Fundamentals of Artificial Intelligence

Local Search Algorithms

Local Search

- The *uninformed and informed search algorithms* that we have seen are designed to explore search spaces systematically.
  - They keep one or more paths in memory and by record which alternatives have been explored at each point along the path.
  - When a goal is found, *the path to that goal also constitutes a solution to the problem*.
  - In many problems, however, the path to the goal is irrelevant.

- If ***the path to the goal does not matter***, we might consider a different class of algorithms that do not worry about paths at all.

  ➔ **local search algorithms**

Local Search

- **Local search algorithms** operate using a *single current node* and generally move only to neighbors of that node.
- **Local search algorithms** ease up on ***completeness and optimality*** in the interest of ***improving time and space complexity***?
- Although **local search algorithms** are not ***systematic***, they have ***two key advantages***:
  1. They use ***very little memory*** (usually a constant amount), and
  2. They can often find ***reasonable solutions*** in large or infinite (continuous) state spaces.

- In addition to finding goals, **local search algorithms** are useful for solving pure **optimization problems**, in which the aim is ***to find the best state*** according to an ***objective function***.
  - In optimization problems, ***the path to goal is irrelevant*** and ***the goal state itself is the solution.***
  - In some optimization problems, the ***goal is not known*** and ***the aim is to find the best state***.

Hill Climbing Search
(Steepest Ascent/Descent)

- At each iteration, the **hill-climbing search algorithm** moves to the ***best successor*** of the ***current node*** according to an ***objective function.***
  - Best successor is the successor with best value (highest or lowest) according to an objective function.
  - If no successors have better value than the current value, it returns.
  - It moves in direction of uphill (hill climbing).
  - It terminates when it reaches a “peak” where no neighbor has a higher value.
- The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function.
- **Hill climbing** does not look ahead beyond the immediate neighbors of the current state.
- **Hill climbing** is sometimes called ***greedy local search*** because it grabs a good neighbor state without thinking ahead about where to go next.
  - Greedy algorithms often perform quite well and
  - Hill climbing often makes rapid progress toward a solution.

Hill Climbing Search
(Steepest Ascent/Descent)

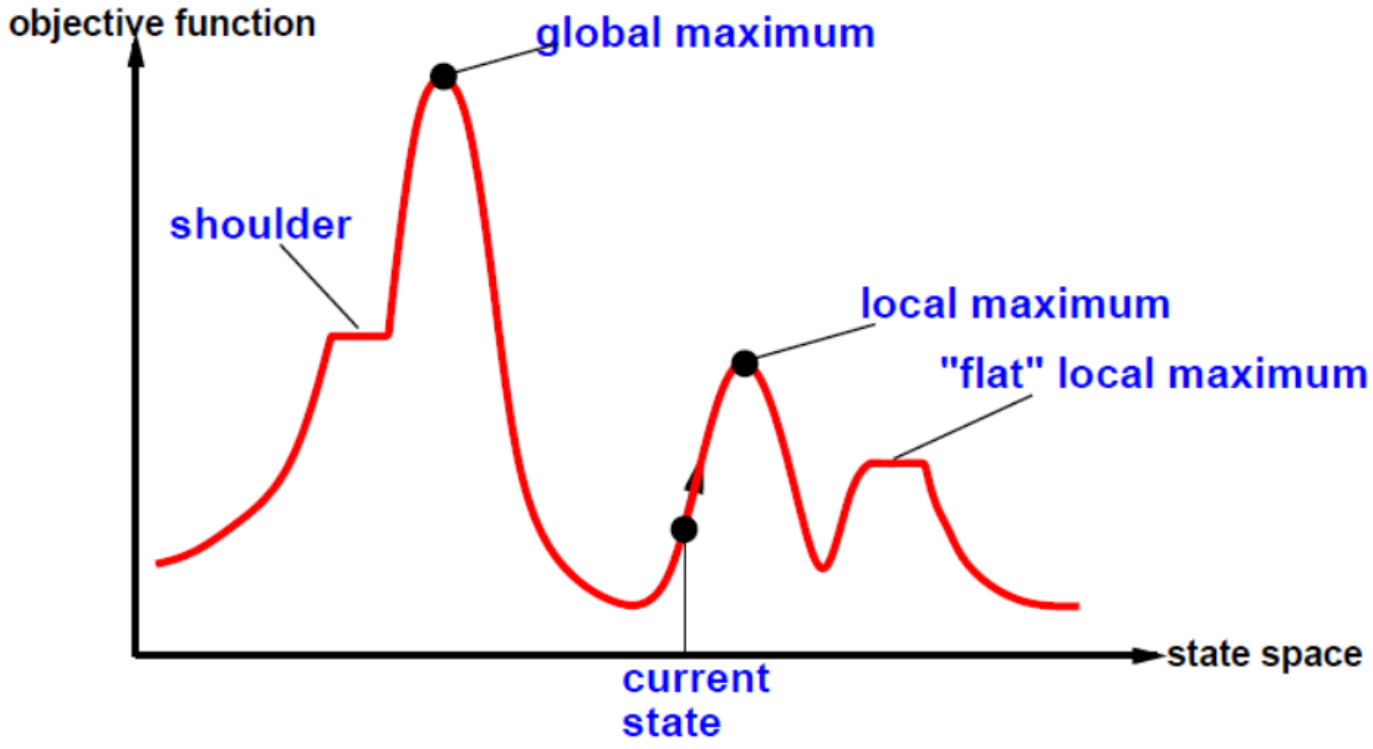
function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

