323
ARTIFICIAL INTELLIGENCE & EXPERT SYSTEMS

Local Search Algorithms

Chapter 4

Dr. Zeynep ORMAN

Local search algorithms

- The search algorithms we have seen so far are systematic.
- This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path and which have not.
- When a goal is found, the path to that goal is also a part of a solution to the problem.
- In many problems, the path to the goal is irrelevant.
- If the path to the goal does not matter, we might consider a different class of algorithms – local search algorithms.

Local search algorithms

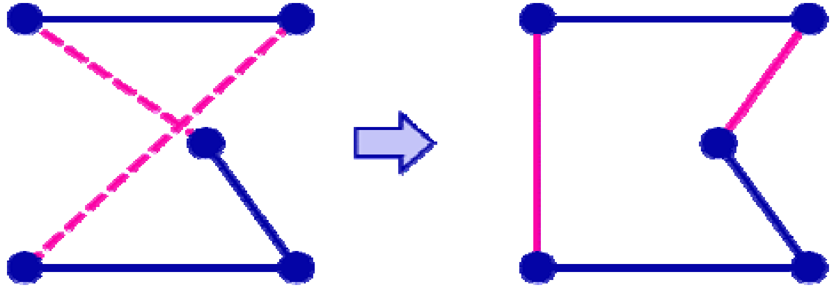
- Local search algorithms work by keeping in memory just one current state (or perhaps a few), moving around the state space based on purely local information.
- The paths followed by the search are not retained.
- Local search algorithms are not systematic,
- They have two advantages:

  (1) they use very little memory—usually a constant amount;

  (2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

Example: Travelling Salesperson Problem

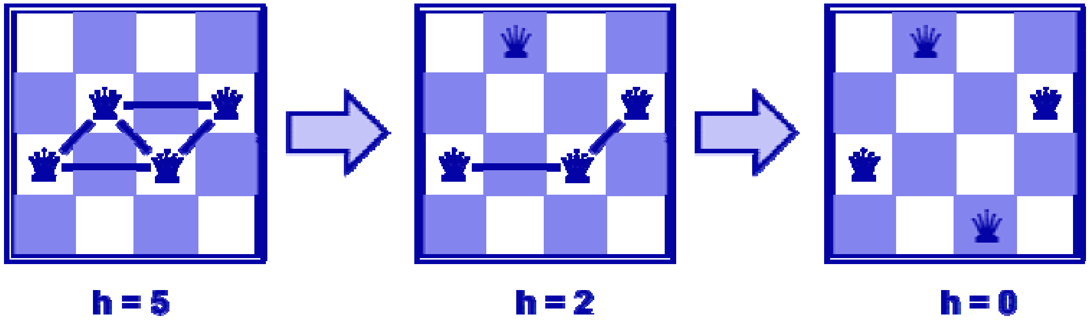
- In addition to finding goals, local search algorithms are useful for solving optimization problems.
  - the aim is to find the best state according to an objective function.
  - start with any complete tour, perform pairwise exchanges



  - variants of this approach get within 1% of optimal very quickly with thousands of cities

Example: *n*-queens

- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal
- What matters is the final configuration of queens, not the order in which they are added.
- Local search: start with all *n*, move a queen to reduce conflicts



- Almost always solves n-queens problems almost instantaneously for very large n, e.g., *n*=1 million

Generate-and-Test

Algorithm

1. Generate a possible solution.
2. Test to see if this is actually a solution.
3. Quit if a solution has been found. Otherwise, return to step 1.

Hill Climbing Search

- Hill Climbing search is simply a loop that continually moves in the direction of increasing value – uphill.
- It terminates when it reaches a "peak" where no neighbor has a higher value.
- The algorithm does not maintain a search tree, so the current node only records the state and objective function value.

Hill Climbing

- Searching for a goal state = Climbing to the top of a hill
- Generate-and-test + direction to move.
- Heuristic function to estimate how close a given state is to a goal state.

Simple Hill Climbing

Algorithm

1. Evaluate the initial state.

2. Loop until a solution is found or there are no new operators left to be applied:
   - Select and apply a new operator
   - Evaluate the new state:

     goal $\rightarrow$ quit

     better than current state $\rightarrow$ new current state

Hill-climbing search

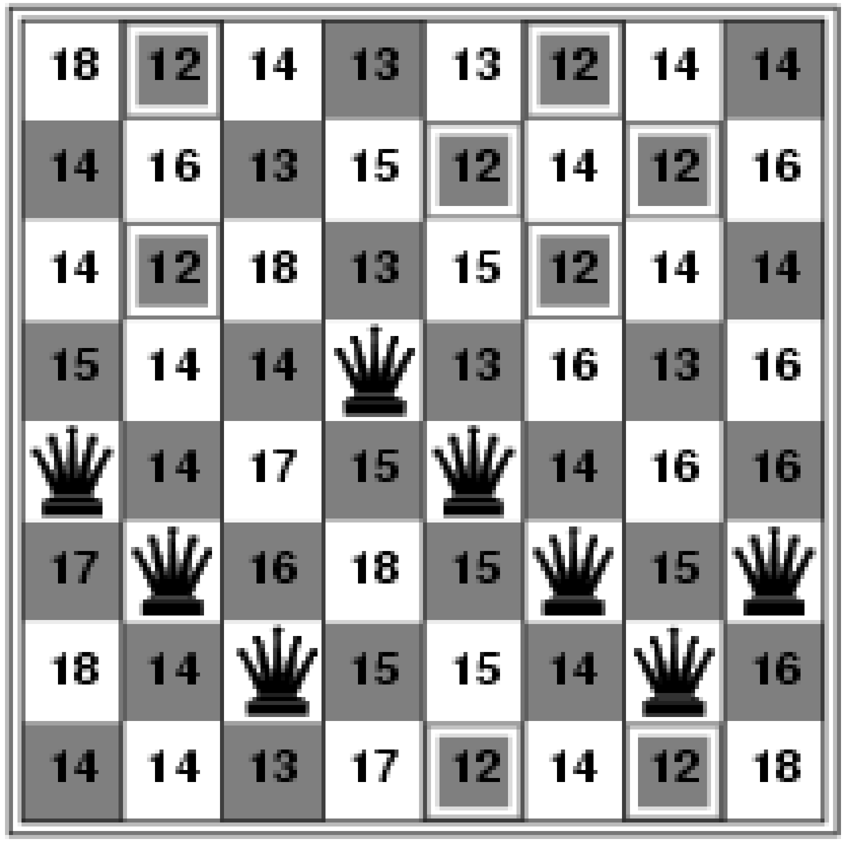
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

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

At each step the current node is replaced by the best neighbor → the neighbor with the highest value.

Hill-climbing search: 8-queens problem

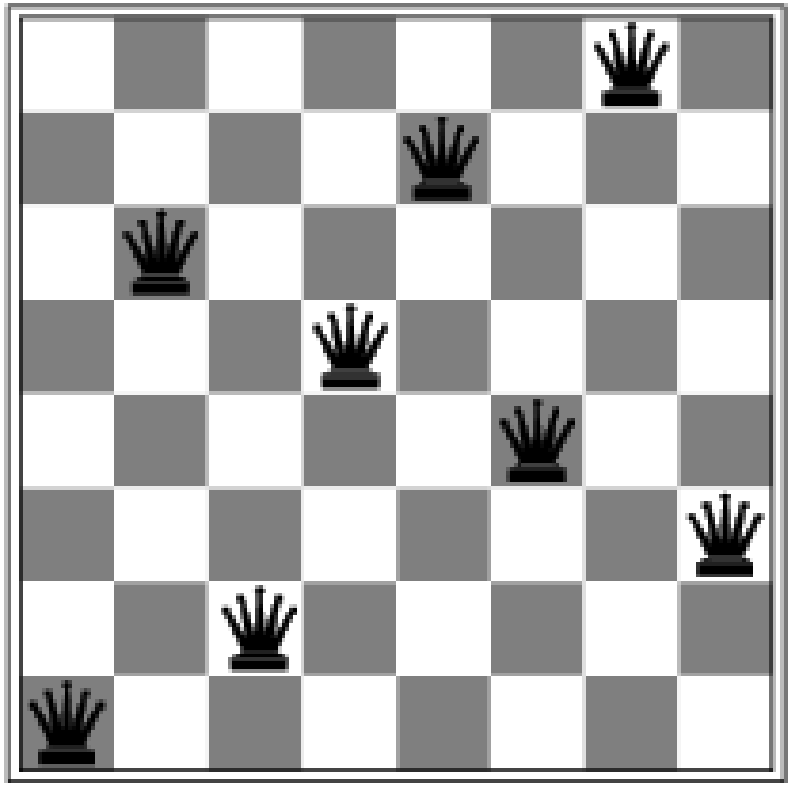
- Each state has 8 queens on the board, one per column.
- The successor function returns all possible states generated by moving a single queen to another square in the same column.
  - so, each state has 8x7=56 successors.
- ***h* = number of pairs of queens that are attacking each other, either directly or indirectly**
- The global minimum of this function is zero.

