

# Artificial Intelligence

## CS-401



### Chapter # 04

# Local Search & Optimization Problems

**Dr. Hafeez Ur Rehman**

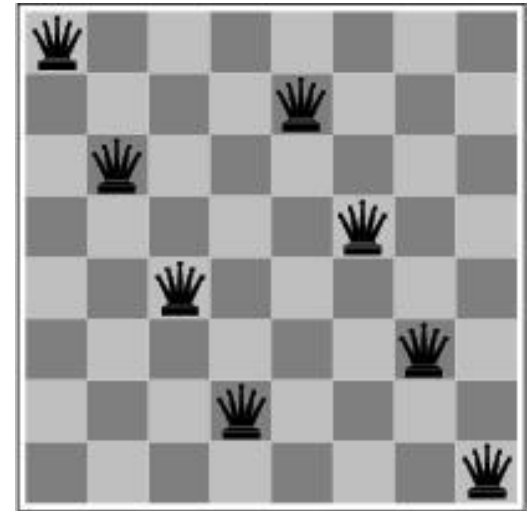
**(Email: [hafeez.urrehman@nu.edu.pk](mailto:hafeez.urrehman@nu.edu.pk))**

# Outline

- Local search techniques and optimization
  - Hill-climbing
  - Simulated annealing
  - Local Beam Search
  - Genetic algorithms

# Local search and optimization

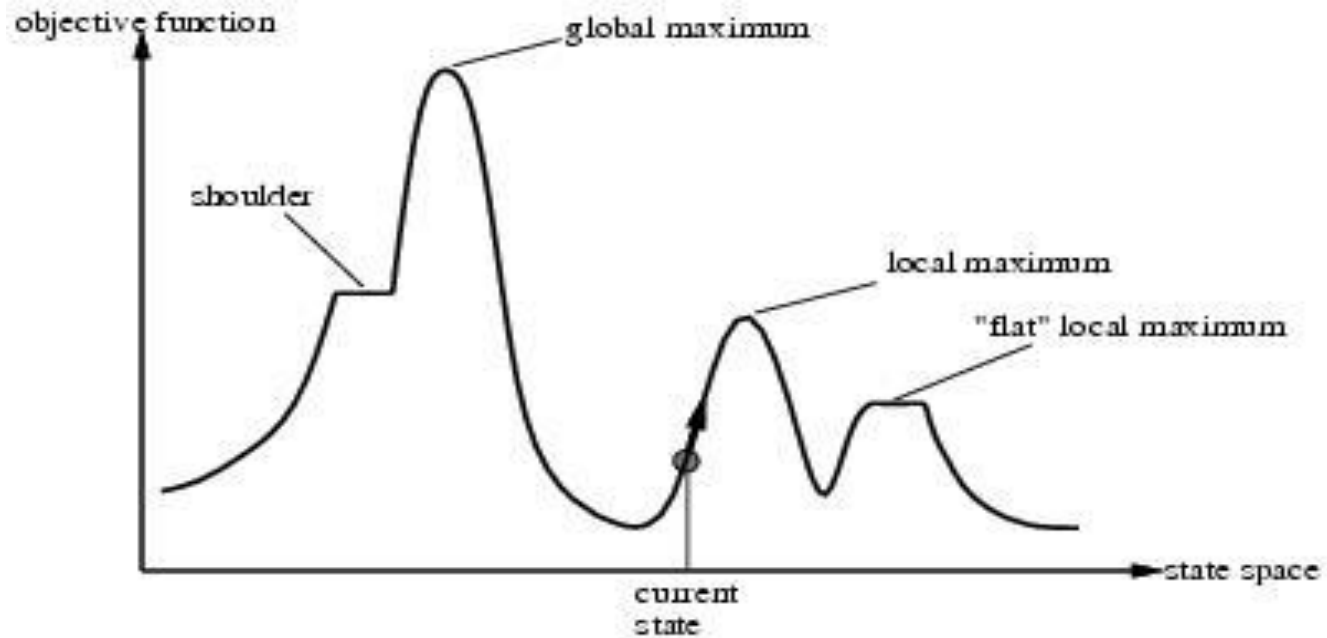
- Previously: **systematic exploration** of search space.
  - Path to goal is solution to problem
- YET, for some problems path is irrelevant.
  - E.g 8-queens
  - Factory Floor Layout
  - Automatic programming
  - Integrated circuit design
  - Vehicle routing
- For such problems different algorithms can be used
  - Local search



# Local search and optimization

- Local search
  - Keep track of single current state
  - Move only to neighboring states
  - Ignore paths
- Advantages:
  - Use very little memory
  - Can often find reasonable solutions in large or infinite (continuous) state spaces.
- “Pure optimization” problems
  - All states have an objective function
  - Goal is to find **state with max (or min)** objective value
  - Some problems do not quite fit into path-cost/goal-state formulation e.g. nature provides **reproductive fitness**, Darwin evolution seems to optimize it.
  - Local search can do quite well on these problems.

# “Landscape” of search



# Hill-climbing search algorithm (1)

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

**input:** *problem*, a problem

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

*neighbor*, a node.

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

**loop do**

*neighbor*  $\leftarrow$  a highest valued successor of *current*

**if** VALUE [*neighbor*]  $\leq$  VALUE[*current*]

**then return** STATE[*current*]

*current*  $\leftarrow$  *neighbor*

# Hill-climbing search

- “a loop that continuously moves in the direction of increasing value”
  - terminates when a peak is reached
  - Aka greedy local search
- Value can be either
  - Objective function value
  - Heuristic function value (minimized)
- Hill climbing **does not look ahead** of the immediate neighbors of the current state.
- Can **randomly** choose among the set of best successors, if multiple have the best value
- Characterized as “**trying to find the top of Mount Everest while in a thick fog**”

# Hill-climbing example

- 8-queens problem, complete-state formulation
  - All 8 queens on the board in some configuration
- Successor function:
  - move a single queen to another square in the same column.
- Example of a heuristic function  $h(n)$ :
  - the number of pairs of queens that are attacking each other (directly or indirectly)
  - (so we want to minimize this)



## Hill-climbing example

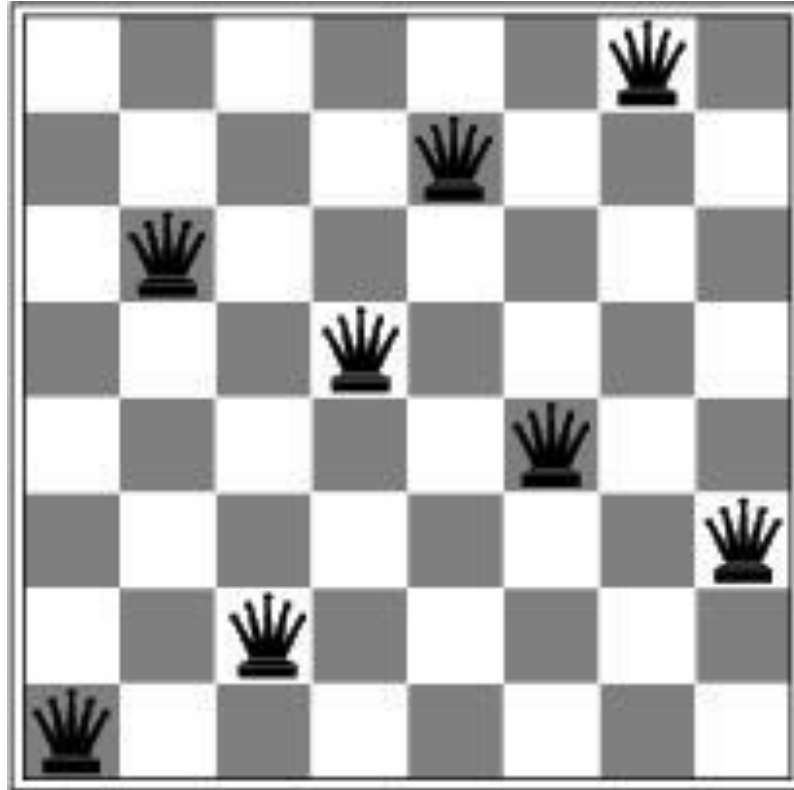
18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♙	13	16	13	16
♙	14	17	15	♙	14	16	16
17	♙	16	18	15	♙	15	♙
18	14	♙	15	15	14	♙	16
14	14	13	17	12	14	12	18

Current state:  $h=17$

$h$  is the number of queens attacking each other.