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

best successor

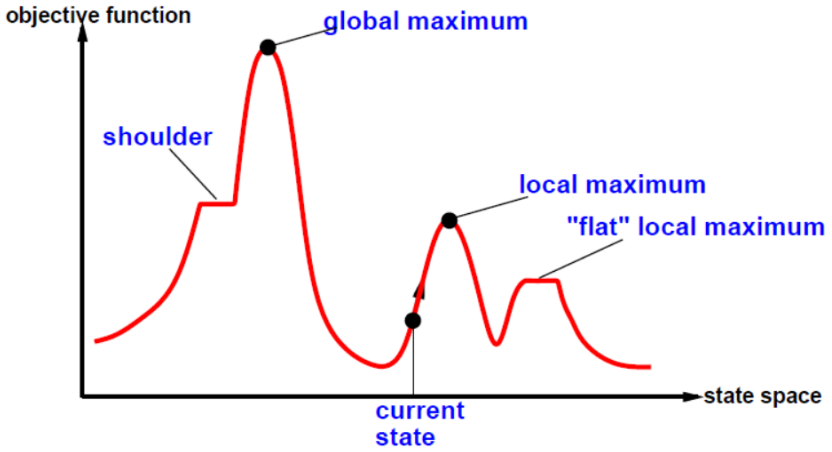
Hill Climbing Search

- To understand local search, it is useful to consider the ***state-space landscape.***
- The aim is to find the highest peak - **a global maximum**.
- Hill-climbing search modifies the current state to try to improve it.



Hill Climbing Search

- A **local maximum** is a peak that is higher than each of its neighboring states but lower than the **global maximum**.
  - Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go.
- A **plateau** is a **flat area** of the state-space landscape. It can be a **flat local maximum**, from which no uphill exit exists, or a **shoulder**, from which progress is possible.
  - A hill-climbing search might get lost on the plateau.
  - **Random sideways moves** can escape from ***shoulders*** but they loop forever on ***flat maxima.***

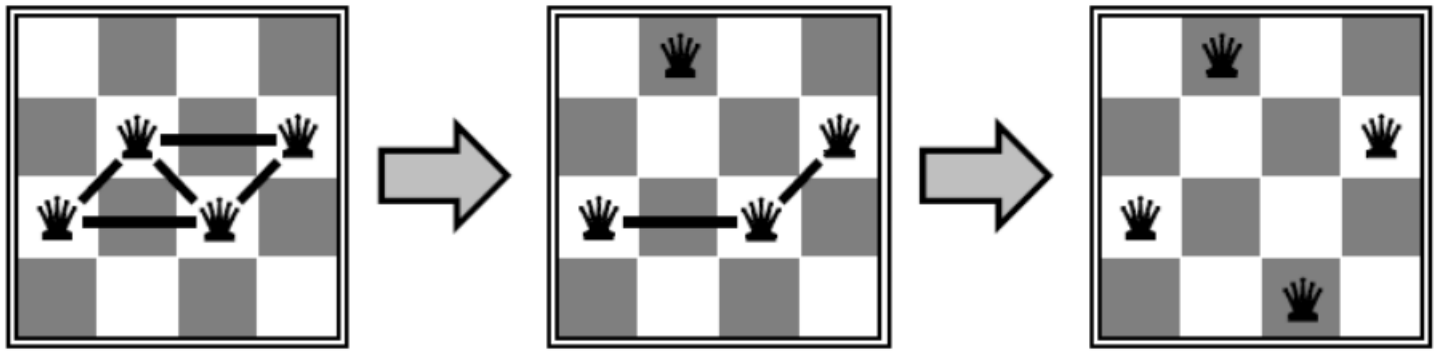


Hill Climbing Example

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.

➔ **Objective function: number of conflicts (no conflicts is global minimum)**

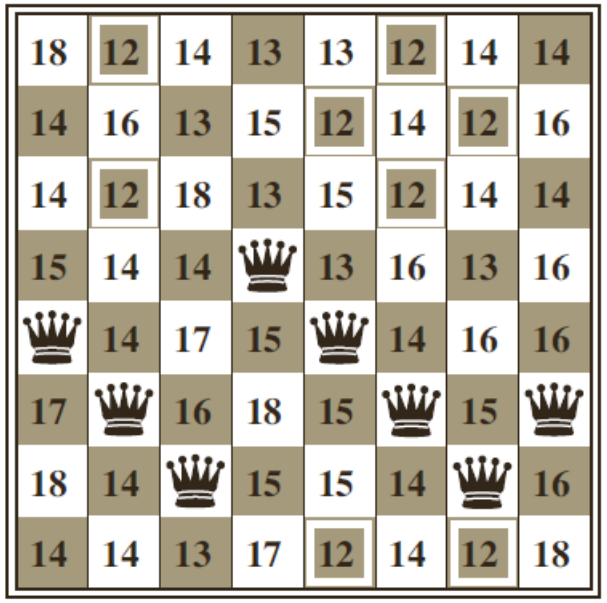


- The successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has n*(n-1) successors).
- The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly.
- The global minimum of this function is zero, which occurs only at perfect solutions.

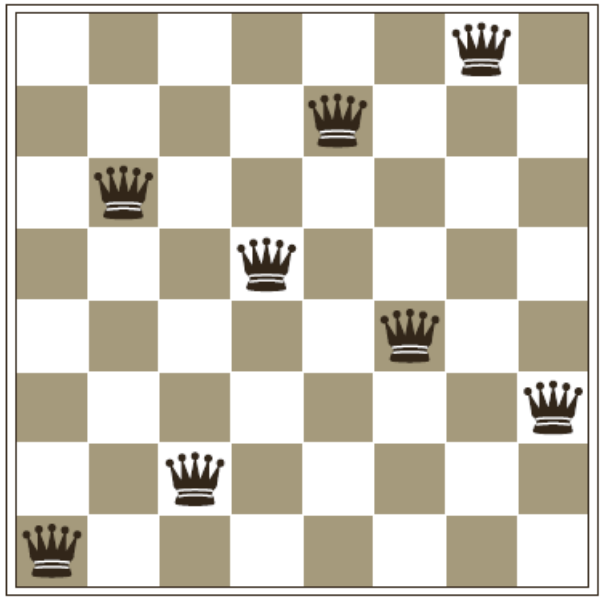
Hill Climbing Example

n-queens

- Hill Climbing may NOT reach to a goal state for n-queens problem.