Hill-climbing search: 8-queens problem



- ***h* = *17*** for the above state
- The figure also shows the values of all its successors, with the best successors ***h=12.***

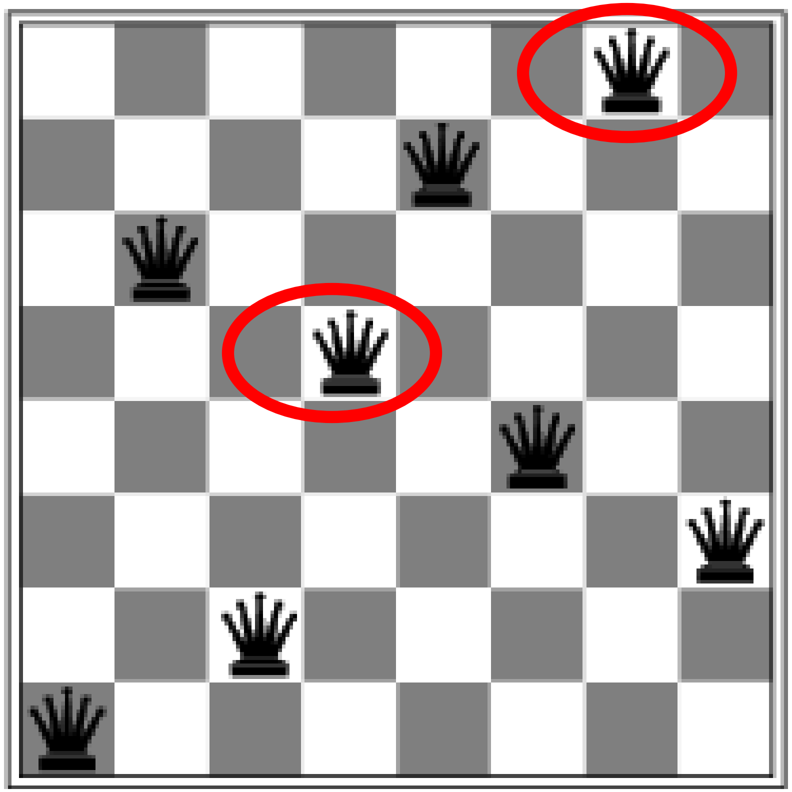
Hill-climbing search: 8-queens problem



Hill-climbing algorithms choose randomly among the set of best successors, if there is more than one.

- A local minimum with *h = 1* but every successor has a higher cost.

Hill-climbing search: 8-queens problem



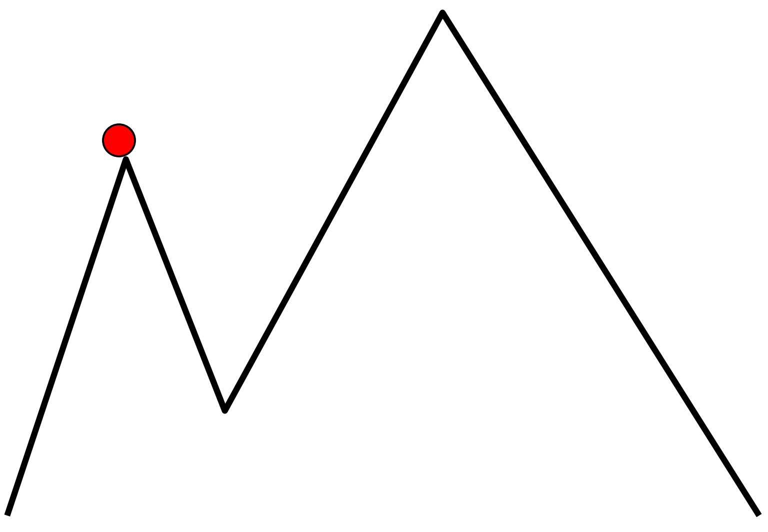
Hill-climbing algorithms choose randomly among the set of best successors, if there is more than one.

- A local minimum with *h = 1* but every successor has a higher cost.

Hill Climbing: Disadvantages

Local maximum

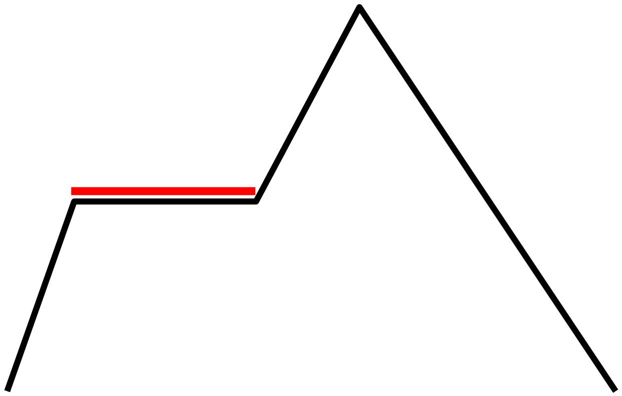
A state that is better than all of its neighbours, but not better than some other states far away.



Hill Climbing: Disadvantages

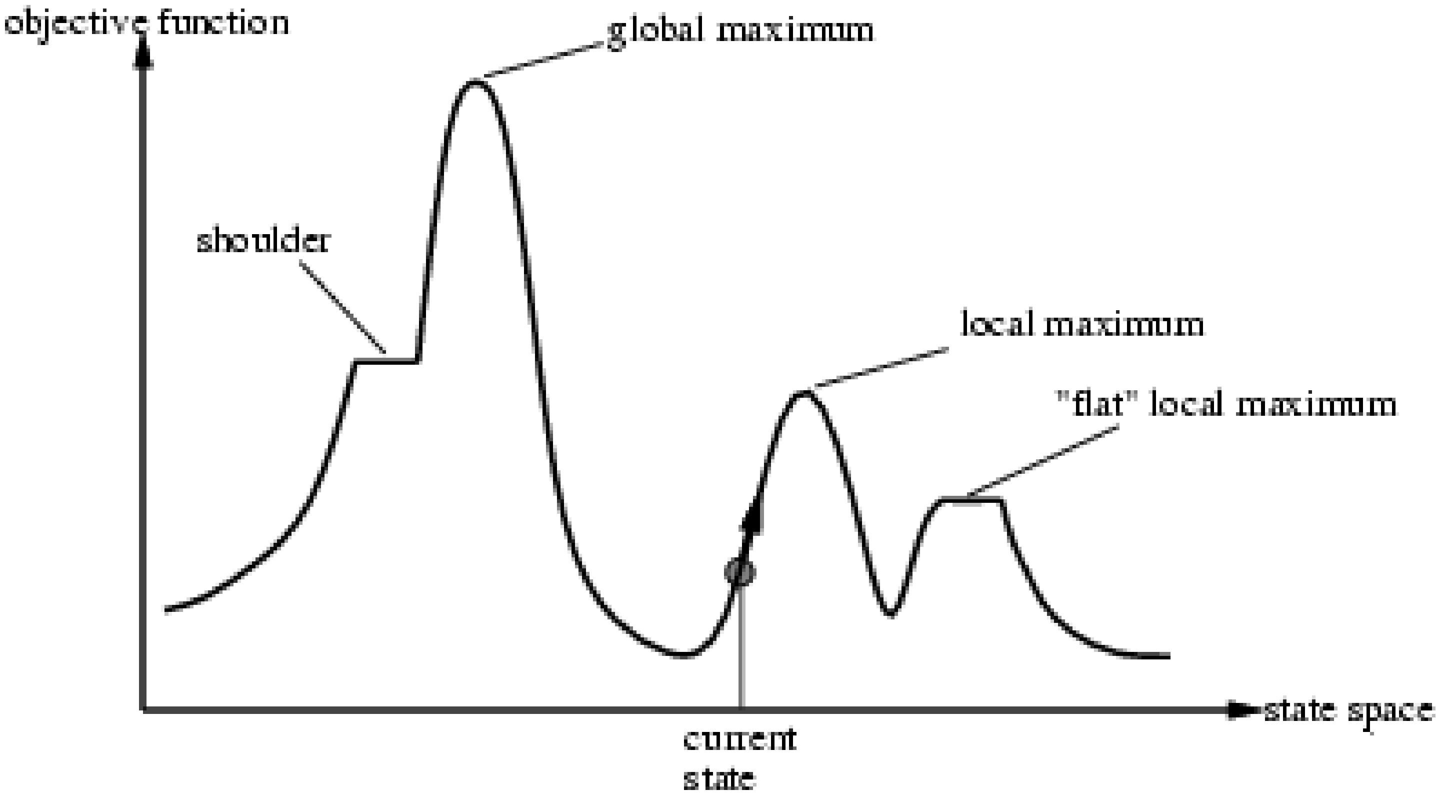
Plateau

A flat area of the search space in which all neighbouring states have the same value.



Hill-climbing search

- Hill-climbing search modifies the current state by trying to improve it, as shown by the arrow
- Problem: depending on initial state, can get stuck in local maxima



Hill Climbing: Disadvantages

Ways Out

- Backtrack to some earlier node and try going in a different direction.

- Make a big jump to try to get in a new section.

- Moving in several directions at once.

Stochastic Hill Climbing

- Generate successors randomly until one is better than the current state
- Good choice when each state has a very large number of successors

- Still, this is an incomplete algorithm
  - We may get stuck in a local maxima

Random Restart Hill Climbing

- Generate start states randomly
- Then proceed with hill climbing
- Will eventually generate a goal state as the initial state
- Hard problems typically have an large number of local maxima
  - This may be a decent definition of “difficult” as related to search strategy

- Sometimes worse must you get in order to find the better

-Yoda

Simulated annealing search

- **Idea:** escape local maxima by allowing some "bad" moves but gradually decrease their frequency.
- e.g. task of getting a ping-pong ball into the deepest crevice in a bumpy surface.
  - If we just let the ball roll, it will come to rest at a local minimum.
  - If we shake the surface, we can bounce the ball out of the local minimum.
- The trick is to shake just hard enough to bounce the ball out of local minima, but not hard enough get it from the global minimum.

Simulated annealing search

- The principle behind SA is similar to what happens when metals are cooled at a controlled rate
- The slowly decrease of temperature allows the atoms in the molten metal to line themselves up to form a regular crystalline structure that possesses a low density and a low energy.

