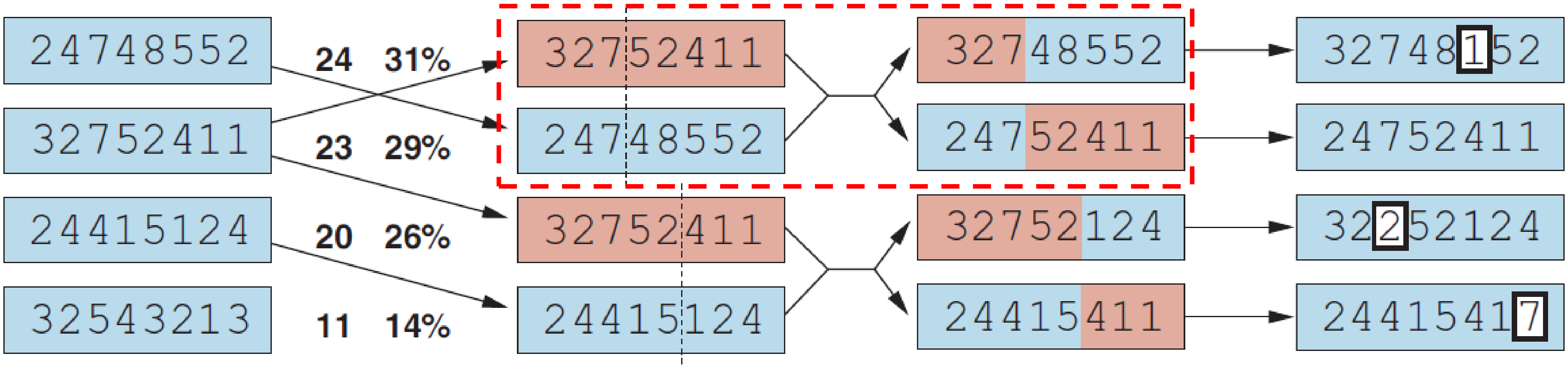

(d)
Crossover

# Crossover



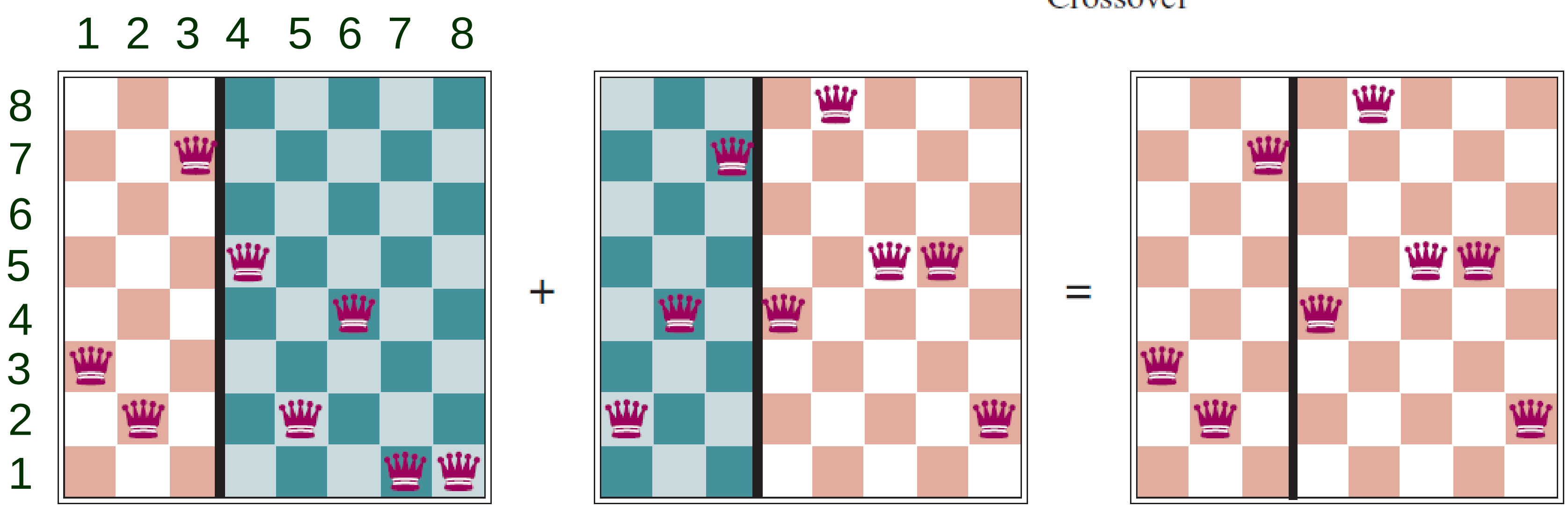

(d)
Crossover

# Crossover




(d)

Crossover



# Genetic Algorithm (Pseudocode)

**function** GENETIC-ALGORITHM(*population*, *fitness*) **returns** an individual
  **repeat**
    *weights* ← WEIGHTED-BY(*population*, *fitness*)
    *population2* ← empty list
    **for** $i$ = 1 **to** SIZE(*population*) **do**
      *parent1*, *parent2* ← WEIGHTED-RANDOM-CHOICES(*population*, *weights*, 2)
      *child* ← REPRODUCE(*parent1*, *parent2*)
      **if** (small random probability) **then** *child* ← MUTATE(*child*)
      add *child* to *population2*
    *population* ← *population2*
  **until** some individual is fit enough, or enough time has elapsed
  **return** the best individual in *population*, according to *fitness*

**function** REPRODUCE(*parent1*, *parent2*) **returns** an individual
  $n$ ← LENGTH(*parent1*)
  $c$ ← random number from 1 to $n$
  **return** APPEND(SUBSTRING(*parent1*, 1, $c$), SUBSTRING(*parent2*, $c + 1$, $n$))

# Applications of GA

- Complex structured problems

  Circuit layout, job-shop scheduling

- Evolving the architecture of deep neural networks
- Finding bugs of hardware
- Molecular structure optimization
- Image processing.
- Learning robots, etc.

# Applications of GA

- Complex structured problems

    Circuit layout, job-shop scheduling

- Evolving the architecture of deep neural networks
- Finding bugs of hardware
- Molecular structure optimization
- Image processing.
- Learning robots, etc.