Shown is the  $h$ -value for each possible successor in each column

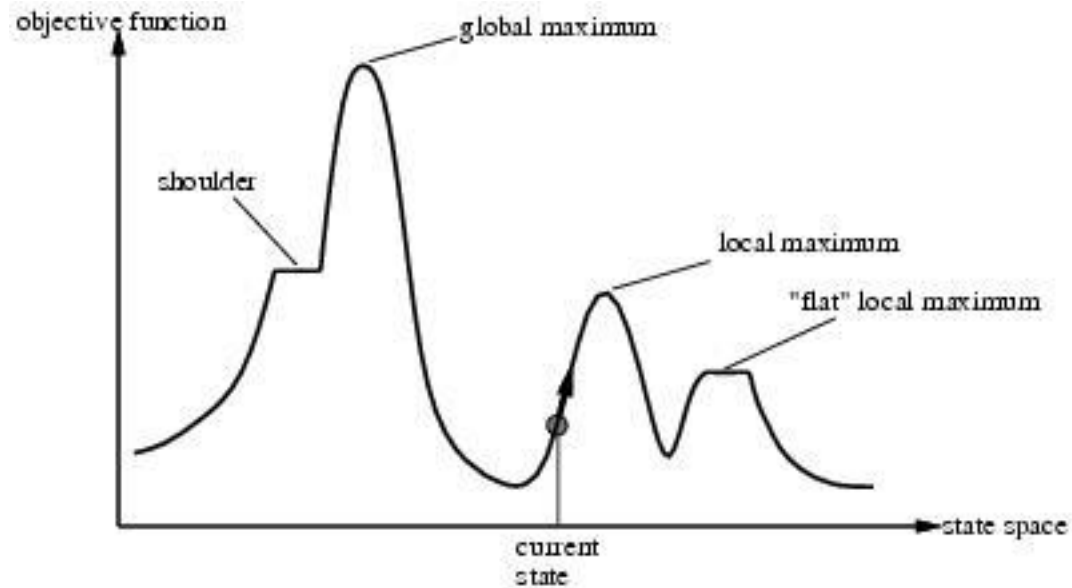
## A local minimum for 8-queens



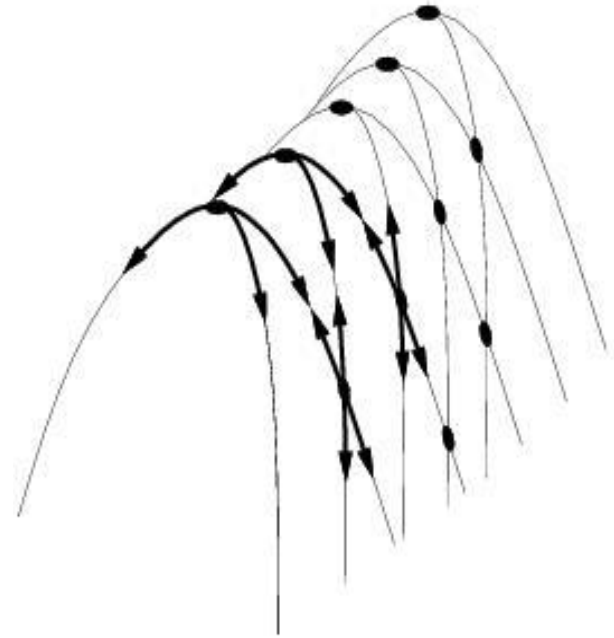
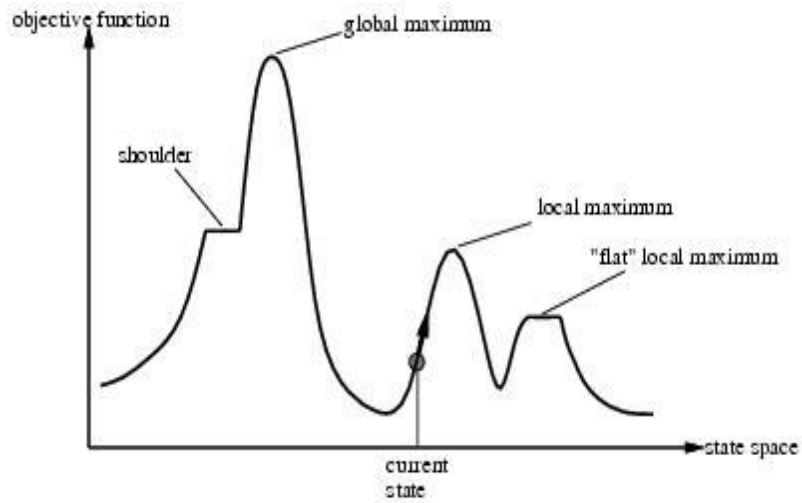
A local minimum in the 8-queens state space ( $h=1$ )

# Problems: Hill climbing and local maxima

- When local maxima exist, hill climbing is suboptimal
- Simple (often effective) solution
  - Multiple random restarts



## Problems: Other drawbacks



- Ridge = sequence of local maxima difficult for greedy algorithms to navigate
- Plateau = an area of the state space where the evaluation function is flat.

# Performance of hill-climbing on 8-queens (Statistics)

- Randomly generated 8-queens starting states...
- 14% the time it solves the problem
- 86% of the time it get stuck at a local minimum
- However...
  - Takes only 4 steps on average when it succeeds
  - And 3 on average when it gets stuck
  - (for a state space with ~17 million states)

## Possible solution...sideways moves

- If no downhill (uphill) moves, allow sideways moves in hope that algorithm can escape
  - Need to place a limit on the possible number of sideways moves to avoid infinite loops
- For 8-queens
  - Now **allow sideways moves with a limit of 100**
  - Raises percentage of problem instances solved from 14 to 94%
  - However on average....
    - 21 steps for every successful solution
    - 64 for each failure

# Hill-climbing variations

## 1. Stochastic hill-climbing

- **Random selection** among the uphill moves.
- The selection probability can vary with the **steepness of the uphill move**.
- This usually converges slowly than **steepest ascent**.

## 2. First-choice hill-climbing

- Stochastic hill climbing by **generating successors randomly until a better one is found**
- Useful when there are a very large number of successors

## 3. Random-restart hill-climbing

- Hill Climbing from randomly generated initial states
- Tries to avoid getting stuck in local maxima.
- Complete

# Hill-climbing with random restarts

- Different variations
  - For each restart: run until termination v. Run for a fixed time
  - Run a fixed number of restarts or run indefinitely
- Analysis
  - Say each search has probability  $p$  of success then expected no of restarts required is:  $1/p$ .
    - E.g., for 8-queens,  $p = 0.14$  with no sideways moves
  - **Expected number of restarts?**  
**Ans:**  $1/p$
  - **Expected number of steps taken?**  
**Ans:**  $p \times \text{avg. of success} + (1-p) \times \text{avg. steps of failure}$



## Search using Simulated Annealing (2)

- Simulated Annealing = hill-climbing with non-deterministic search (i.e. randomness)
- Basic ideas:
  - 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  $\delta$
  - if  $\delta$  is **positive**, then move to that state
  - **otherwise**:
    - move to this state with probability proportional to  $\delta$
    - thus: worse moves (very large negative  $\delta$ ) are executed less often
  - however, 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)

# Physical Interpretation of Simulated Annealing

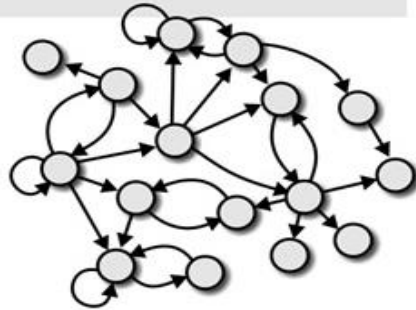
- **A Physical Analogy:**
  - imagine letting a ball roll downhill on the function surface
    - this is like hill-climbing (for minimization)
  - now imagine **shaking the surface, while the ball rolls, gradually reducing the amount of shaking**
    - this is like simulated annealing
- **Annealing** = physical process of cooling a liquid or metal until particles achieve a certain frozen crystal state
  - simulated annealing:
    - free variables are like particles
    - seek “low energy” (high quality) configuration
    - get this by slowly reducing temperature  $T$ , which particles move around randomly

# Physical Interpretation of Simulated Annealing

Huang et al.,  
Semin Cell Dev Biol.  
2009 Sep; 20(7): 869-7.

**COMPLEX NETWORK**

( $N$  gene genome)



**Epigenetic barrier**  
(can be lowered  
by mutation)

**Cancer attractor**  
(preexisting or  
generated by  
mutation,  
ormally unused)

**developmental  
trajectories**

**"Potential"**  
=elevation  
reflecting  
stability  
of a state

state space coordinate (projection)

each point is a  
network state:



**un-stable**  
= transient state



**stable**  
= mature cell type



**stable**  
= another mature cell type

# 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 probability of downward steps

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

**for** *t*  $\leftarrow$  1 **to**  $\infty$  **do**

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

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

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

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

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

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

# More Details on Simulated Annealing

- Lets say there are 3 moves available, with changes in the objective function of  $d1 = -0.1$ ,  $d2 = 0.5$ ,  $d3 = -5$ . (Let  $T = 1$ ).
- pick a move randomly:
  - if  $d2$  is picked, move there.
  - if  $d1$  or  $d3$  are picked, probability of move =  $\exp(d/T)$
  - move 1:  $\text{prob1} = \exp(-0.1) = 0.9$ ,
    - i.e., 90% of the time we will accept this move
  - move 3:  $\text{prob3} = \exp(-5) = 0.05$ 
    - i.e., 5% of the time we will accept this move
- $T$  = “temperature” parameter
  - high  $T \Rightarrow$  probability of “locally bad” move is higher
  - low  $T \Rightarrow$  probability of “locally bad” move is lower
  - typically,  $T$  is decreased as the algorithm runs longer
    - i.e., there is a “temperature schedule”