➔ five steps to reach this state



- An 8-queens state with heuristic cost estimate h=17
- The value of h for each possible successor obtained by moving a queen within its column.
- The best moves are marked with value 12.

- A local minimum in the 8-queens state space; the state has h=1
- but every successor has a higher cost.
- Hill Climbing will stuck here

Hill Climbing Example

n-queens

- Starting from a randomly generated 8-queens state, **steepest-ascent hill climbing** gets stuck 86% of the time, solving only 14% of problem instances.

- The Hill Climbing algorithm halts if it reaches a **plateau.**
  - One possible solution is to allow **sideways move** in the hope that the **plateau** is really a **shoulder**.
  - If we always allow sideways moves when there are no uphill moves, an infinite loop will occur whenever the algorithm reaches a flat local maximum that is not a shoulder.
  - One common solution is to put a limit on the number of consecutive sideways moves allowed.
  - For example, we could allow up to 100 consecutive sideways moves in the 8-queens problem. This raises the percentage of problem instances solved by hill climbing from 14% to 94%.

Hill Climbing Example

8-puzzle

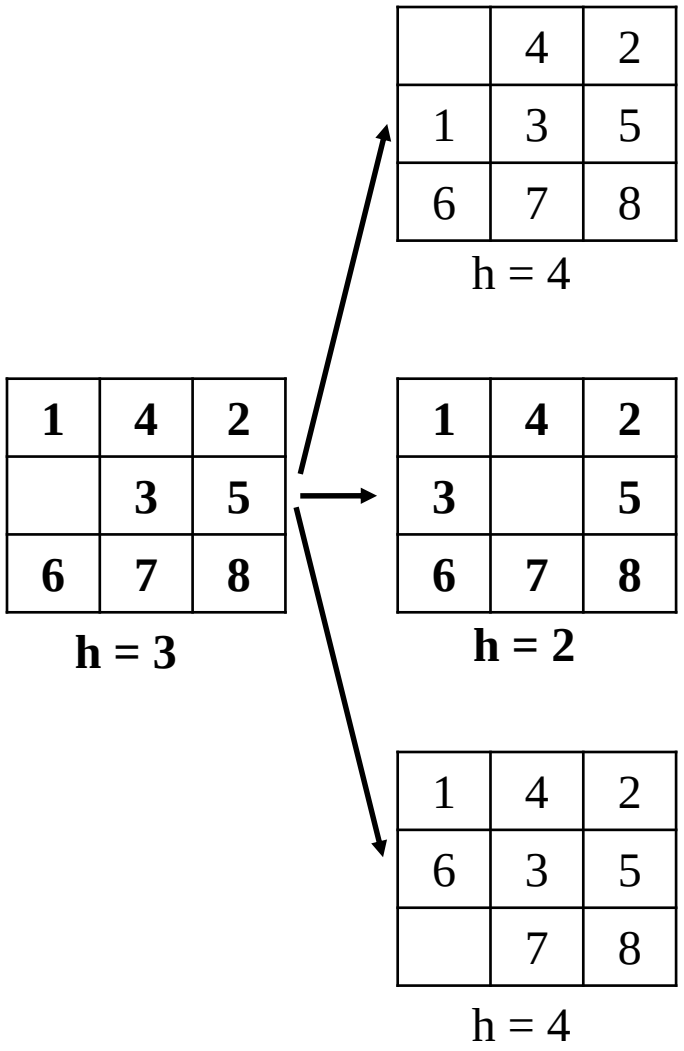
Heuristic function is Manhattan Distance