Simulated annealing search

- One can prove: If *T* decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1.
- Widely used in VLSI layout, airline scheduling, etc

Simulated annealing search

- like hill-climbing identify the quality of the local improvements
- instead of picking the best move, pick one randomly- say the change in objective function is δ
- if δ is positive, then move to that state
- otherwise:
  - move to this state with probability proportional to δ
  - thus: worse moves (very large negative δ) are executed less often

Simulated annealing search

- There is always a chance of escaping from local maxima over time, make it less likely to accept locally bad moves
- (Can also make the size of the move random as well, i.e., allow "large" steps in state space)

Simulated annealing

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

input: *problem*, a problem

schedule, a mapping from time to temperature

local variables: *current***,** a node.

*next***,** a node.

*T***,** a "temperature" controlling the prob. of downward steps

current $\leftarrow$ MAKE-NODE(INITIAL-STATE[*problem*])

for t $\leftarrow$ **1 to** ∞ **do**

T $\leftarrow$ *schedule*[*t*]

if *T* = *0* **then return** *current*

next $\leftarrow$ a randomly selected successor of *current*

$\Delta E \leftarrow$ VALUE[*next*] - VALUE[*current*]

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

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

Temperature T

- high T: probability of “locally bad” move is higher
- low T: probability of “locally bad” move is lower
- typically, T is decreased as the algorithm runs longer
- i.e., there is a “temperature schedule”

Simulated Annealing in Practice

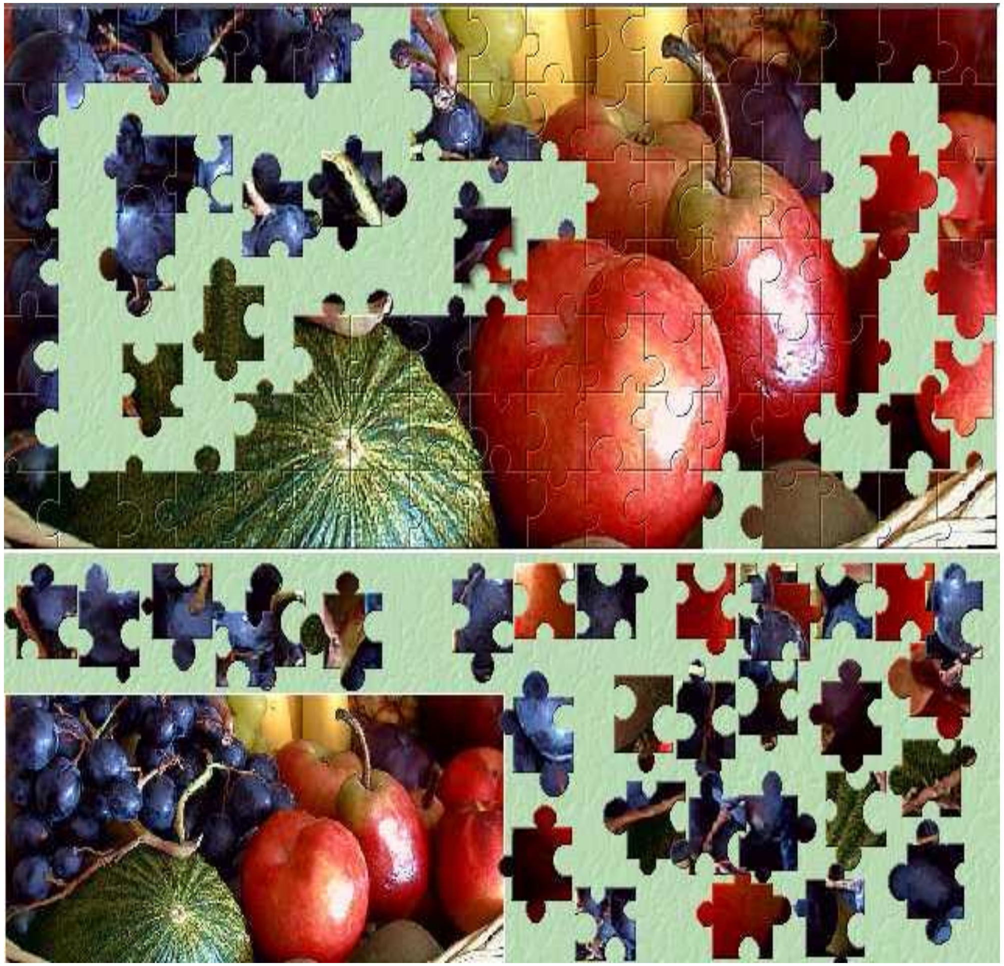
- method proposed in 1983 by IBM researchers for solving VLSI layout problems (Kirkpatrick et al, *Science*, 220:671-680, 1983).
  - theoretically will always find the global optimum

- Other applications: Traveling salesman, Graph partitioning, Graph coloring, Scheduling, Facility Layout, Image Processing, …

- useful for some problems, but can be very slow
  - slowness comes about because T must be decreased very gradually to retain optimality

Jigsaw puzzles – Intuitive usage of Simulated Annealing

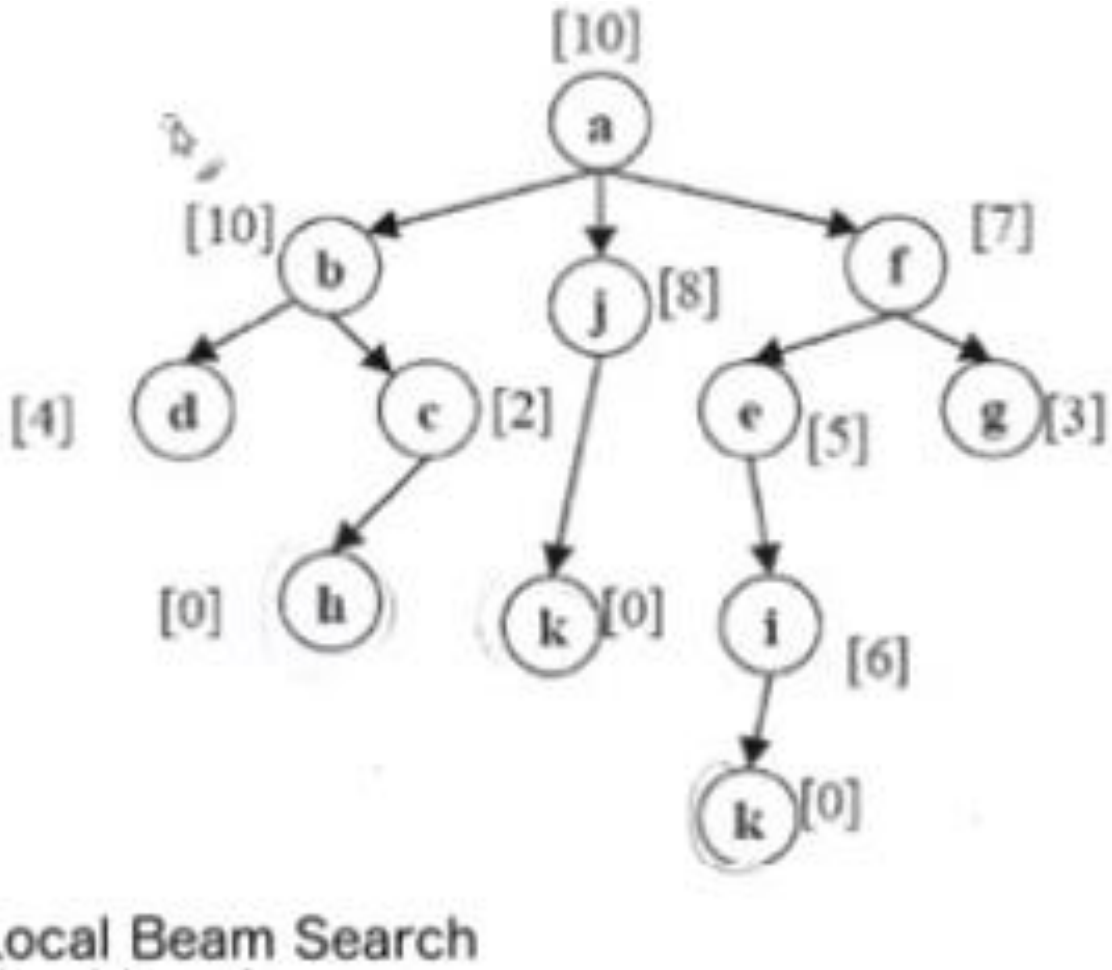
- Given a jigsaw puzzle such that one has to obtain the final shape using all pieces together.
- Starting with a random configuration, the human brain unconditionally chooses certain moves that tend to the solution.
- However, certain moves that may or may not lead to the solution are accepted or rejected with a certain small probability.
- The final shape is obtained as a result of a large number of iterations.



Local beam search

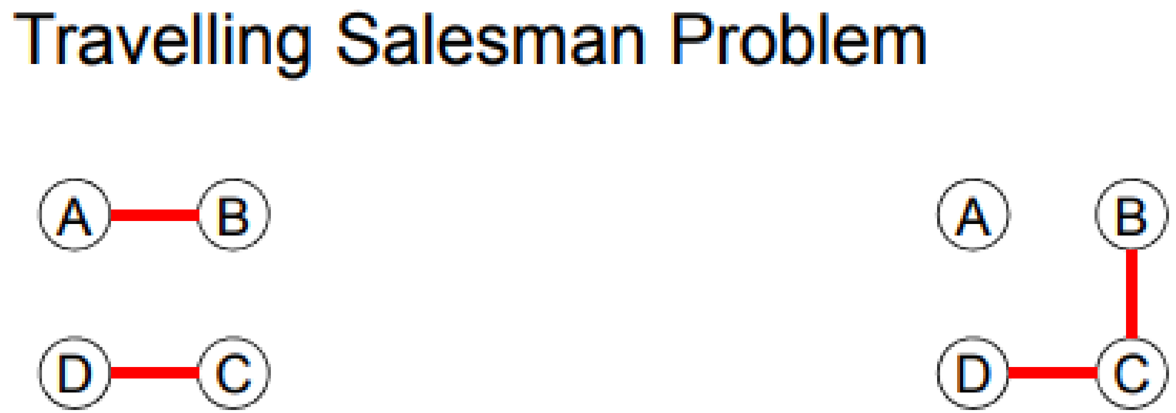
- Keep track of *k* states rather than just one
- Start with *k* randomly generated states
- At each iteration, all the successors of all *k* states are generated
- If any one is a goal state, stop; else select the *k* best successors from the complete list and repeat.