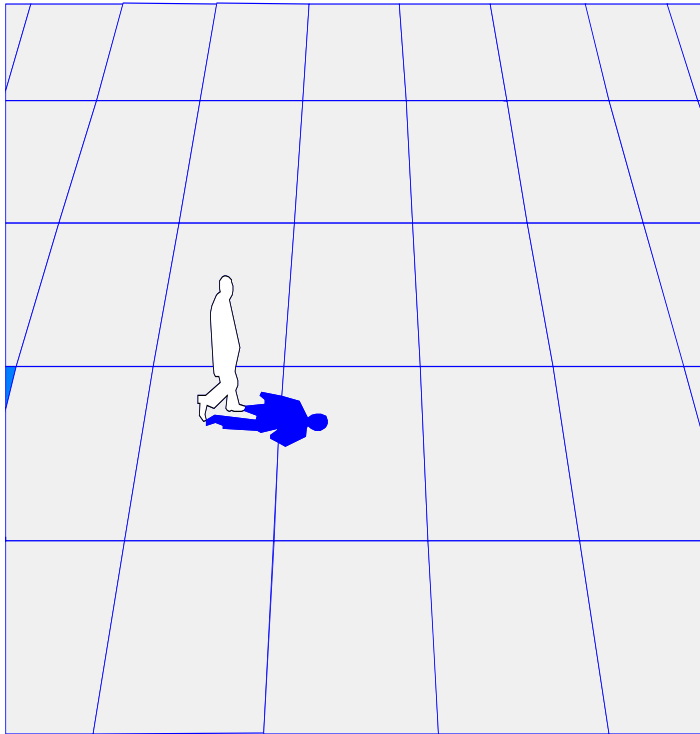
# Simulated Annealing in Practice

- method proposed in 1983 by IBM researchers for solving **VLSI layout** problems (Kirkpatrick et al, *Science*, 220:671-680, 1983).
  - Very-large-scale integration (**VLSI**) is the process of creating an integrated circuit (IC) by combining thousands of transistors into a single chip. **VLSI** began in the 1970s when complex semiconductor and communication technologies were being developed. The microprocessor is a **VLSI** device.
  - theoretically will always find the global optimum (the best solution)
- useful for some problems, but can be very slow
  - slowness comes about because  $T$  must be decreased very gradually to retain optimality
  - In practice how do we decide the rate at which to decrease  $T$ ? (this is a practical problem with this method)

## Local beam search (3)

- Keep track of  $k$  states instead of one
  - Initially:  $k$  randomly selected states
  - Next: determine all successors of  $k$  states
  - If any of successors is goal  $\rightarrow$  finished
  - Else **select  $k$  best from ALL successors** and repeat.
- Major difference with random-restart search
  - Information is shared among  $k$  search threads.
- Can suffer from lack of diversity.
  - Stochastic beam search
    - choose  $k$  successors at random proportional to state quality.

# Genetic Algorithms



*“Genetic Algorithms are good at taking large, potentially huge search spaces and navigating them, looking for optimal combinations of things, solutions you might not otherwise find in a lifetime.”*

- Salvatore Mangano  
*Computer Design*, May 1995



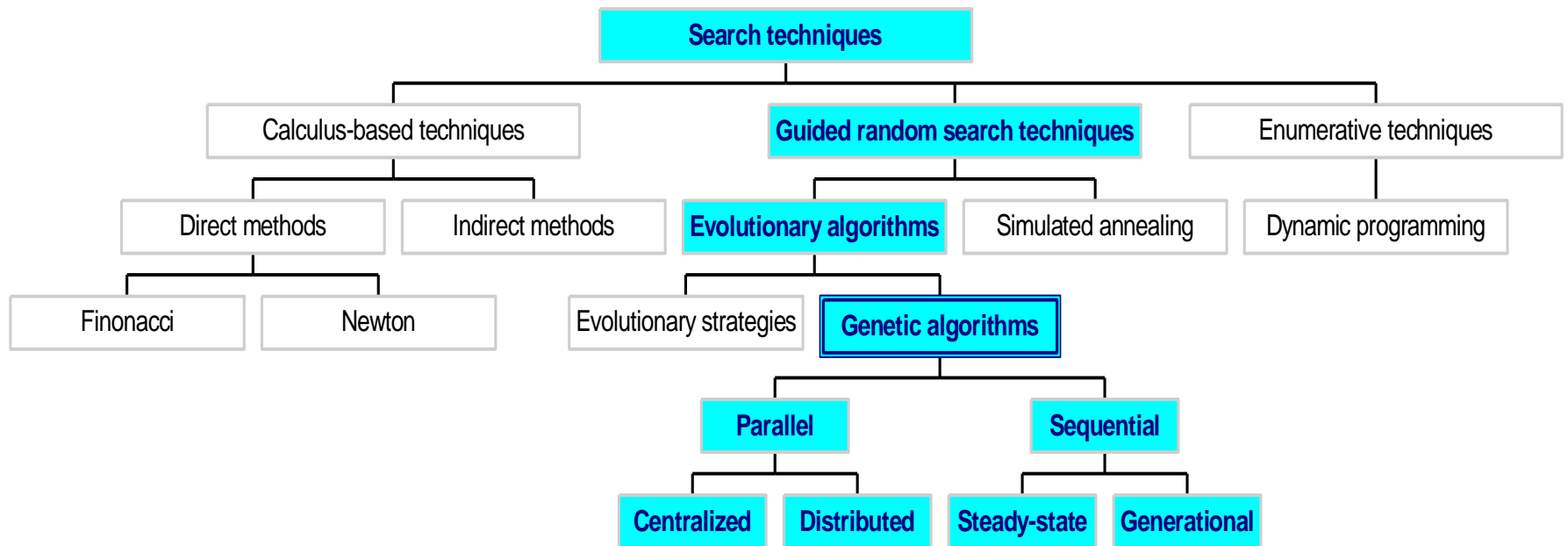
# The Genetic Algorithm

- Directed search algorithms based on the mechanics of biological evolution
- Developed by John Holland, University of Michigan (1970's)
  - ♦ To understand the adaptive processes of natural systems
  - ♦ To design artificial systems software that retains the robustness of natural systems

# The Genetic Algorithm (cont.)

- Provide efficient, effective techniques for optimization and machine learning applications
- Widely-used today in business, scientific and engineering circles

# Classes of Search Techniques



# Components of a GA

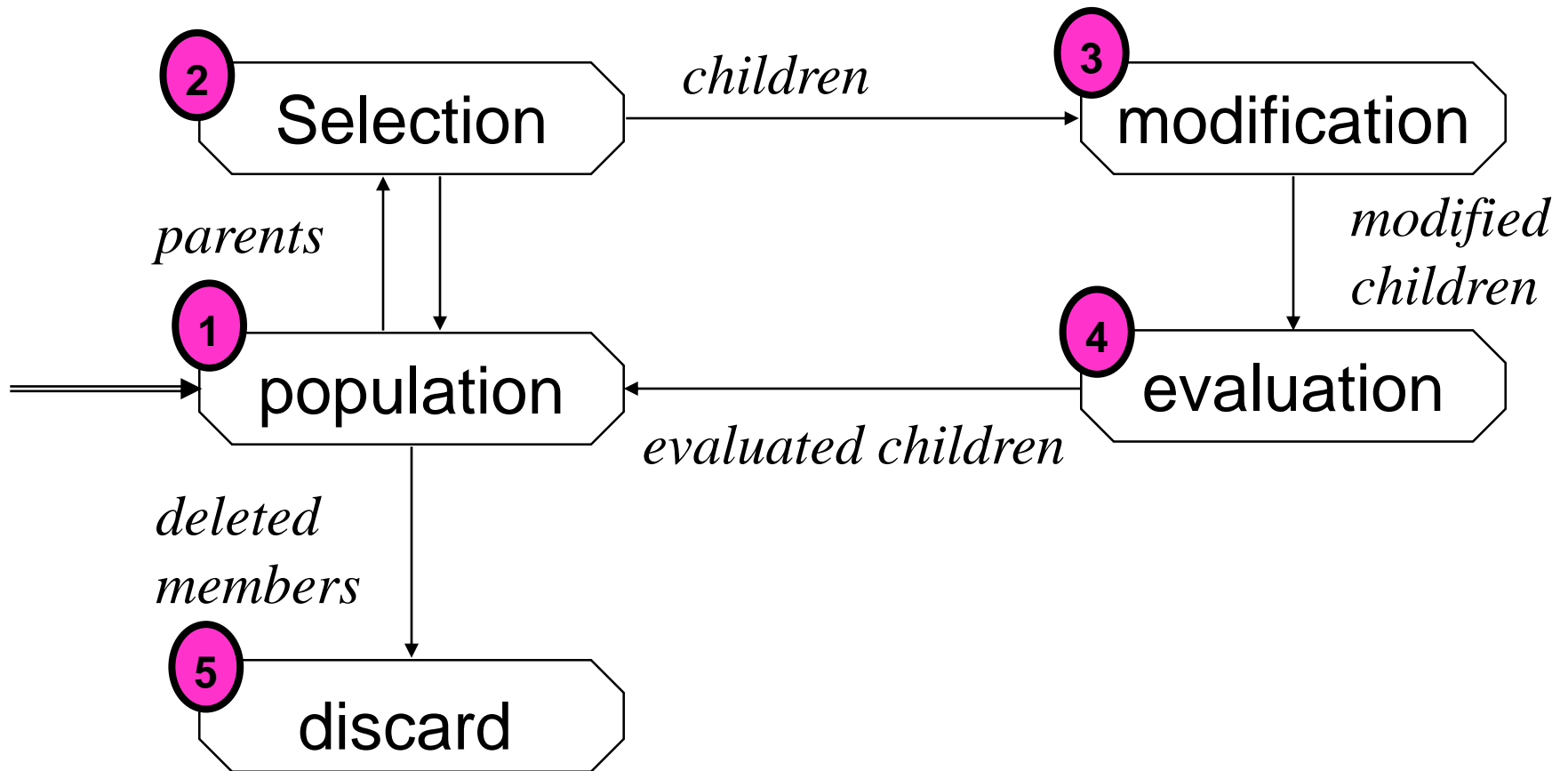
A problem to solve, and ...