1	**4**	**2**
	3	**5**
6	**7**	**8**

h = 3

Hill Climbing Example

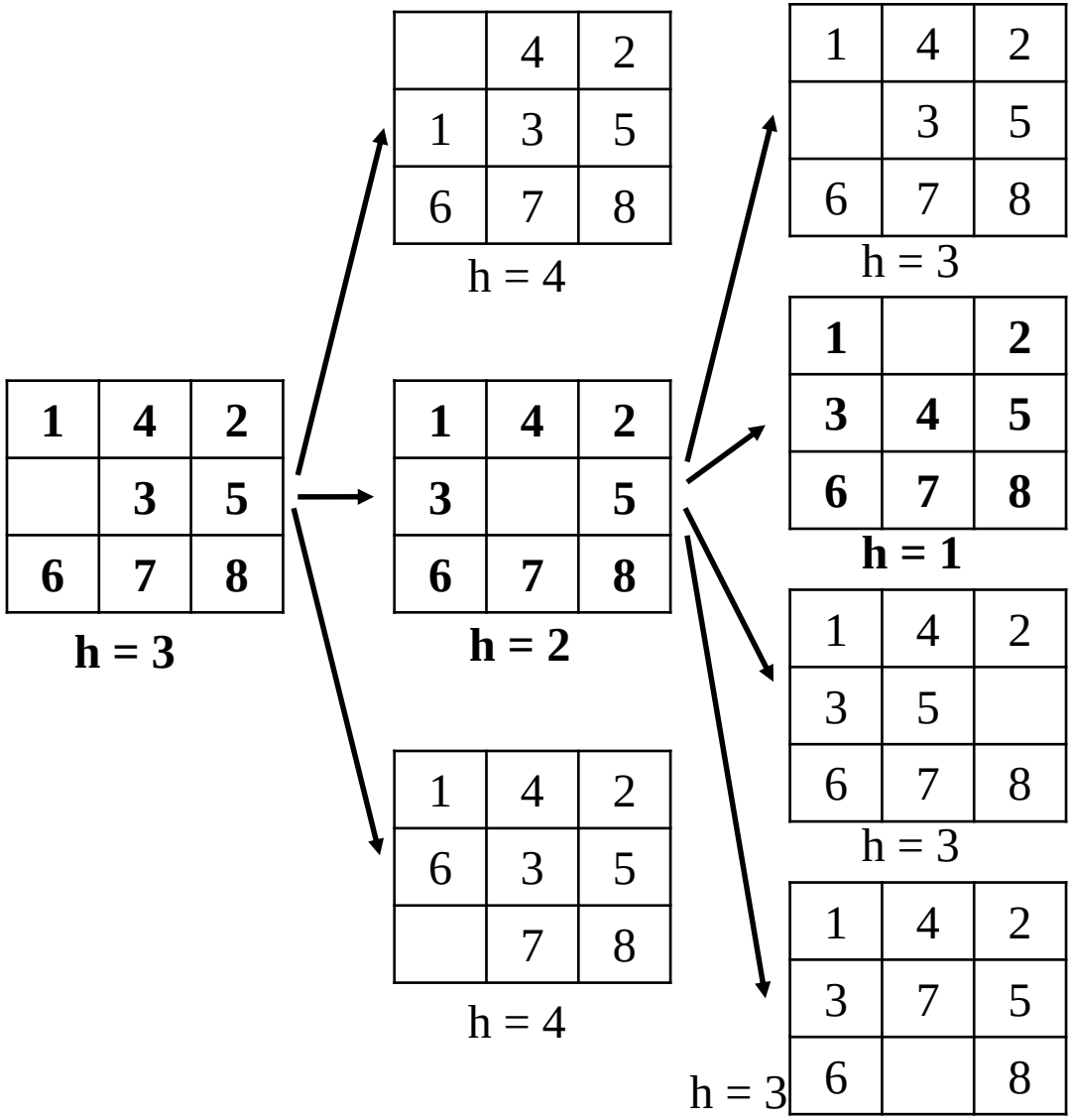
8-puzzle



Heuristic function is Manhattan Distance

Hill Climbing Example

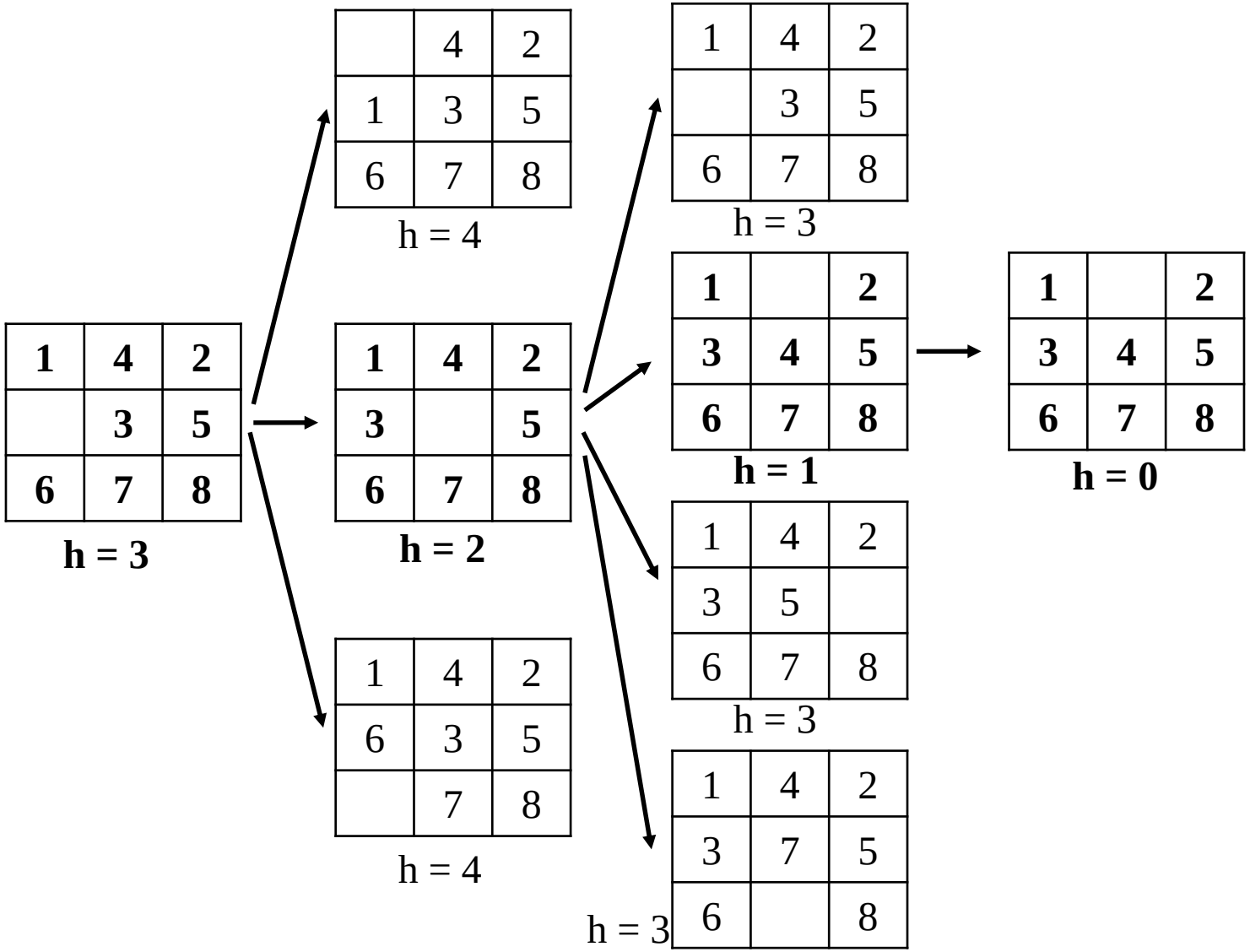
8-puzzle



Heuristic function is Manhattan Distance

Hill Climbing Example

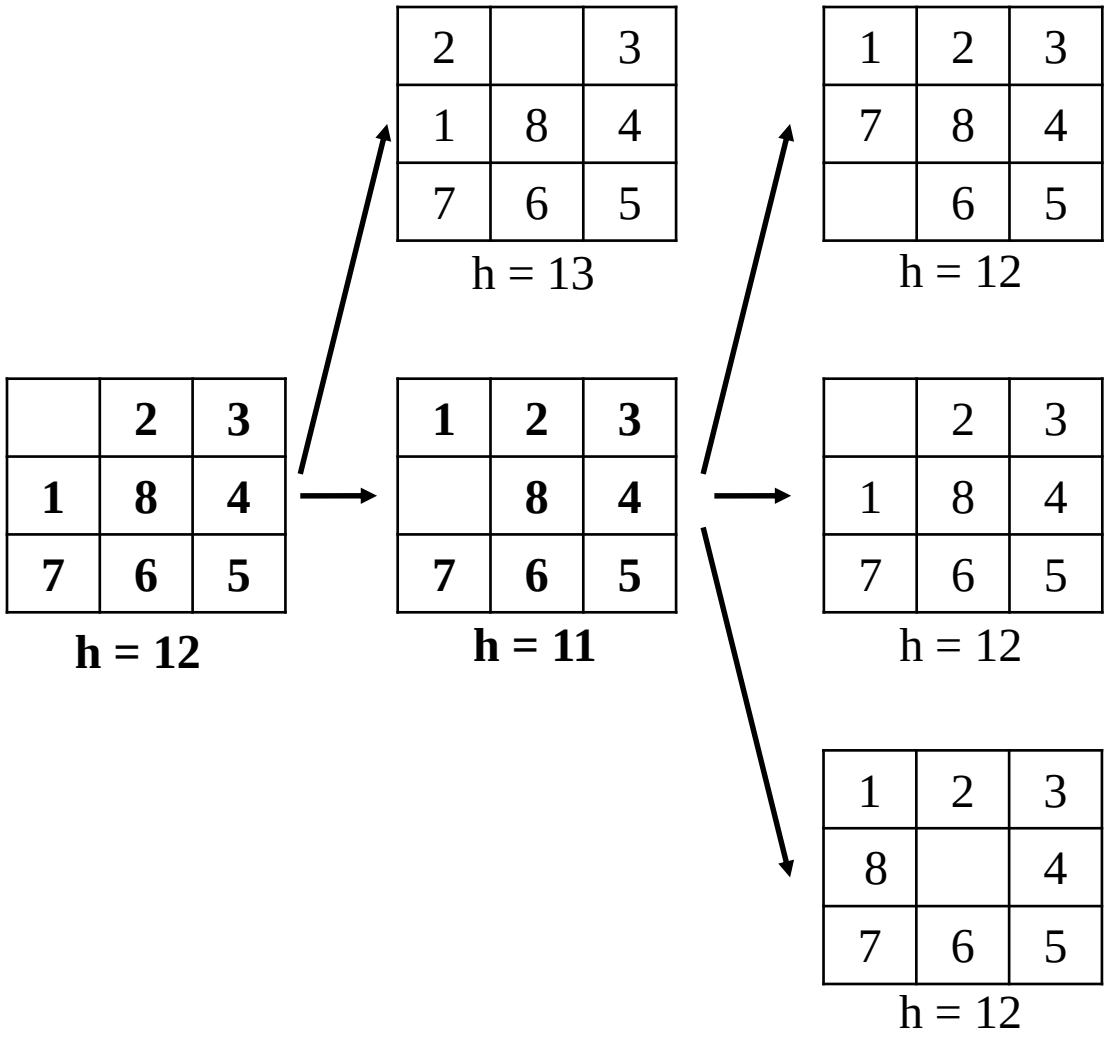
8-puzzle: a solution case



Heuristic function is Manhattan Distance

Hill Climbing Example

8-puzzle: stuck at local maximum



Heuristic function is Manhattan Distance

We are stuck with a local maximum.

Hill Climbing

- Hill Climbing is **NOT complete**.
- Hill Climbing is **NOT optimal**.

- **Why use local search?**
  - Low memory requirements – usually constant
  - Effective – Can often find good solutions in extremely large state spaces
  - Randomized variants of hill climbing can solve many of the drawbacks in practice.

- Many variants of hill climbing have been invented.

Stochastic Hill Climbing

- **Stochastic hill climbing** chooses at random from among the uphill moves;
- The **probability of selection** can vary with the ***steepness of the uphill move***.
- **Stochastic hill climbing** usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.

- **Stochastic hill climbing** is NOT complete, but it may be less likely to get stuck.

First-Choice Hill Climbing

- **First-choice hill climbing** implements ***stochastic hill climbing*** by generating successors randomly until one is generated that is better than the current state.
- This is a good strategy when a state has many of successors.

- **First-choice hill climbing** is also NOT complete,

Random-Restart Hill Climbing

- **Random-Restart Hill Climbing** conducts a ***series of hill-climbing searches*** from ***randomly generated initial states***, until a ***goal is found***.
- Random-Restart Hill Climbing is **complete** if ***infinite (or sufficiently many tries) are allowed***.
- If each hill-climbing search has a ***probability p*** of success, then the ***expected number of restarts required*** is **1/p**.
- For 8-queens instances with no sideways moves allowed, $p \approx 0.14$, so we need roughly 7 iterations to find a goal (6 failures and 1 success).
  - For 8-queens, then, random-restart hill climbing is very effective indeed.
  - Even for three million queens, the approach can find solutions in under a minute.
- The ***success of hill climbing*** depends very much on the ***shape of the state-space landscape***:
  - If there are few local maxima and plateau, ***random-restart hill climbing*** will find a good solution very quickly.
  - On the other hand, many real problems have ***many local maxima*** to get stuck on.
  - NP-hard problems typically have an ***exponential number of local maxima*** to get stuck on.