Local beam search



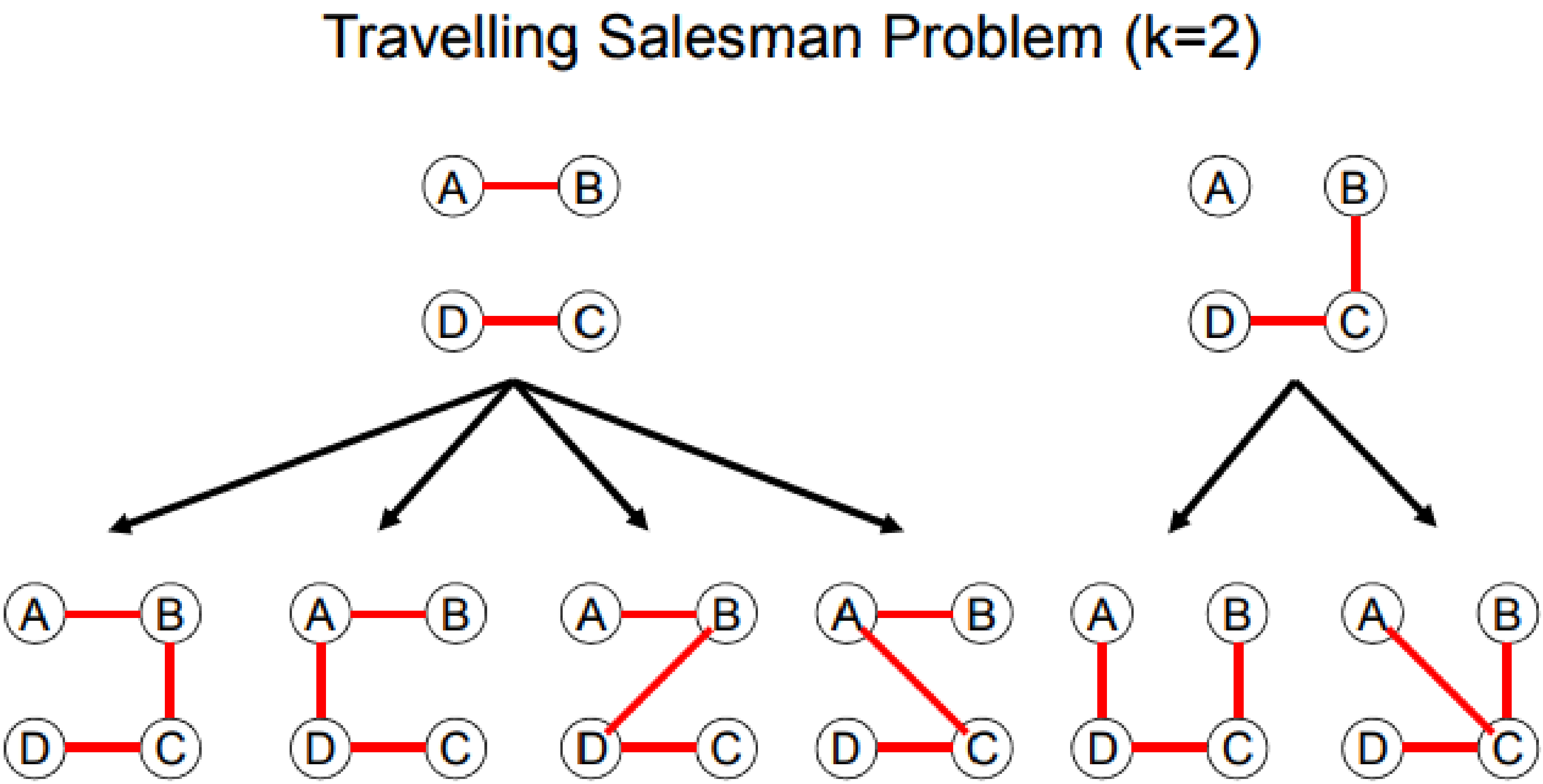
k is the beam width.

Local beam search



Keeps track of k states rather than just 1. k=2 in this example. Start with k randomly generated states.

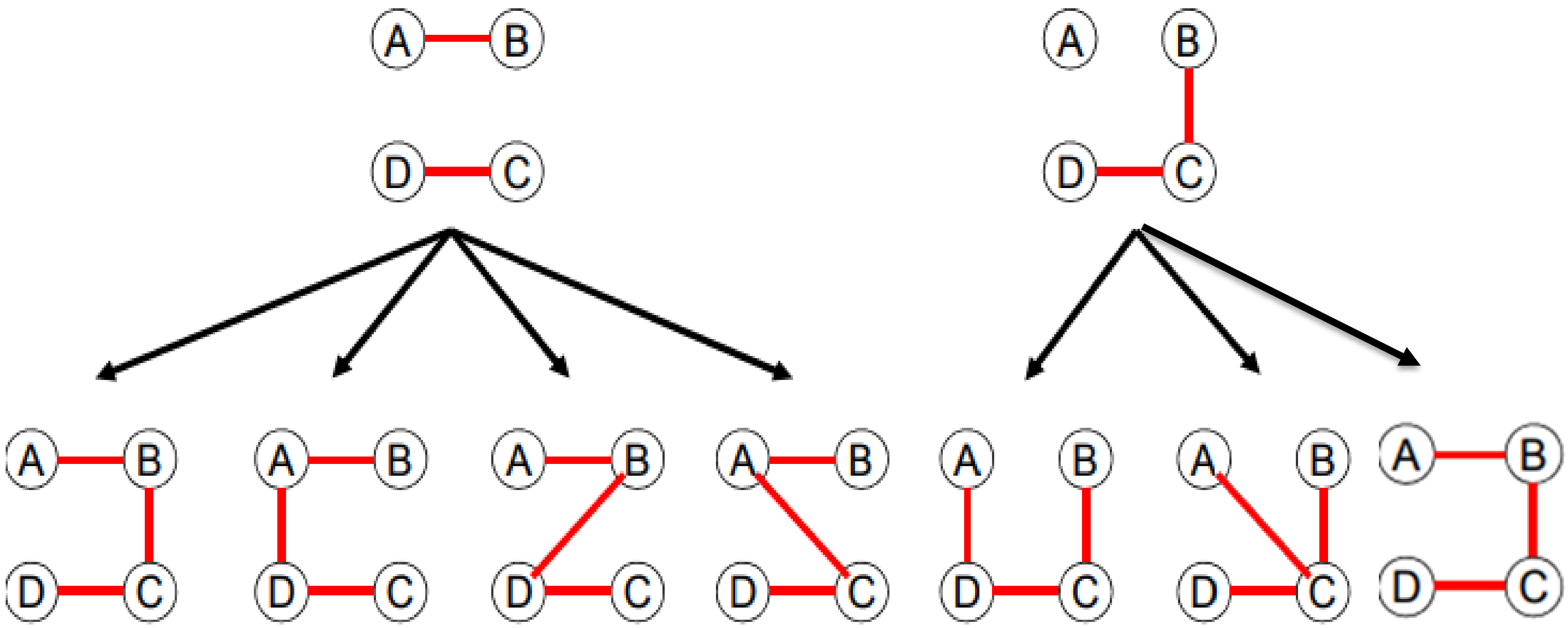
Local beam search



Generate all successors of all the k states

Local beam search

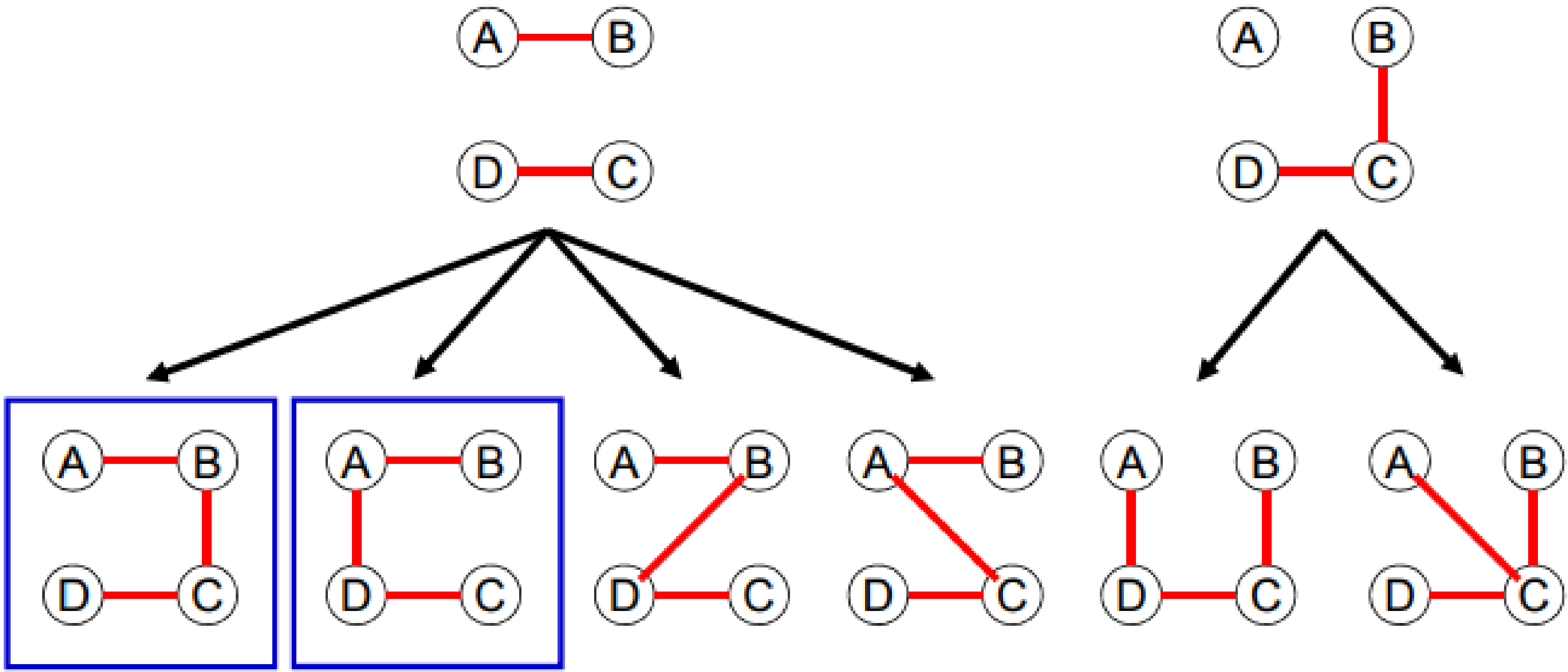
Travelling Salesman Problem (k=2)