- Encoding technique *(gene, chromosome)*
- Initialization procedure *(creation)*
- Evaluation function *(environment)*
- Selection of parents *(reproduction)*
- Genetic operators *(mutation, recombination)*
- Parameter settings *(practice and art)*

# Simple Genetic Algorithm

```
{  
  initialize population;  
  evaluate population;  
  while TerminationCriteriaNotSatisfied  
  {  
    select parents for reproduction;  
    perform recombination and mutation;  
    evaluate population;  
  }  
}
```

# The GA Cycle of Reproduction



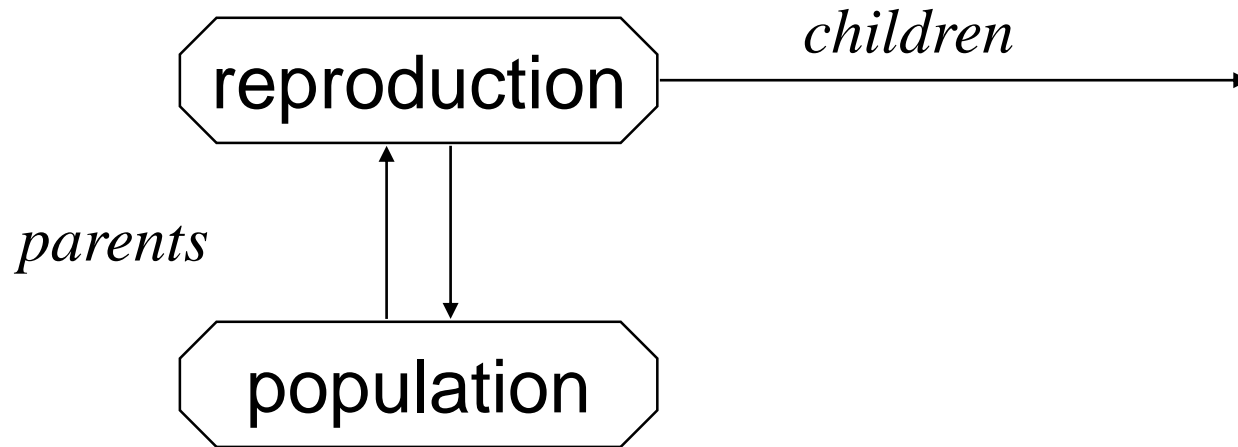
# 1- Population



Chromosomes could be:

- ♦ Bit strings (0101 ... 1100)
- ♦ Real numbers (43.2 -33.1 ... 0.0 89.2)
- ♦ Permutations of element (E11 E3 E7 ... E1 E15)
- ♦ Lists of rules (R1 R2 R3 ... R22 R23)
- ♦ Program elements (genetic programming)
- ♦ ... any data structure ...

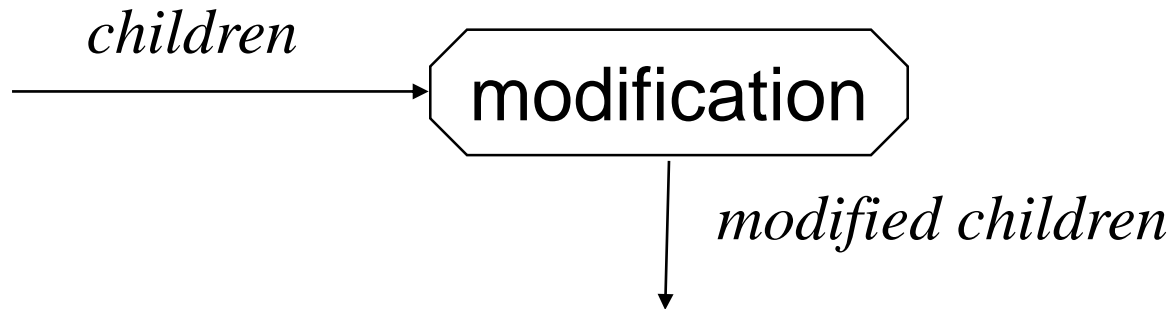
## 2- Reproduction/Selection



Parents are selected at random with selection chances biased in relation to chromosome evaluations.

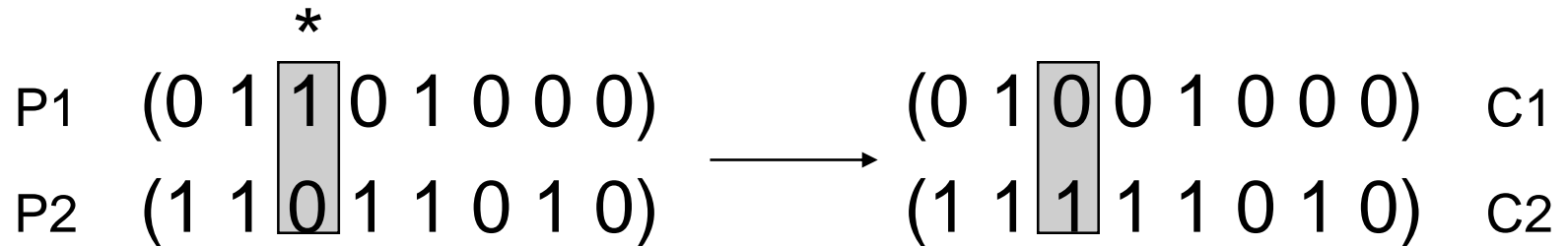


# 3- Chromosome Modification



- Modifications are stochastically triggered
- Operator types are:
  - ♦ Mutation
  - ♦ Crossover (recombination)

# Crossover: Recombination



Crossover is a critical feature of genetic algorithms:

- ◆ It greatly accelerates search early in evolution of a population
- ◆ It leads to effective combination of schemata (subsolutions on different chromosomes)

# Mutation: Local Modification

Before: (1 0 1 1 0 1 1 0)

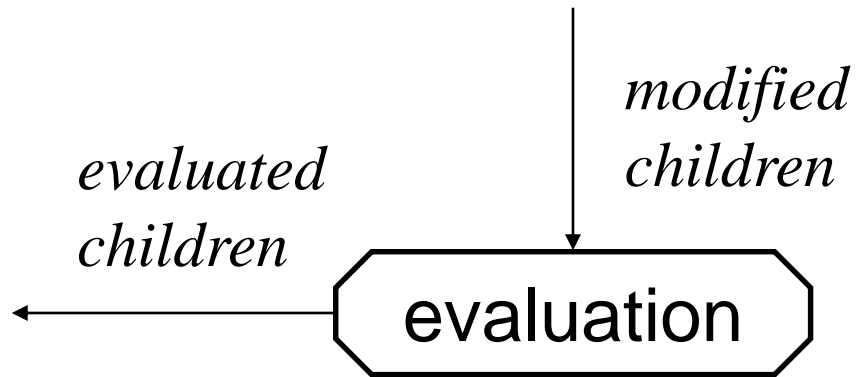
After: (0 1 1 0 0 1 1 0)

Before: (1.38 -69.4 326.44 0.1)

After: (1.38 -67.5 326.44 0.1)

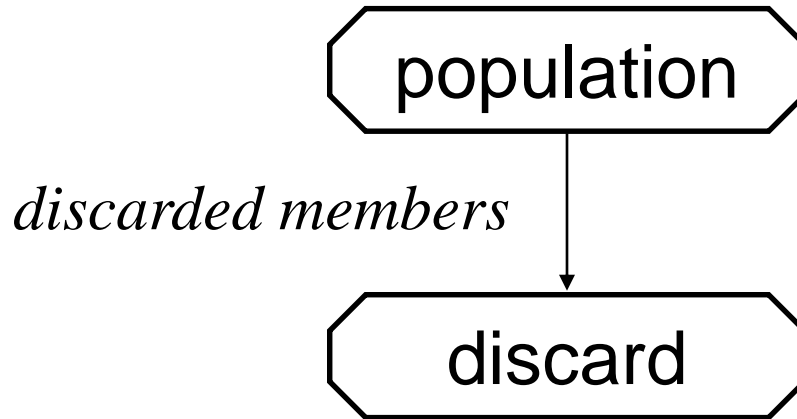
- Causes movement in the search space (local or global)
- Restores lost information to the population

# 4- Evaluation



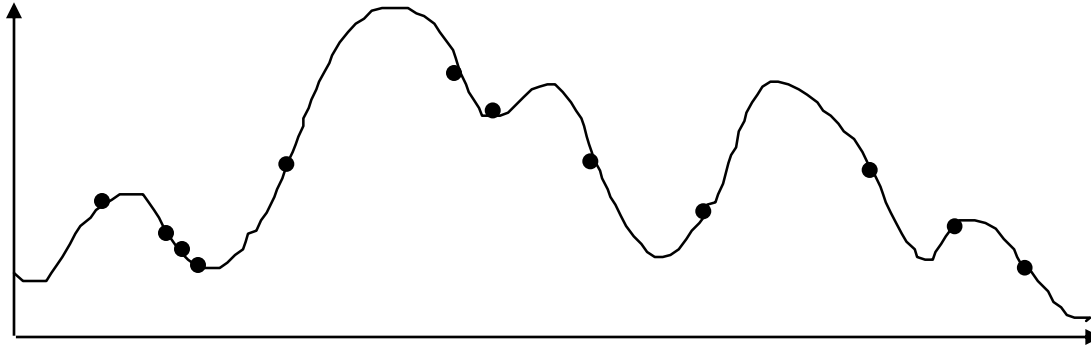
- The evaluator decodes a chromosome and assigns it a fitness measure
- The evaluator is the only link between a classical GA and the problem it is solving