Simulated Annealing

- A **hill-climbing algorithm** that never makes "downhill" moves toward states with lower value (or higher cost) is guaranteed to be **incomplete**, because ***it can get stuck on a local maximum***.
  - In contrast, a **purely random walk**—that is, moving to a successor chosen uniformly at random from the set of successors—is **complete but extremely inefficient**.
  - Therefore, it seems reasonable to ***combine hill climbing with a random walk in some way that yields both efficiency and completeness***.
- **Idea:** ***escape local maxima by allowing some "bad" moves but gradually decrease their size and frequency.***
- The **simulated annealing algorithm**, a version of ***stochastic hill climbing*** where some downhill moves are allowed.
- **Annealing:** the process of gradually cooling metal to allow it to form stronger crystalline structures
- **Simulated annealing algorithm:** gradually "cool" search algorithm from Random Walk to First-Choice Hill Climbing

Simulated Annealing

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

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

- **Downhill moves** are accepted readily early in the annealing schedule and then less often as time goes on.
- The **schedule** input determines the value of the temperature T as a function of time.

Simulated Annealing

- Instead of picking the best move, Simulated Annealing picks a random move.
  - If the move improves the situation, it is always accepted.
  - Otherwise, the algorithm accepts the move with some probability less than 1.
- The probability decreases exponentially with the “badness” of the move—the amount ΔE by which the evaluation is worsened.
- The probability also decreases as the “temperature” T goes down: “bad” moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases.
- **If the schedule lowers T slowly enough, the algorithm will find a best state with probability approaching 1.**

- Simulated Annealing is widely used in VLSI layout and airline scheduling.

Local Beam Search

- The **local beam search algorithm** keeps track of ***k states rather than just one***.
  - It begins with k randomly generated states.
  - At each step, all the successors of all k states are generated.
  - If any one is a goal, the algorithm halts.
  - Otherwise, it selects the k best successors from the complete list and repeats.

- The **local beam search algorithm** is **not** the same as ***k searches run in parallel***!
  - In a **local beam search**, searches that find good states recruit other searches to join them.
  - In a random-restart search, each search process runs independently of the others.

Stochastic Beam Search

- **Problem:** quite often, all k states end up on same local hill in ***local beam search algorithm***
- **Idea:** choose k successors randomly, biased towards good ones

  ➔ **Stochastic Beam Search**

- Instead of choosing the best k from the pool of candidate successors, **stochastic beam search** chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value.

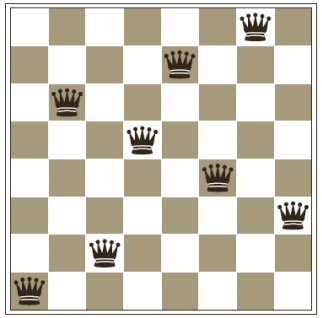
- **Stochastic beam search** bears some resemblance to the process of natural selection, whereby the "successors" (offspring) of a "state" (organism) populate the next generation according to its "value" (fitness).

Genetic Algorithms

- A **genetic algorithm (GA)** is a variant of ***stochastic beam search*** in which successor states are generated by combining two parent states rather than by modifying a single state.
- Like beam searches, **GAs** begin with a set of ***k randomly generated states***, called the **population**.
- Each state, or **individual**, is represented as a string over a finite alphabet—most commonly, a string of 0s and 1s.
  - An 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so it requires 8× log2 8=24 bits.
  - Alternatively, the state could be represented as 8 digits, each in the range from 1 to 8.
- Each state is rated by an ***objective function***, or (in GA terminology) the **fitness function**.
- Pairs of individuals are randomly selected for ***reproduction***.
- From each pair new offsprings (children) are generated using **crossover** operation.
  - A crossover point is chosen randomly from the positions in the string.
  - The offsprings are created by crossing over the parent strings at the crossover point.
- Each child is subject to random **mutation** with a small independent probability.

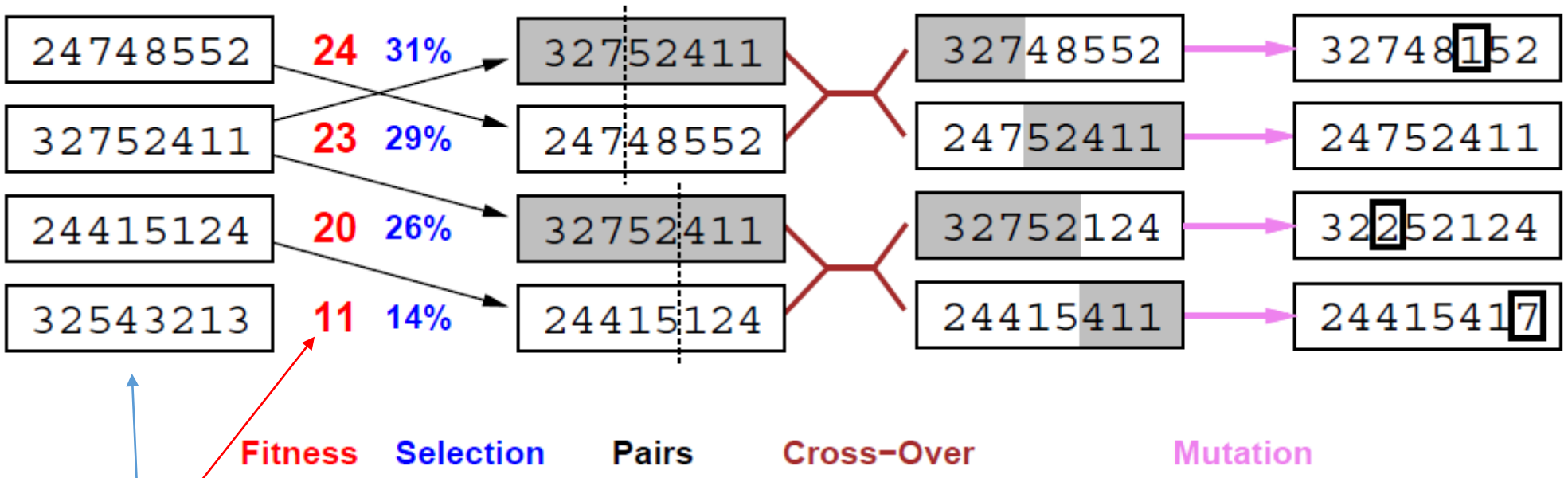
Genetic Algorithms

- A state can be represented using a 8 digit string.
- Each digit in the range from 1 to 8 to indicate the position of the queen in that column.