None of these is a goal state so we continue

Local beam search

Travelling Salesman Problem (k=2)



Select the best k successors from the **complete** list

Local beam search

Travelling Salesman Problem (k=2)

Repeat the process until goal found

Local beam search

- How is this different from k random restarts in parallel?
- **Random-restart search**: each search runs independently of the others.
- **Local beam search**: useful information is passed among the k parallel search threads
  - Eg. One state generates good successors while the other k-1 states all generate bad successors, then the more promising states are expanded

Local beam search

- **Disadvantage**: all k states can become stuck in a small region of the state space
- To fix this, use **stochastic beam search**
- **Stochastic beam search**:
  - Doesn’t pick best k successors
  - Chooses k successors at random, with probability of choosing a given successor being an increasing function of its value

Genetic algorithms

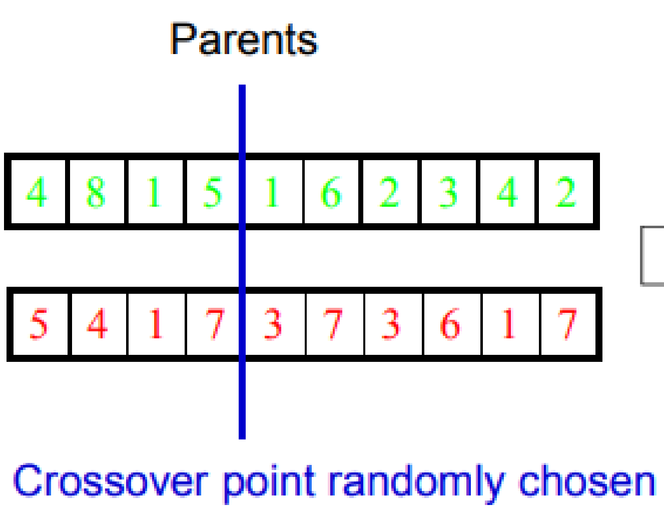
- Like natural selection in which an organism creates offspring according to its fitness for the environment.
- Essentially a variant of stochastic beam search that combines two parent states
- Over time, population contains individuals with high fitness

Genetic algorithms - definitions

- A successor state is generated by combining two parent states rather than modifying a single state.
- Start with *k* randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function). Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation

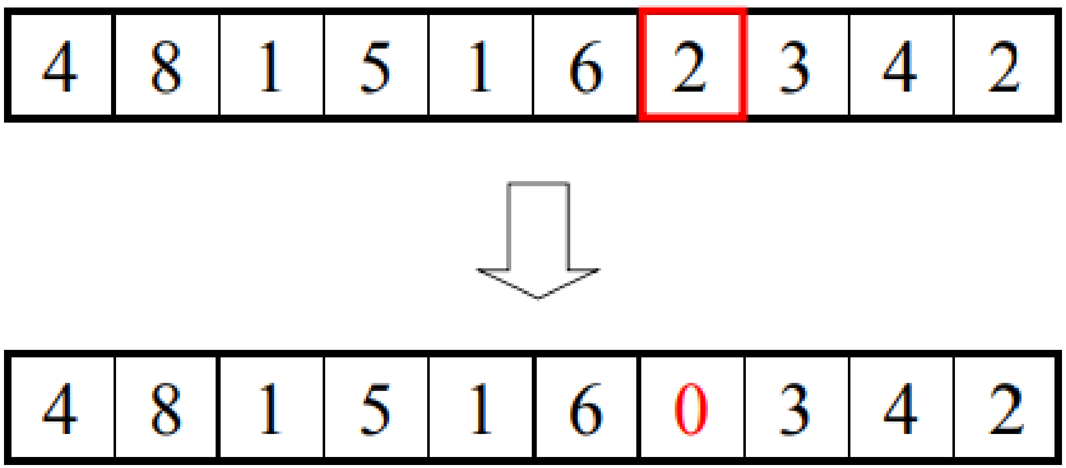
Genetic algorithms - definitions

- Selection: Pick two random individuals for reproduction
- Crossover: Mix the two parent strings at the crossover point



Genetic algorithms - definitions

- Mutation: randomly change a location in an individual’s string with a small independent probability



Randomness aids in avoiding small local extrema

Genetic algorithms - overview

Population = Initial population

Iterate until some individual is fit enough or enough time has elapsed:

    NewPopulation = Empty

    For 1 to size(Population)

        Select pair of parents (P_1,P_2) using Selection(P,Fitness Function)

        Child C = Crossover($P_{1,}$ P_2)

        With small random probability, Mutate(C)

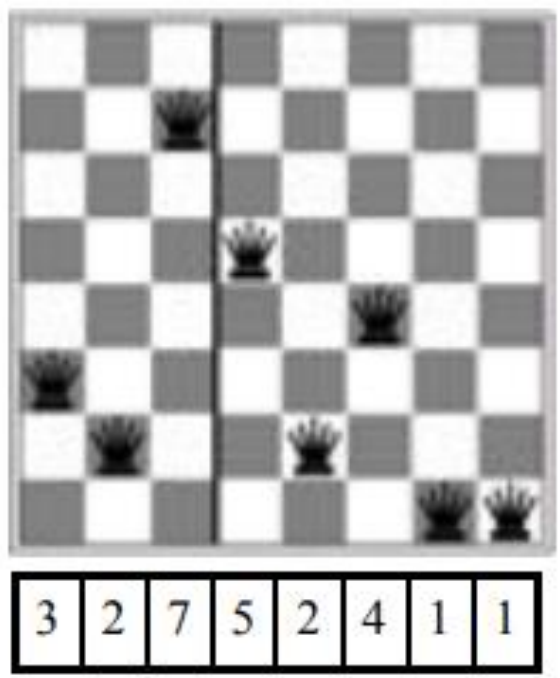
        Add C to NewPopulation

    Population = NewPopulation

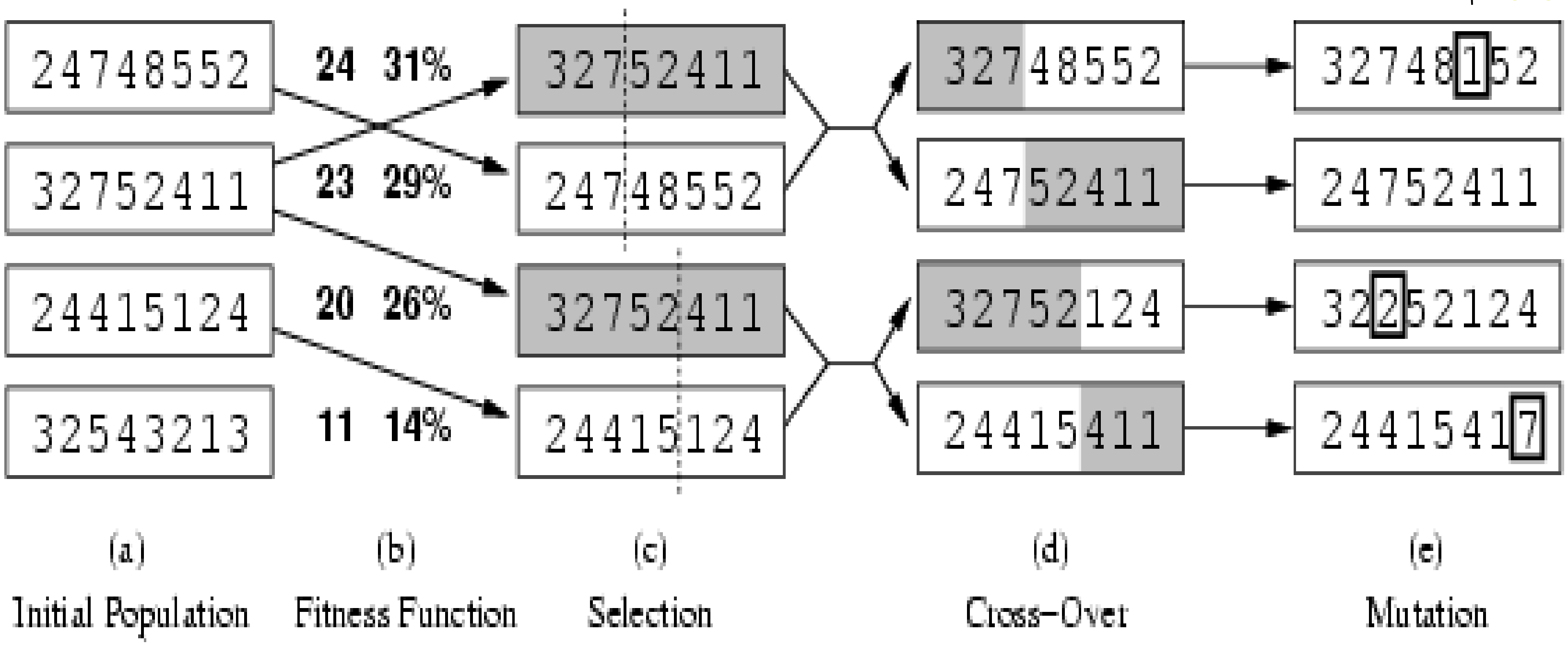
Return individual in Population with best Fitness Function