## 5- Deletion

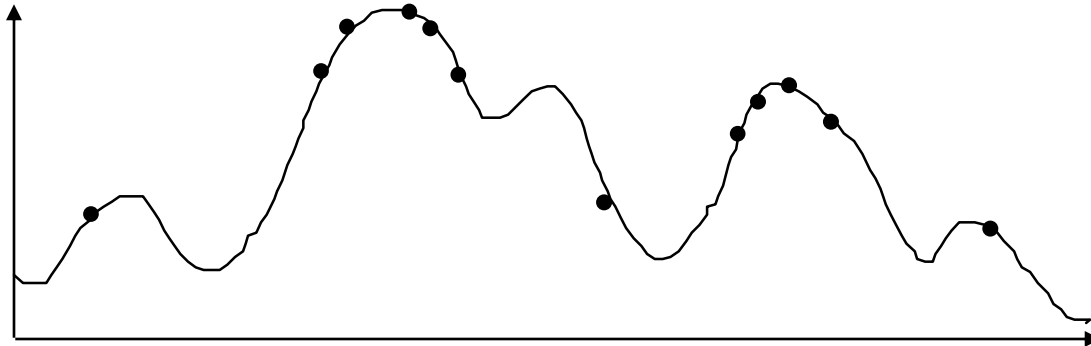


- *Generational GA*:  
entire populations replaced with each iteration
- *Steady-state GA*:  
a few members replaced each generation

# An Abstract Example

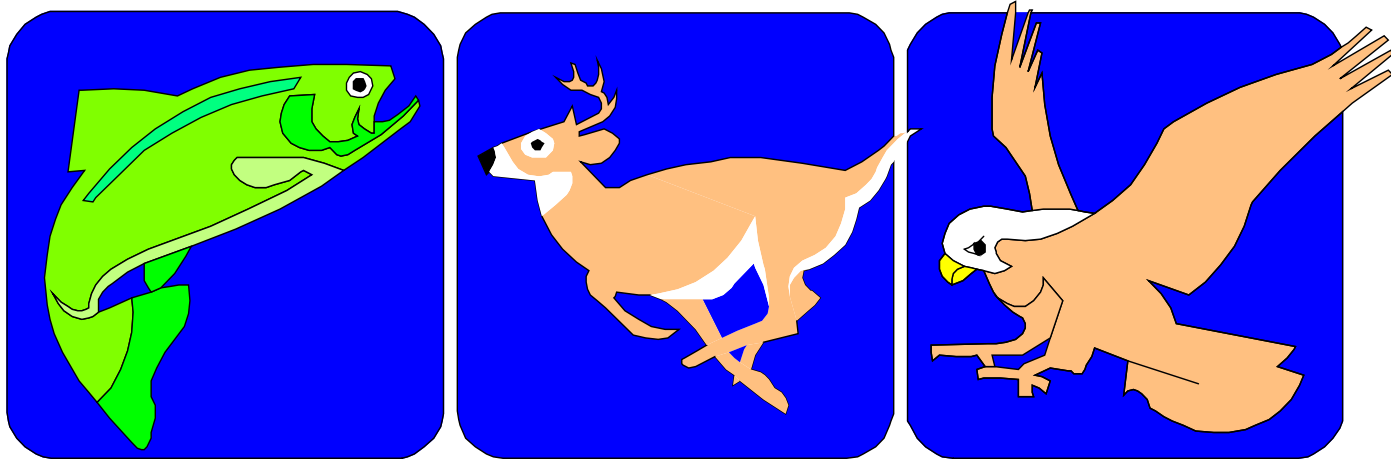


*Distribution of Individuals in Generation 0*



*Distribution of Individuals in Generation N*

# A Simple Example



*“The Gene is by far the most sophisticated program around.”*

- Bill Gates, *Business Week*, June 27, 1994

# A Simple Example

The Traveling Salesman Problem:

Find a tour of a given set of cities so that

- ♦ each city is visited only once
- ♦ the total distance traveled is minimized



# Representation & Selection

Representation is an ordered list of city numbers known as an *order-based* GA.

1) London	3) Dunedin	5) Beijing	7) Tokyo
2) Venice	4) Singapore	6) Phoenix	8) Victoria

CityList1      (3   5   7   2   1   6   4   8)

CityList2      (2   5   7   6   8   1   3   4)

# Crossover

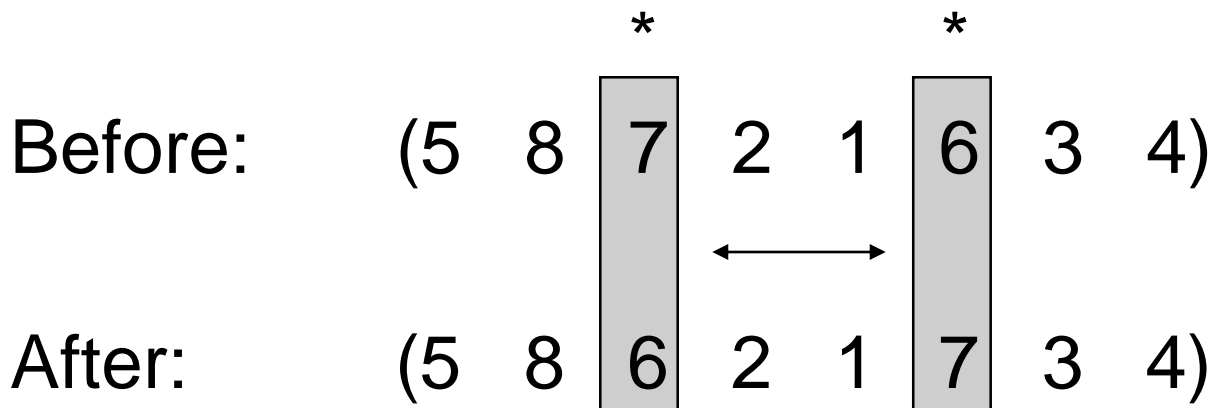
Crossover combines inversion and recombination:

			*		*		
Parent1	(3	5	7	2	1	6	4 8)
Parent2	(2	5	7	6	8	1	3 4)
<hr/>							
Child	(5	8	7	2	1	6	3 4)

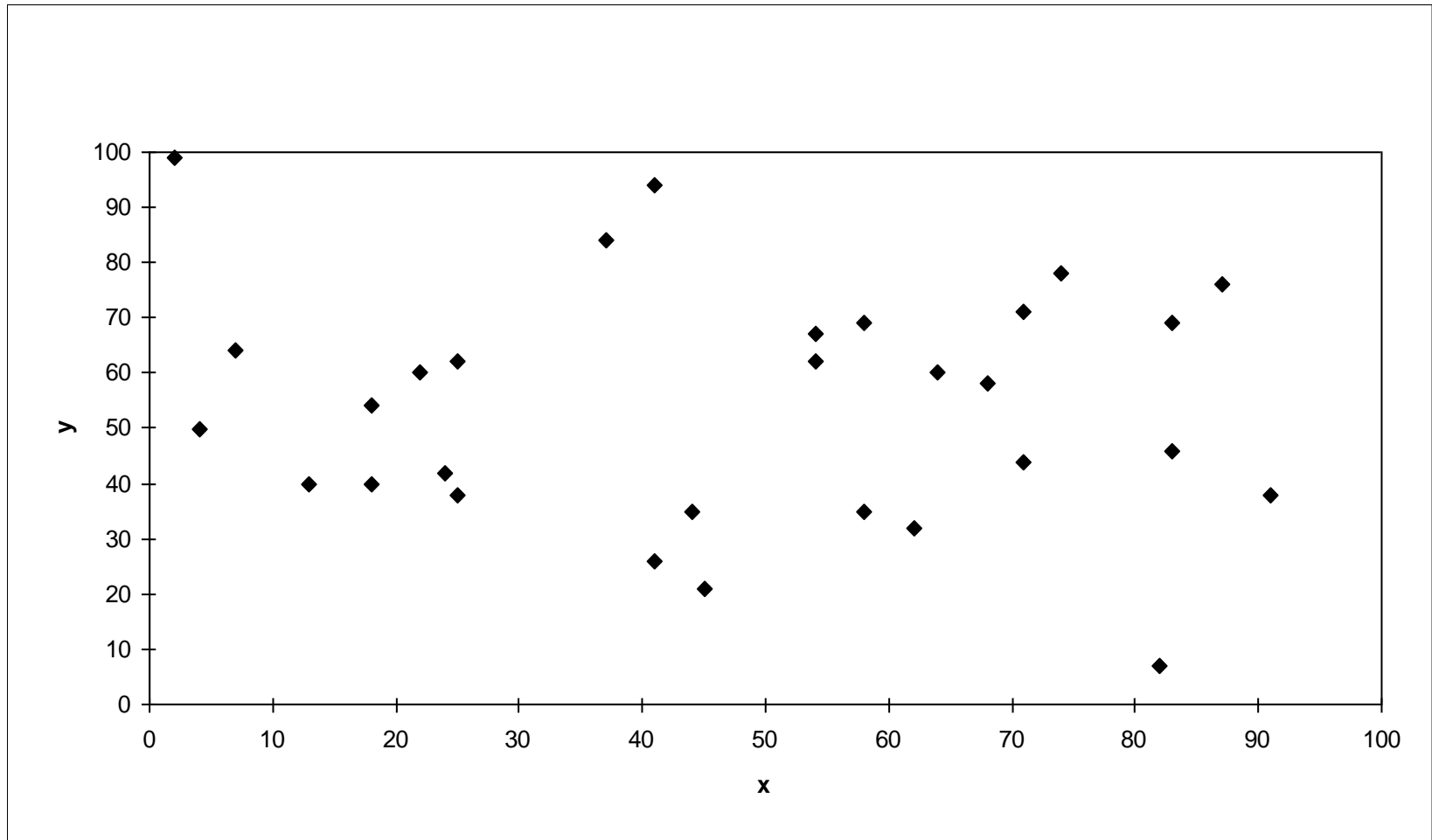
This operator is called the *Order1* crossover.