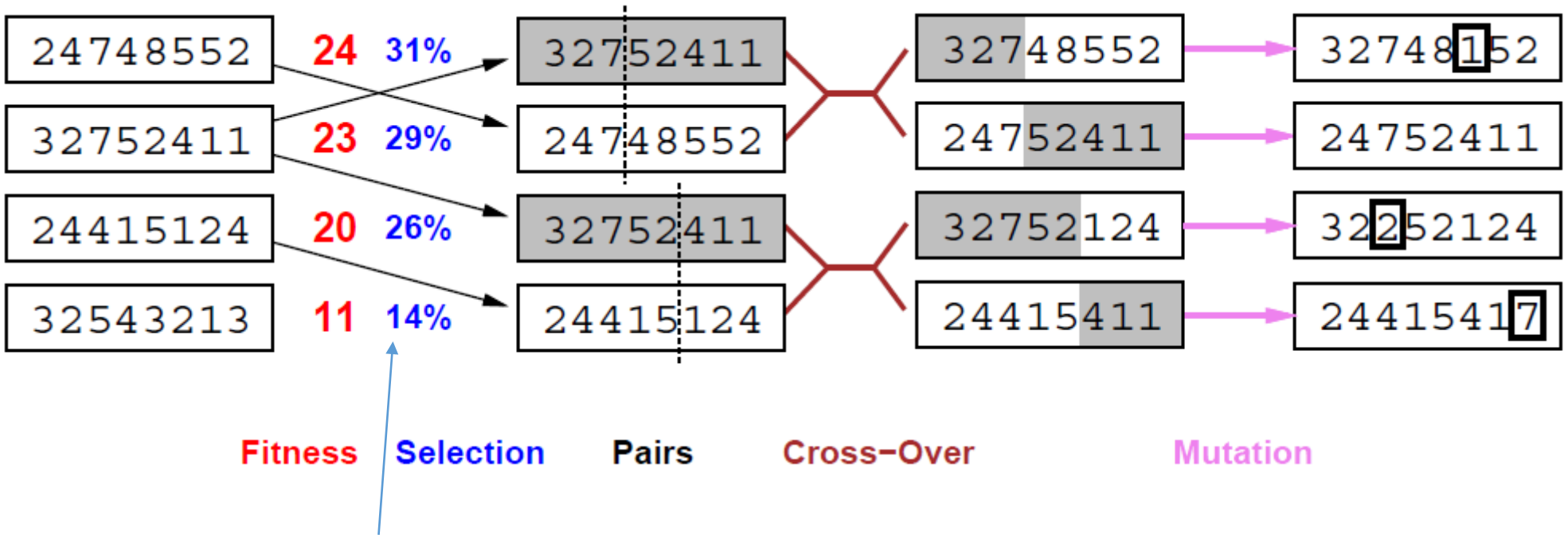
➔ 1 6 2 5 7 4 8 3

Genetic Algorithms



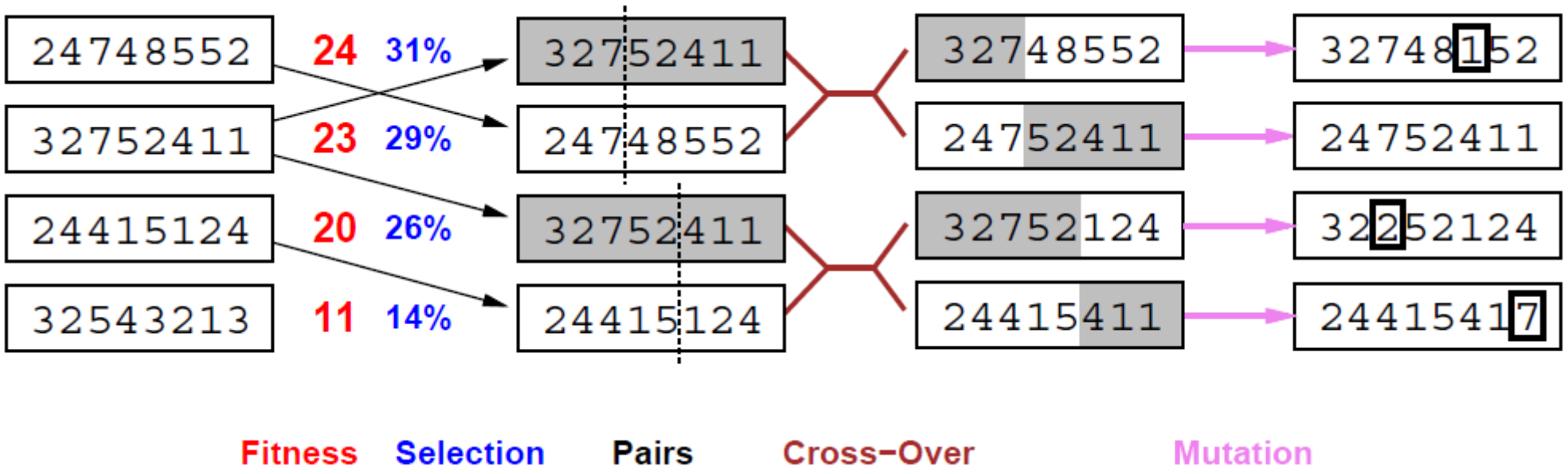
- The **initial population** has 4 states.
- A **fitness function** should return higher values for better states, so, for the 8-queens problem we use the number of non-attacking pairs of queens, which has a value of 28 for a solution.
  - The values of the four states are 24, 23, 20, and 11.