Example- 8 queens

- Fitness Function: number of nonattacking pairs of queens (28 is the value for the solution)
- Represent 8-queens state as an 8 digit string in which each digit represents position of queen



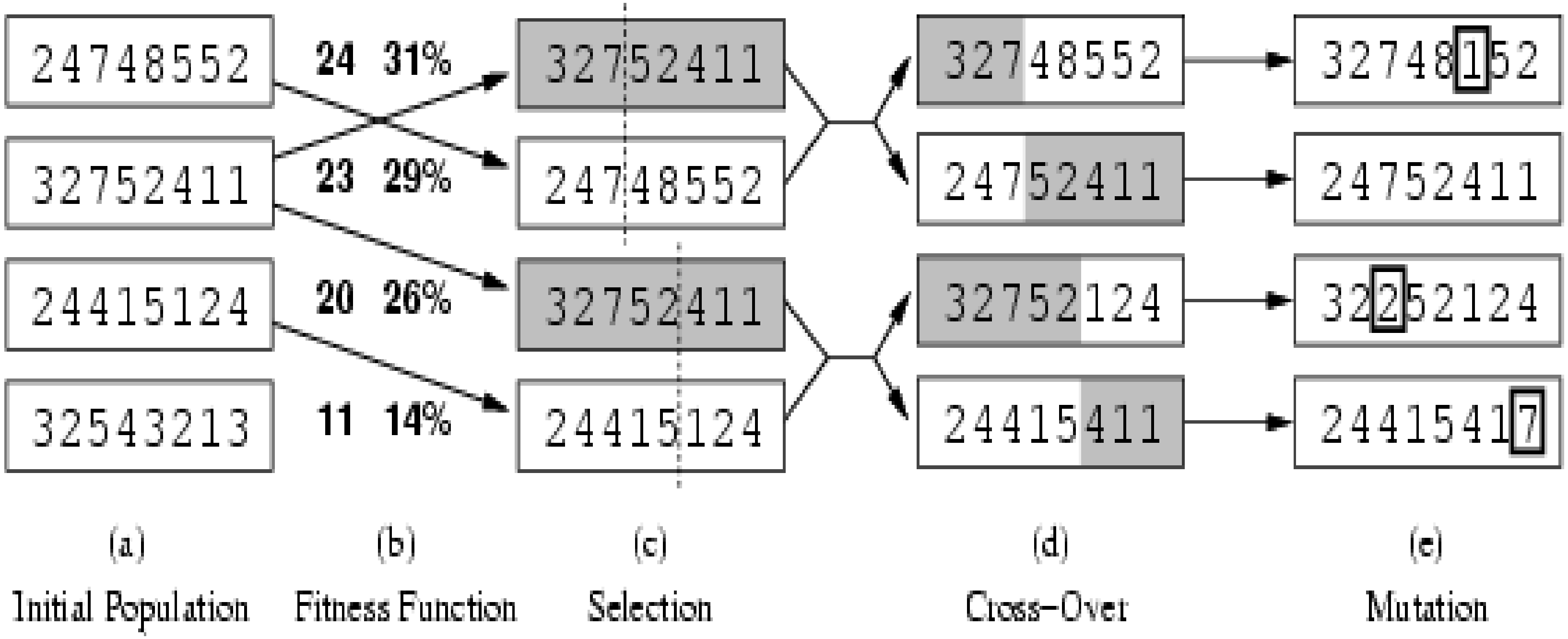
Example- 8 queens



(a) shows a population of four 8-digit strings representing 8-queens states, each range from 1 to 8.

The production of the next generation of states is shown in Figure (b) – (e).

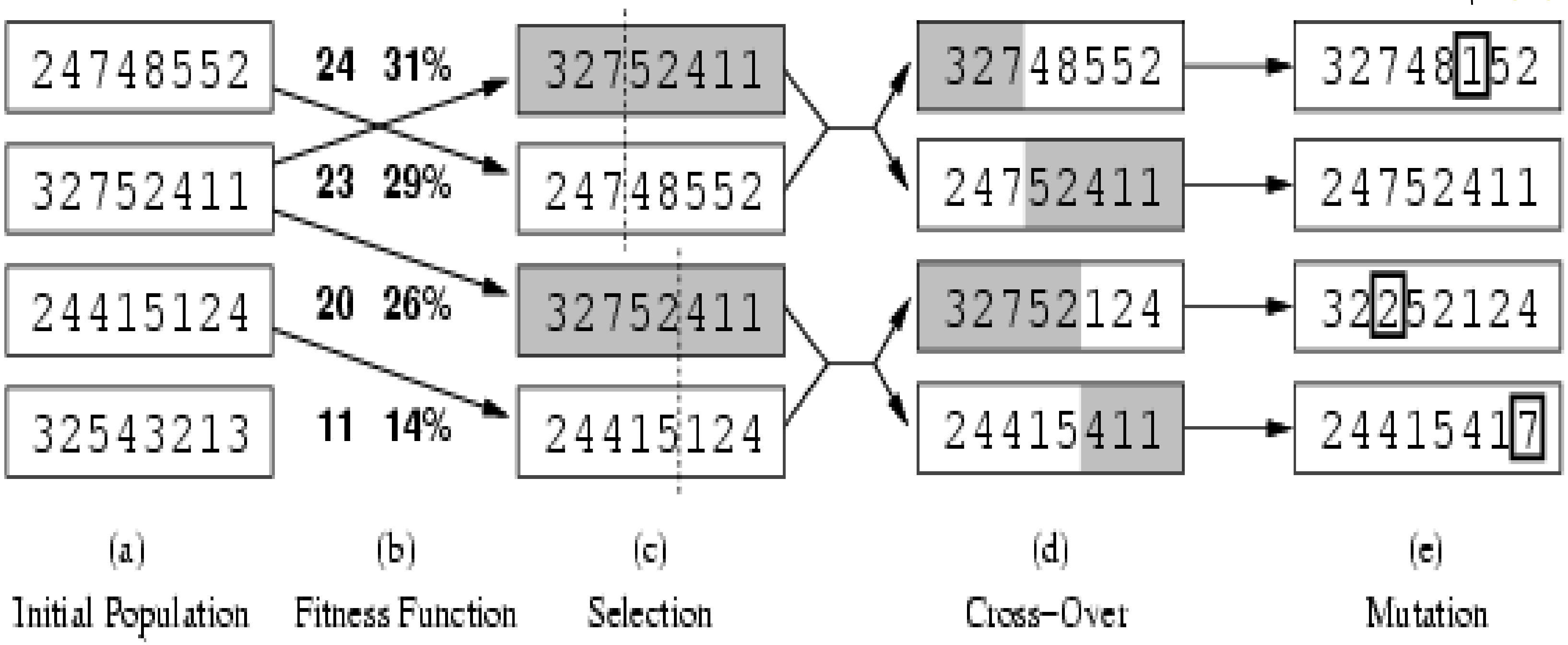
Example- 8 queens



In (b), each state is rated by the evaluation function – fitness function. We use the number of nonattacking pairs of queens – 28 for a solution.

The values of the four states are 24, 23, 20 and 11.