# Mutation

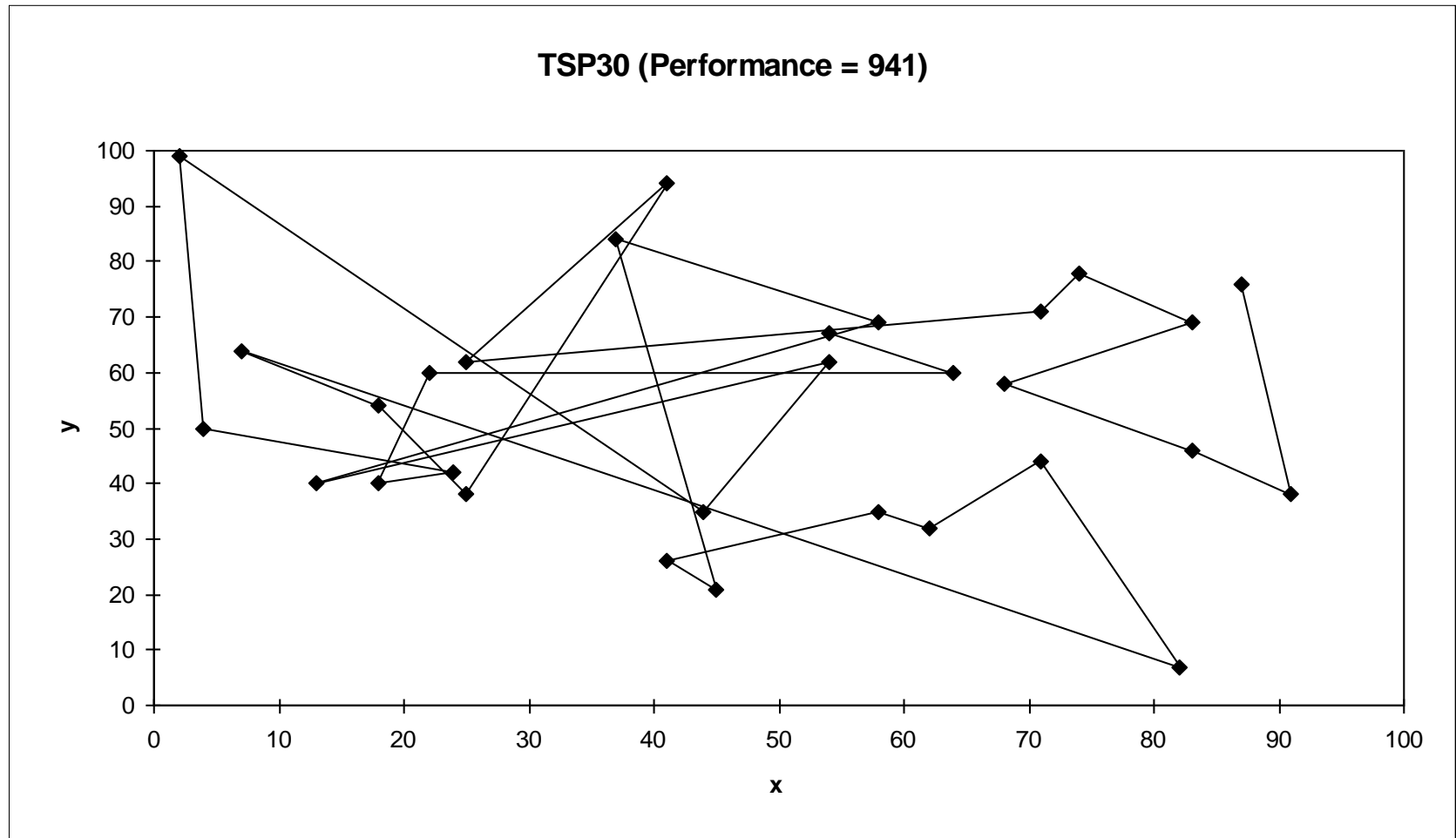
Mutation involves reordering of the list:



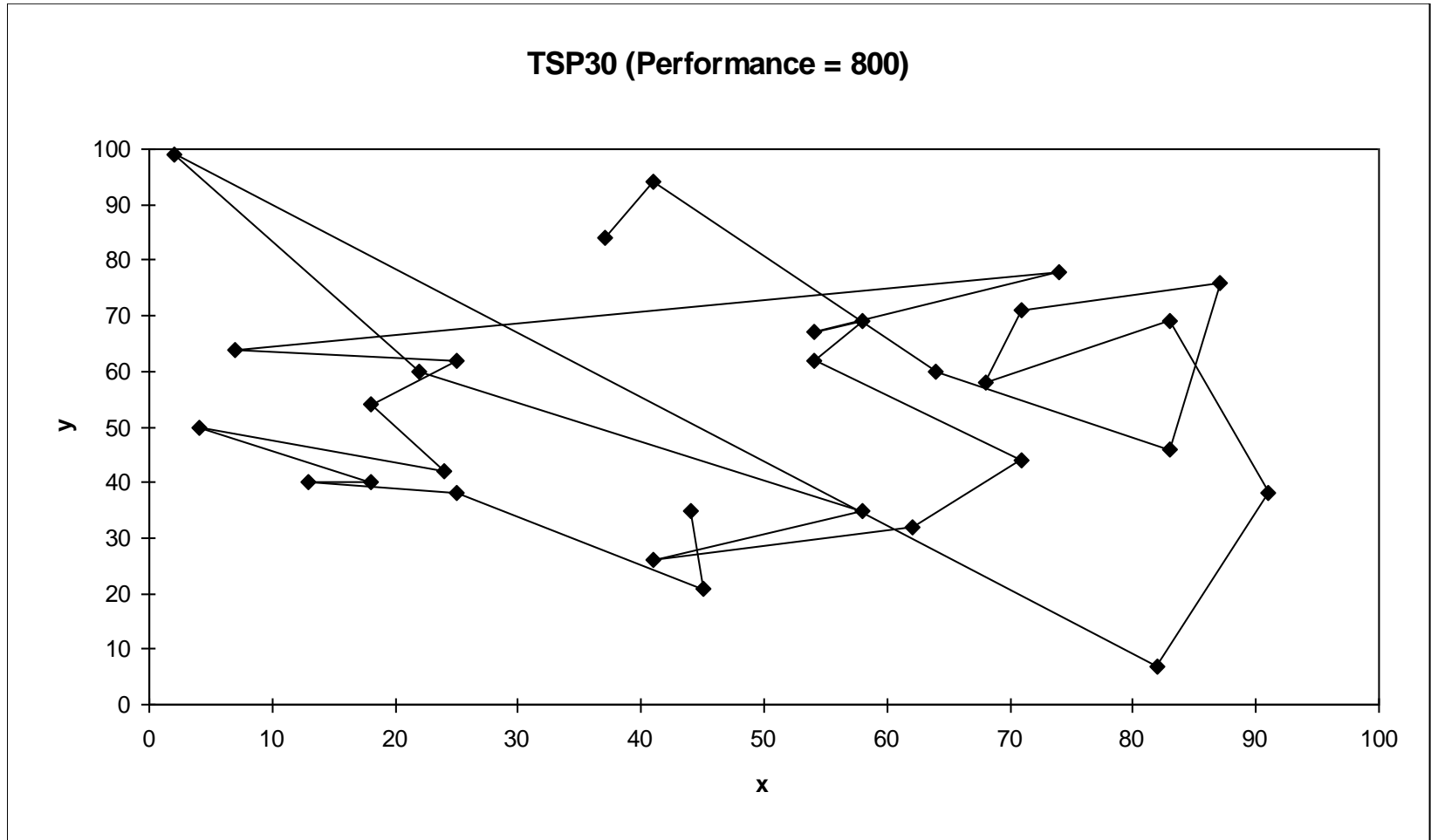
# TSP Example: 30 Cities



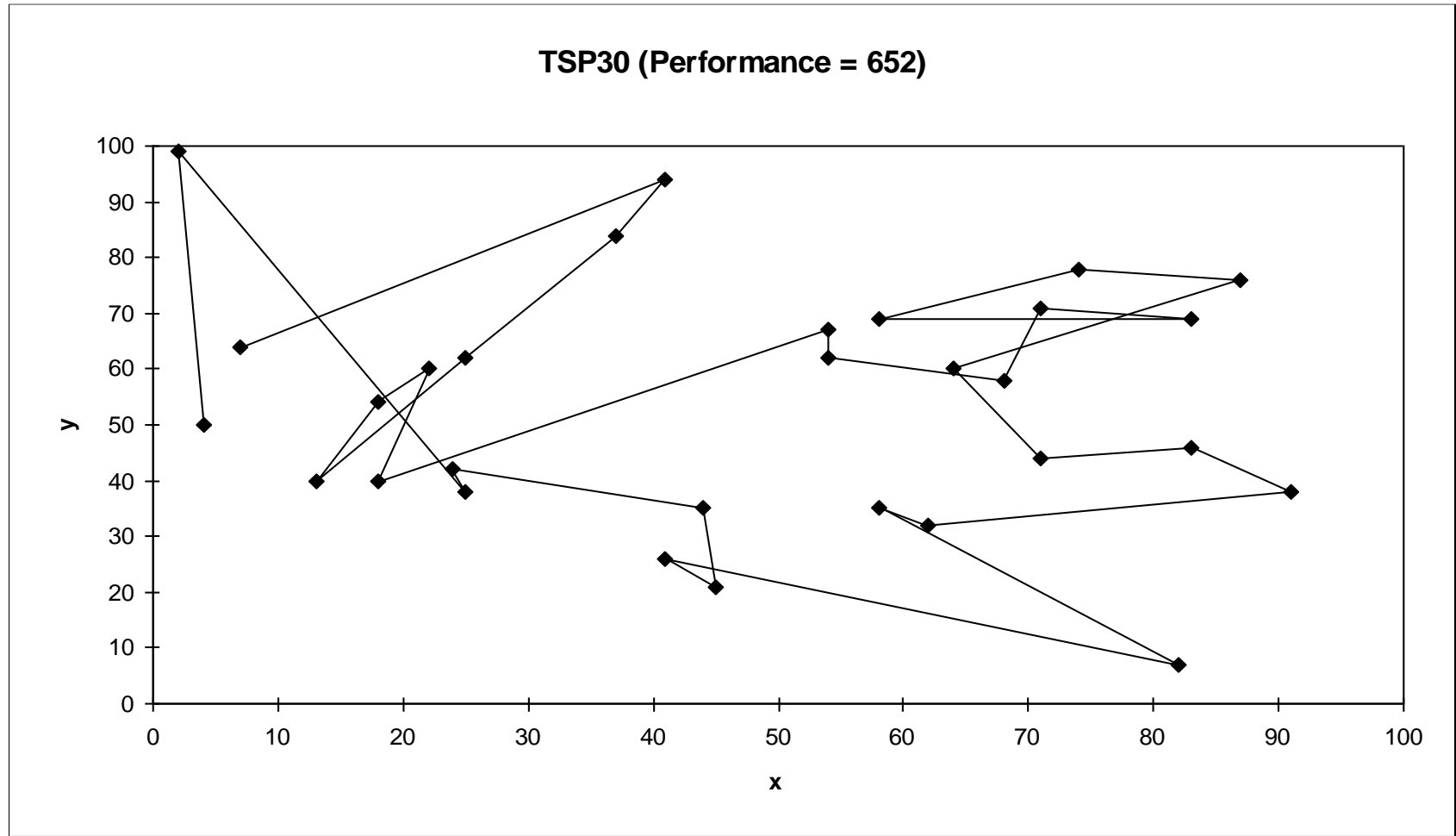
# Solution $i$ (Distance = 941)