Genetic Algorithms



- The probability of being chosen for reproducing is directly proportional to the fitness score.
- Two pairs are selected at random for reproduction, in accordance with the probabilities.
  - Notice that one individual is selected twice and one not at all.

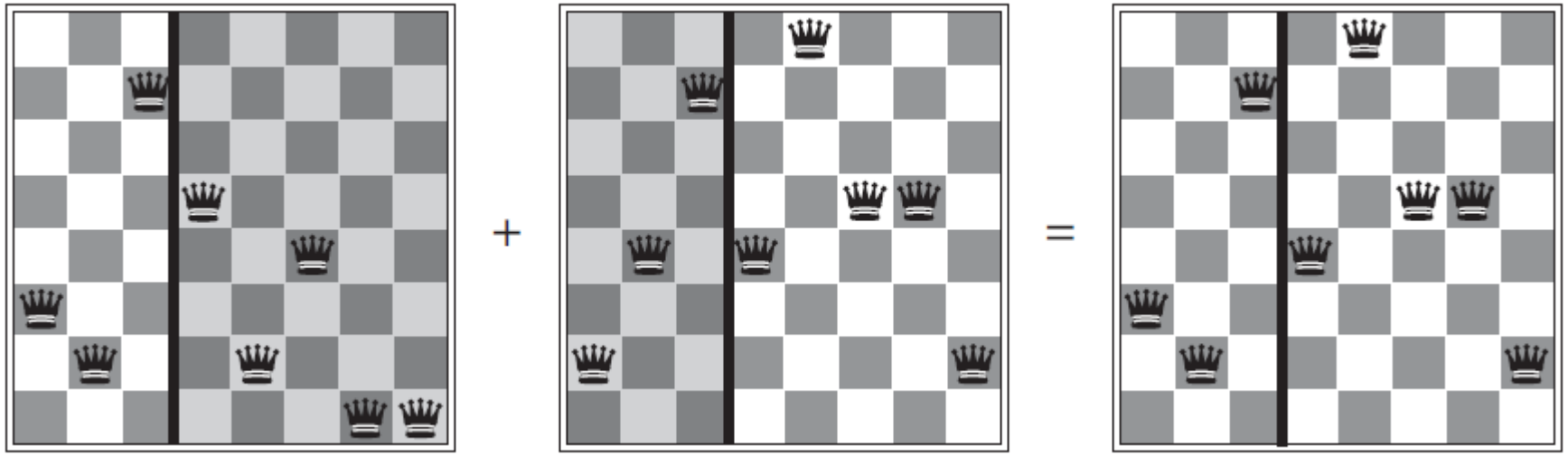
Genetic Algorithms



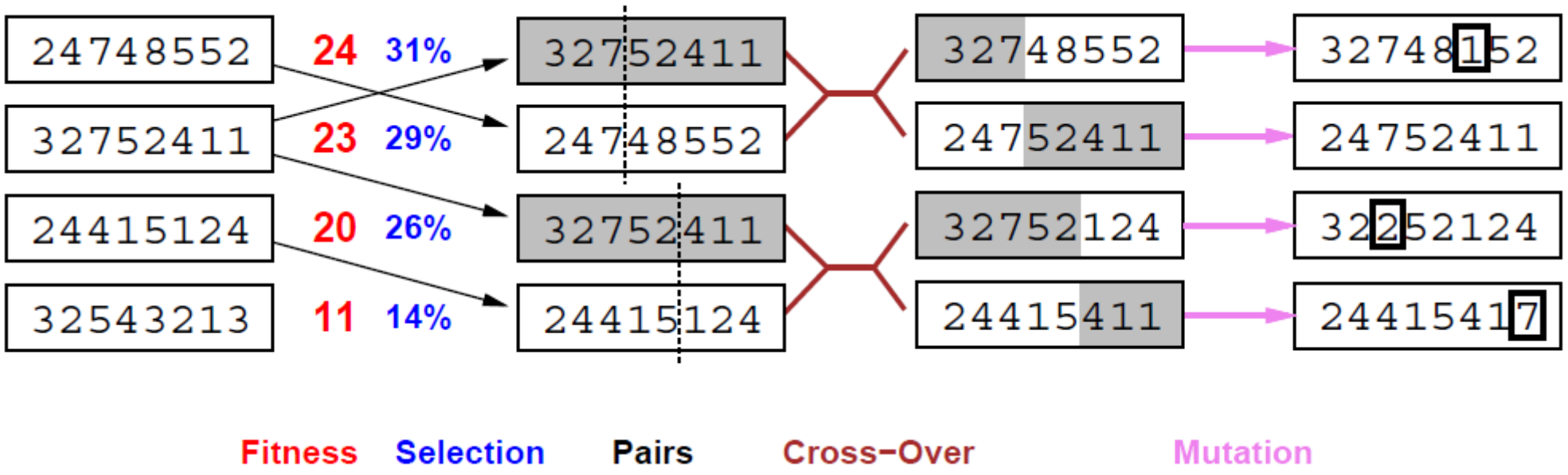
- The **crossover points** are after third digit in first pair and after fifth digit in second pair.
- The first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent, whereas the second child gets the first three digits from the second parent and the rest from the first parent.

Genetic Algorithms

- Crossover helps iff substrings are meaningful components.
- The shaded columns are lost in the crossover step and the unshaded columns are retained.



Genetic Algorithms



- One digit was **mutated** in the first, third, and fourth offspring.
- In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column.

Genetic Algorithms

A genetic algorithm: *each mating of two parents produces only one offspring, not two.*

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

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

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