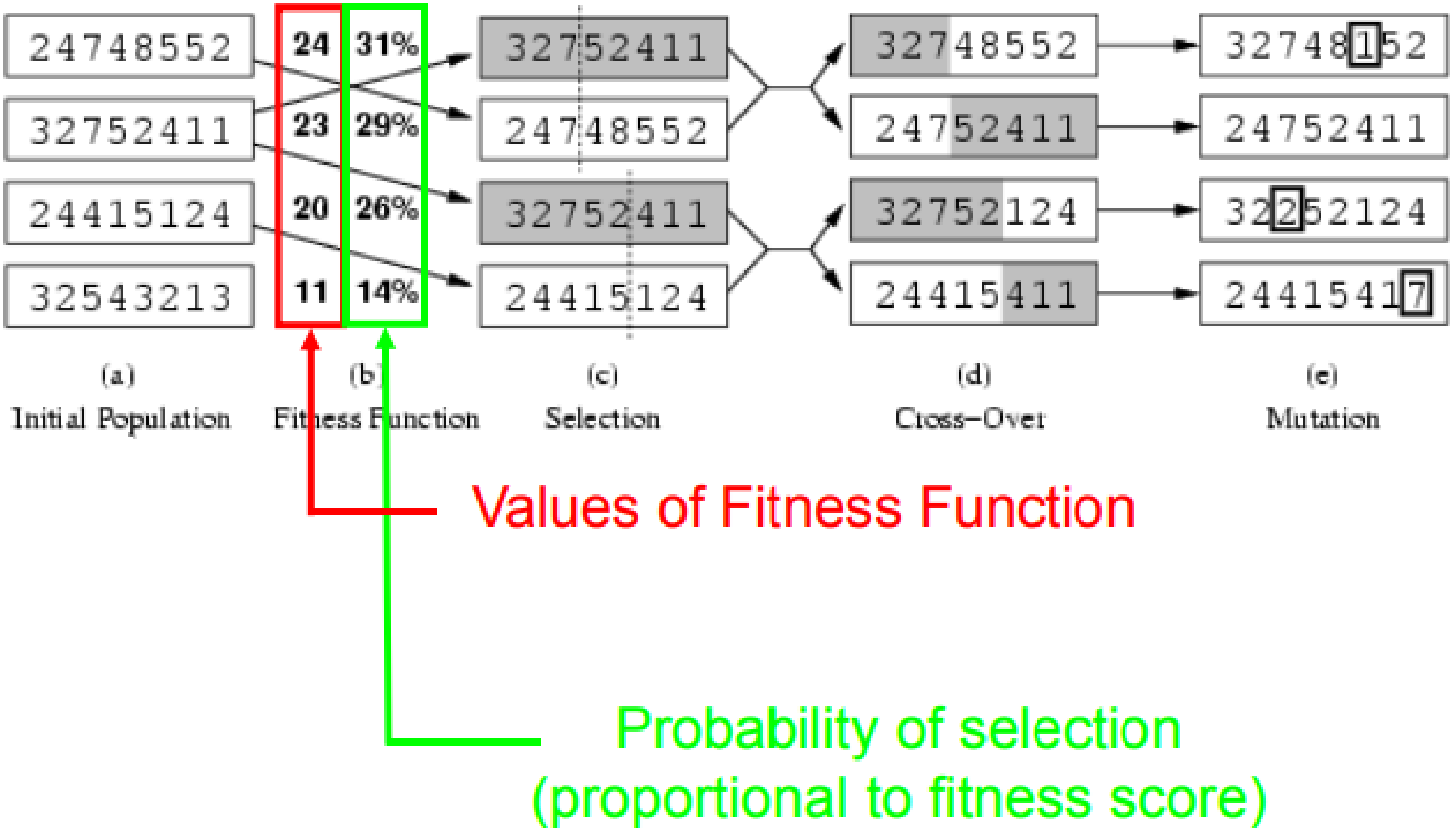
Example- 8 queens



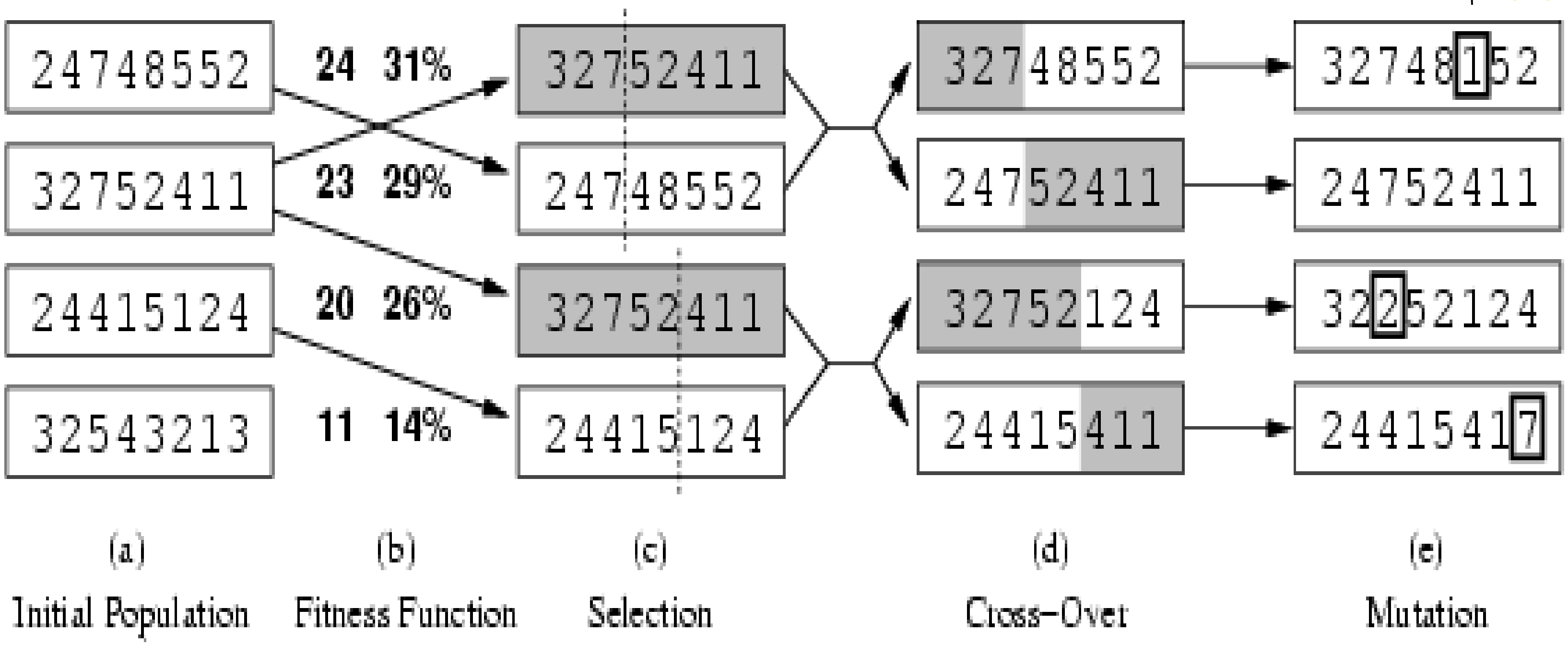
The probability of being chosen for reproducing is proportional to the fitness score :

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

Example- 8 queens (fitness function)



Example- 8 queens



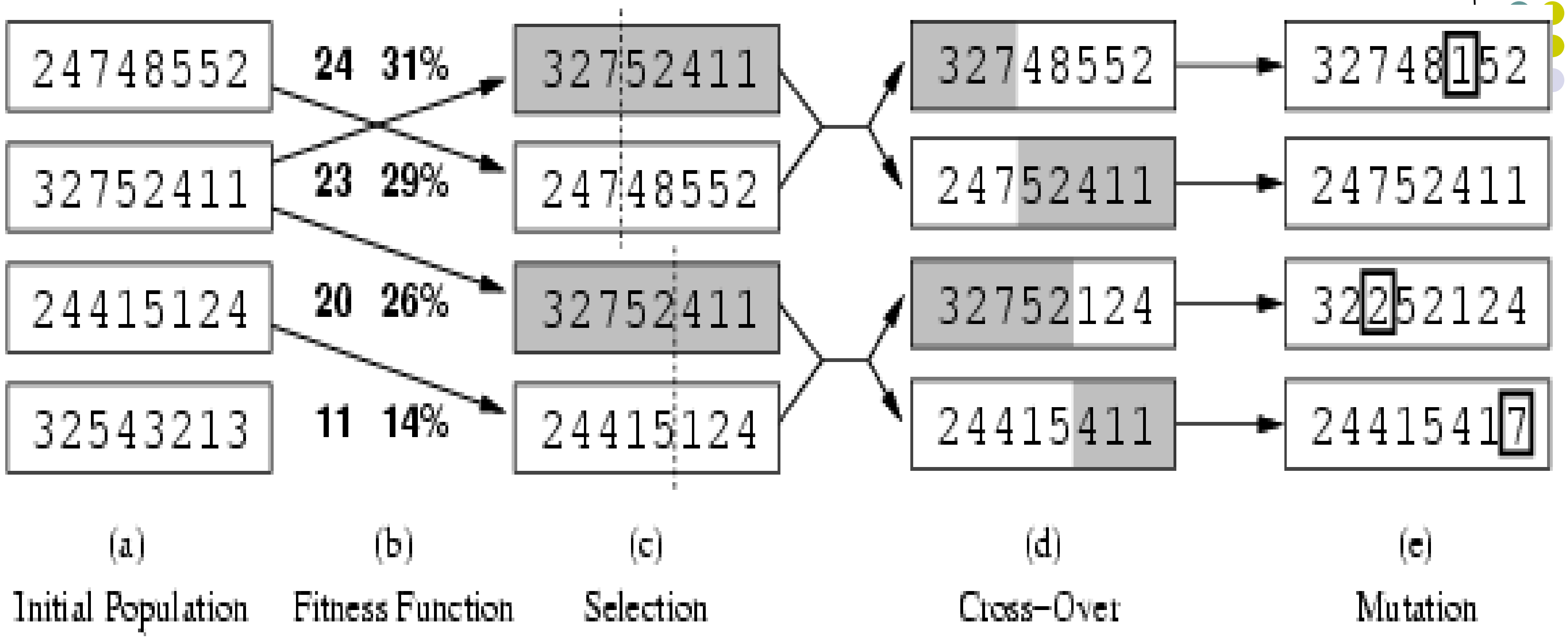
In (c), a random choice of two pairs is selected for reproduction, in accordance with the probabilities in (b).

- one individual is selected twice and one not at all.