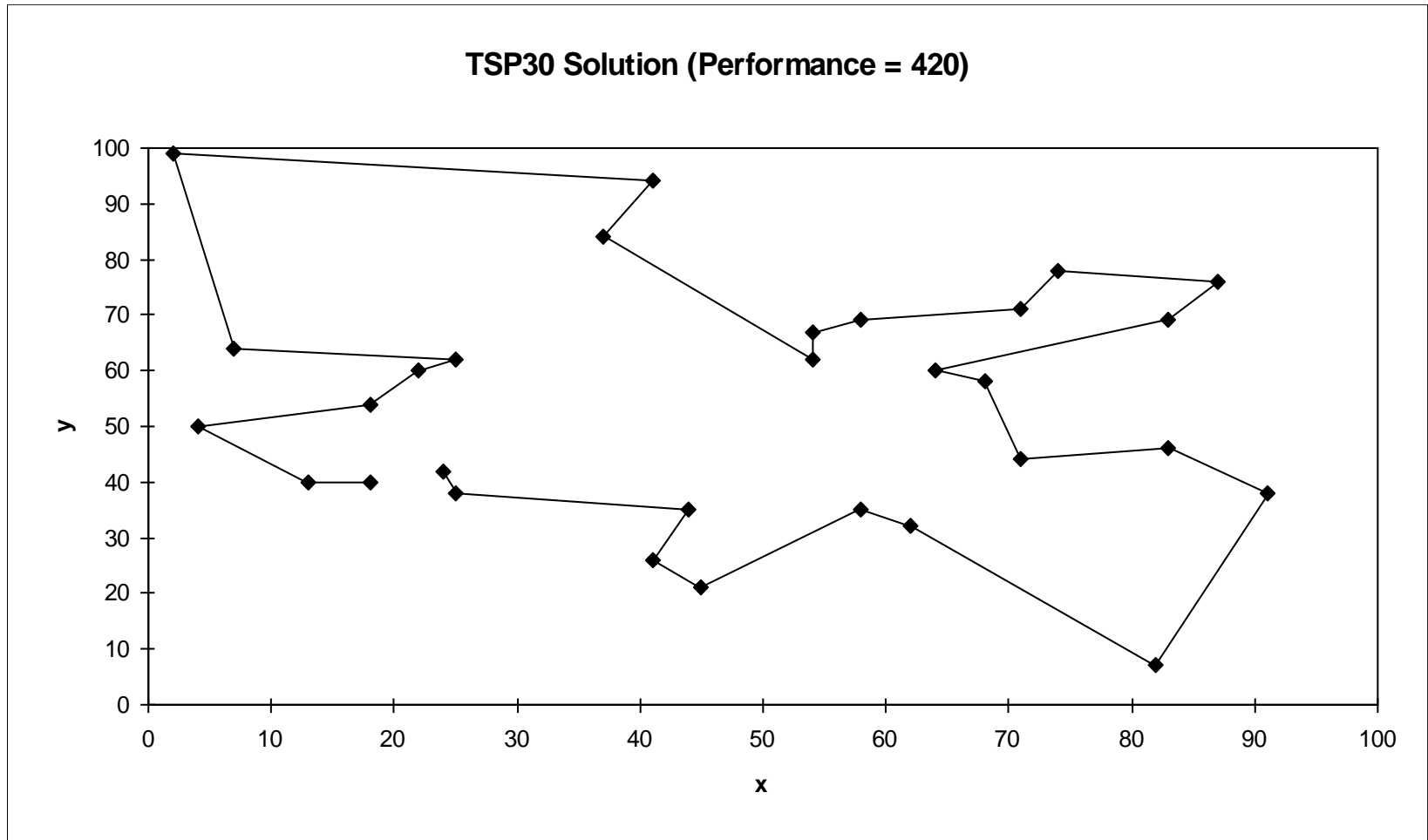
# Solution $j$ (Distance = 800)



# Solution $_k$ (Distance = 652)



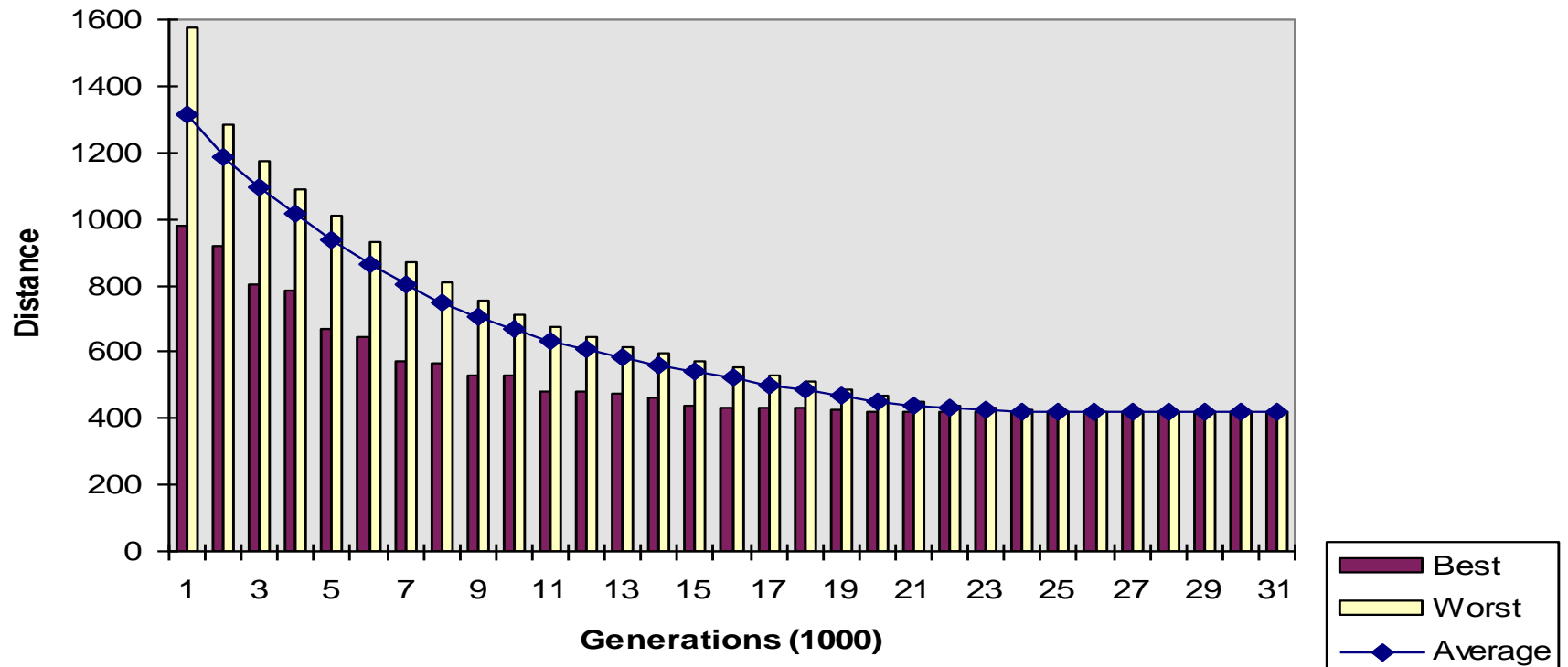
# Best Solution (Distance = 420)





# Overview of Performance

**TSP30 - Overview of Performance**



# Genetic algorithms- 8 Queen Example

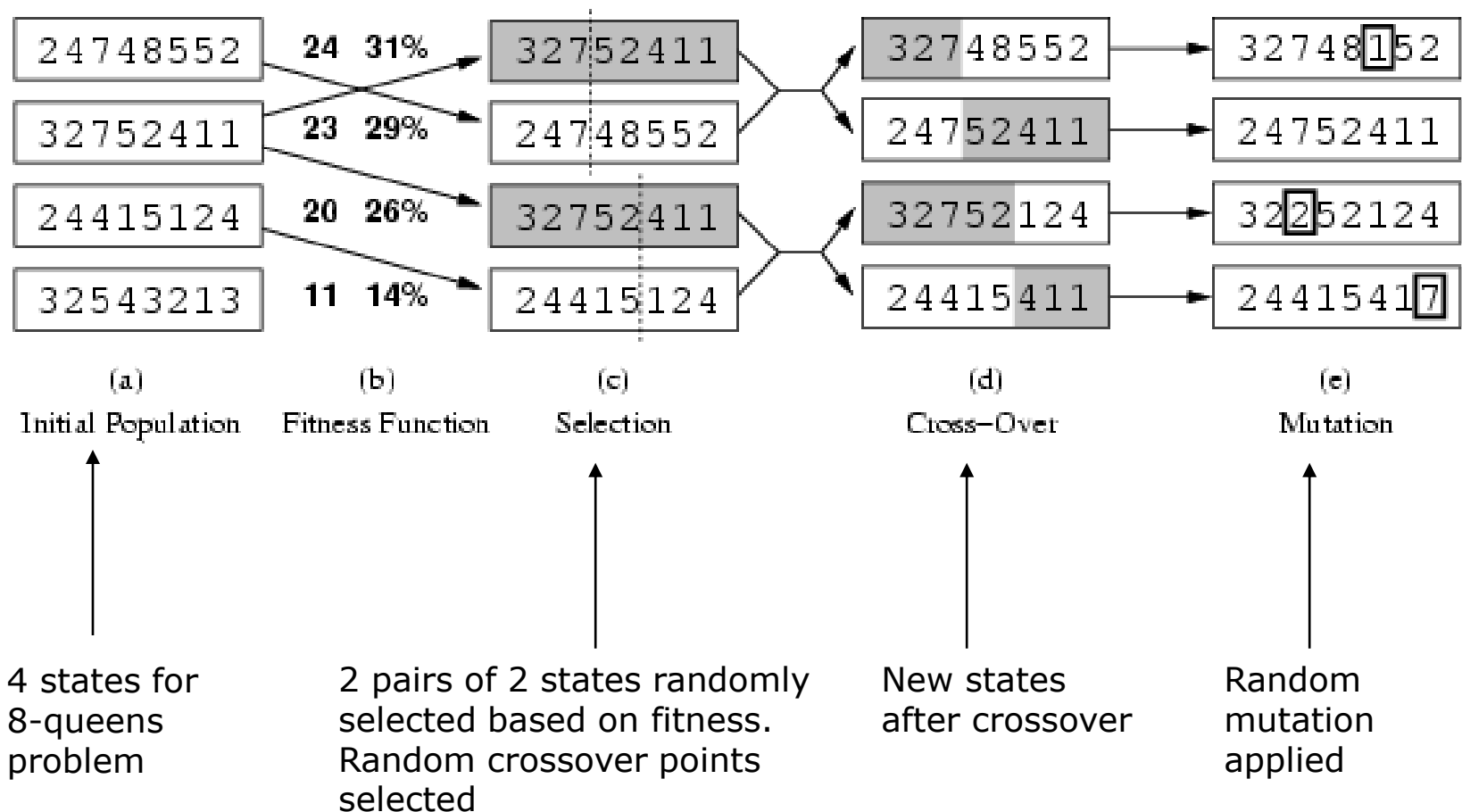
- A state is represented as a string over a finite alphabet (e.g. binary)
  - 8-queens
    - State = position of 8 queens each in a column  
=>  $8 \times \log(8)$  bits = 24 bits (for binary representation)
- **Start with  $k$  randomly generated states (population)**
- **Evaluation function (fitness function).**
  - Higher values for better states.
  - Opposite to heuristic function, e.g., # non-attacking pairs in 8-queens
  - Solution has a **value of 28**
- **Produce the next generation of states by “simulated evolution”**
  - **Random selection**
  - **Crossover**
  - **Random mutation**

# Genetic algorithms

**Fitness function:** number of non-attacking pairs of queens (min = 0, max =  $8 \times 7/2 = 28$ )

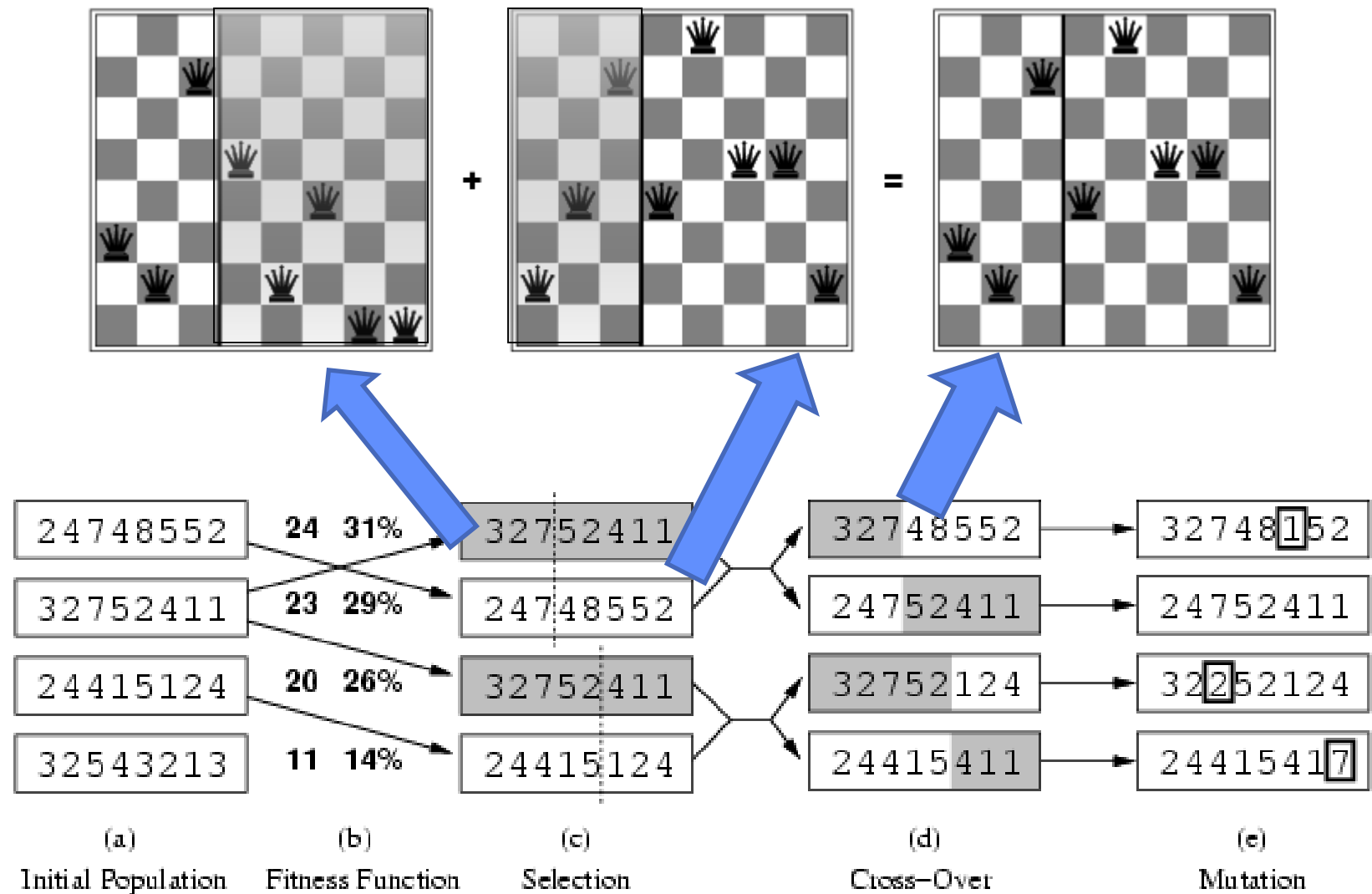
$24/(24+23+20+11) = 31\%$

$23/(24+23+20+11) = 29\%$  etc



# Genetic algorithms

Like Simulated Annealing **Cross Over** takes **large** steps at the **beginning** of the search process while **small** steps when the population (individuals) are quite similar.



# Genetic algorithm pseudocode

**function** GENETIC\_ALGORITHM( *population*, FITNESS-FN) **return** an individual

**input:** *population*, a set of individuals

FITNESS-FN, a function which determines the quality of the individual

**repeat**

*new\_population*  $\leftarrow$  empty set

**loop for** *i* **from** 1 **to** SIZE(*population*) **do**

*x*  $\leftarrow$  RANDOM\_SELECTION(*population*, FITNESS\_FN)

*y*  $\leftarrow$  RANDOM\_SELECTION(*population*, FITNESS\_FN)

*child*  $\leftarrow$  REPRODUCE(*x*,*y*)

**if** (small random probability) **then** *child*  $\leftarrow$  MUTATE(*child*)

add *child* to *new\_population*

*population*  $\leftarrow$  *new\_population*

**until** some individual is fit enough or enough time has elapsed

**return** the best individual

# Issues for GA Practitioners

- Choosing basic implementation issues:
  - ♦ representation
  - ♦ population size, mutation rate, ...
  - ♦ selection, deletion policies
  - ♦ crossover, mutation operators
- Termination Criteria
- Performance, scalability
- Solution is only as good as the evaluation function (often hardest part)

# Benefits of Genetic Algorithms

- Concept is easy to understand
- Modular, separate from application
- Supports multi-objective optimization
- Good for “noisy” environments
- Always an answer; answer gets better with time
- Inherently parallel; easily distributed

# Benefits of Genetic Algorithms (cont.)

- Many ways to speed up and improve a GA-based application as knowledge about problem domain is gained
- Easy to exploit previous or alternate solutions
- Flexible building blocks for hybrid applications
- Substantial history and range of use