Example- 8 queens (selection)

Notice 3 2 7 5 2 4 1 1 is selected twice while 3 2 5 4 3 2 1 3 is not selected at all

Example- 8 queens

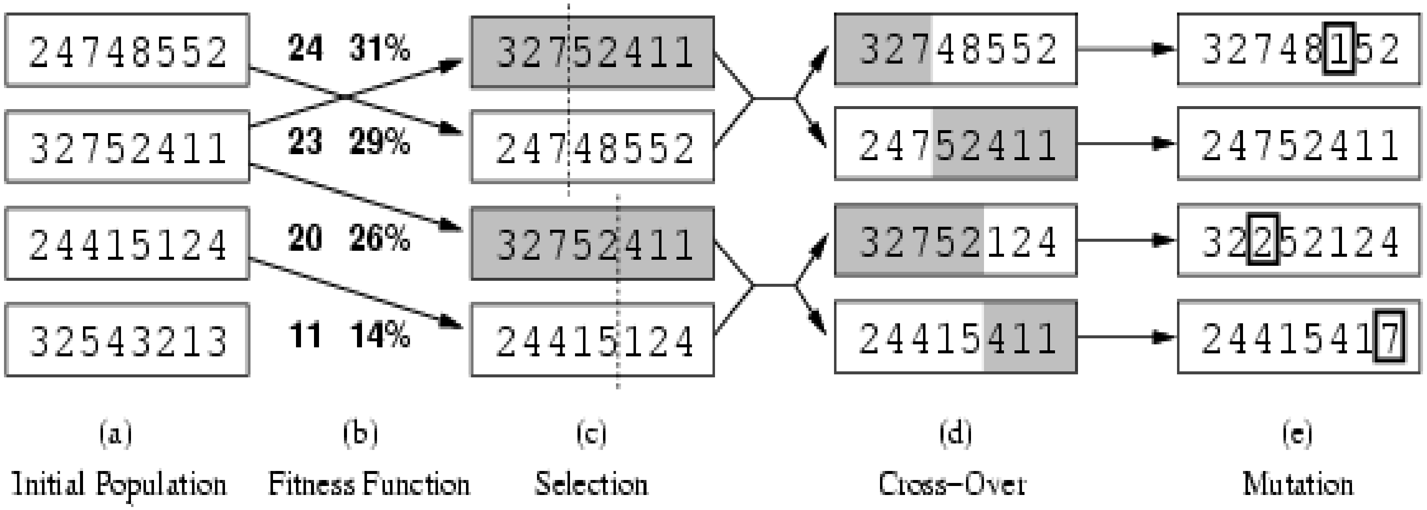


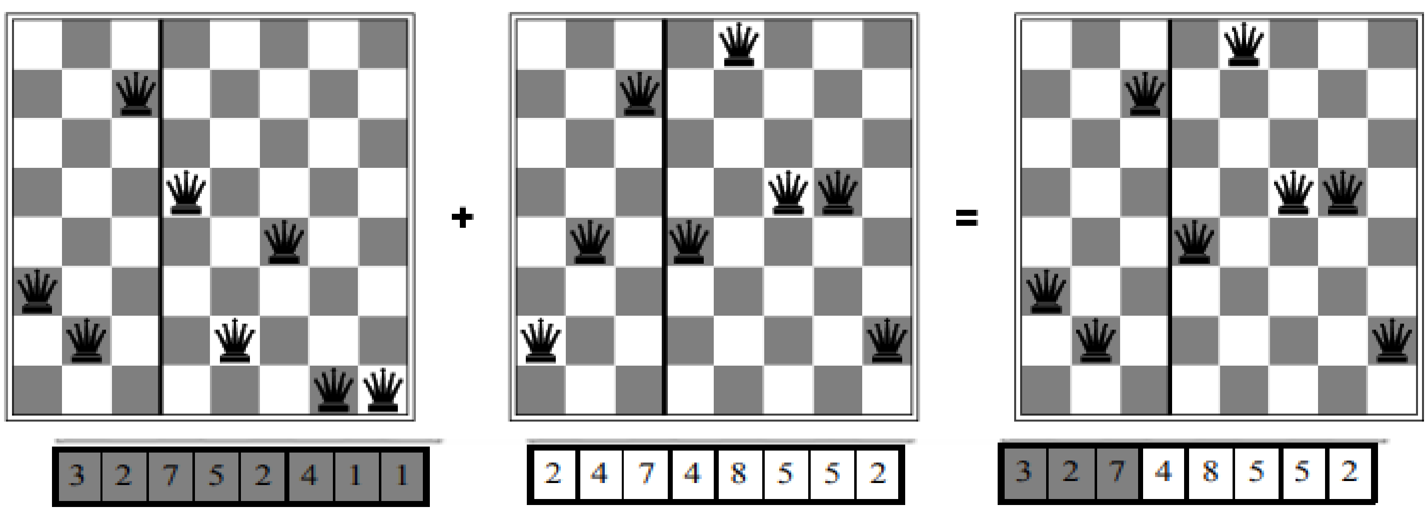
For each pair, a croossover point is randomly chosen.

In (d), the offsprings are created by crossing over the parent strings at the crossover point.

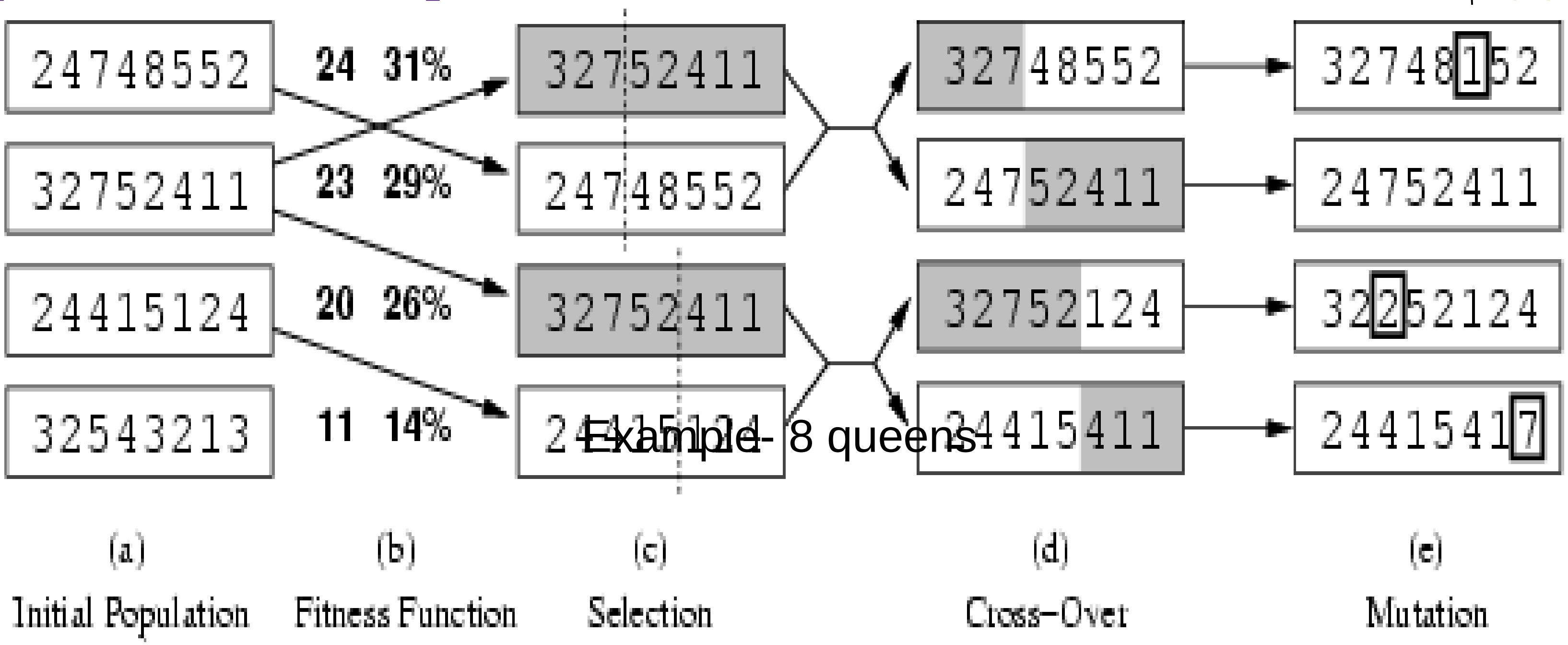
- e.g. the first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent.

Example- 8 queens (crossover)





Example- 8 queens (mutation)



Finally, in (e), each location is subject to random mutation with probability.

- e.g. in the 8-queens problem, choose a queen at random and move it to a random square in its column.

Hill Climbing Example

Consider the following search problem:

- the set of states $S = \{s_0, s_1, s_2, s_3, s_4, s_5\}$
- successors of s_0 are $\{s_0, s_1, s_2\}$
- successors of s_1 are $\{s_1, s_2, s_3\}$
- successors of s_2 are $\{s_2, s_3\}$
- successors of s_3 are $\{s_0, s_3, s_4\}$
- successors of s_4 are $\{s_4, s_5\}$
- successors of s_5 are $\{s_2, s_3, s_5\}$
- objective function f is as follows: $f(s_0) = 0$, $f(s_1) = 3$, $f(s_2) = 2$, $f(s_3) = 4$, $f(s_4) = 1$, and $f(s_5) = 5$.

Hill Climbing Example

- Trace hill climbing search starting in state s0 (indicate which state is considered at each iteration, and which solution is returned when the search terminates).
- Does it return the optimal solution?
- Which (if any) states are local maxima and global maxima?

Problem Example

Problem Example

Consider a GA with chromosomes consisting of six genes $x_i = abcdef$, and each gene is a number between 0 and 9. Suppose we have the following population of four chromosomes:

$$x_1 = 4\,3\,5\,2\,1\,6 \qquad x_2 = 1\,7\,3\,9\,6\,5$$

$$x_3 = 2\,4\,8\,0\,1\,2 \qquad x_4 = 9\,0\,8\,1\,2\,3$$

and let the fitness function be $f(x) = (a + c + e) - (b + d + f)$.

1. Sort the chromosomes by their fitness
2. Do one-point crossover in the middle between the 1st and 2nd fittest, and two-points crossover (points 2, 4) for the 2nd and 3rd.
3. Calculate the fitness of all the offspring