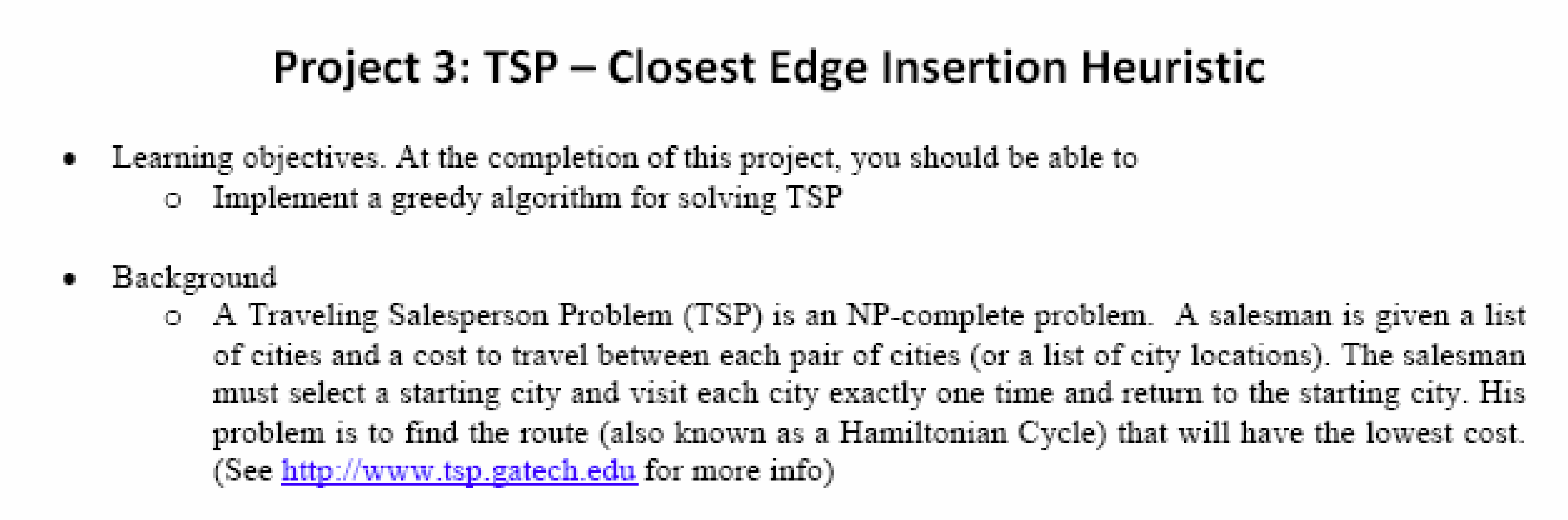
Project 3

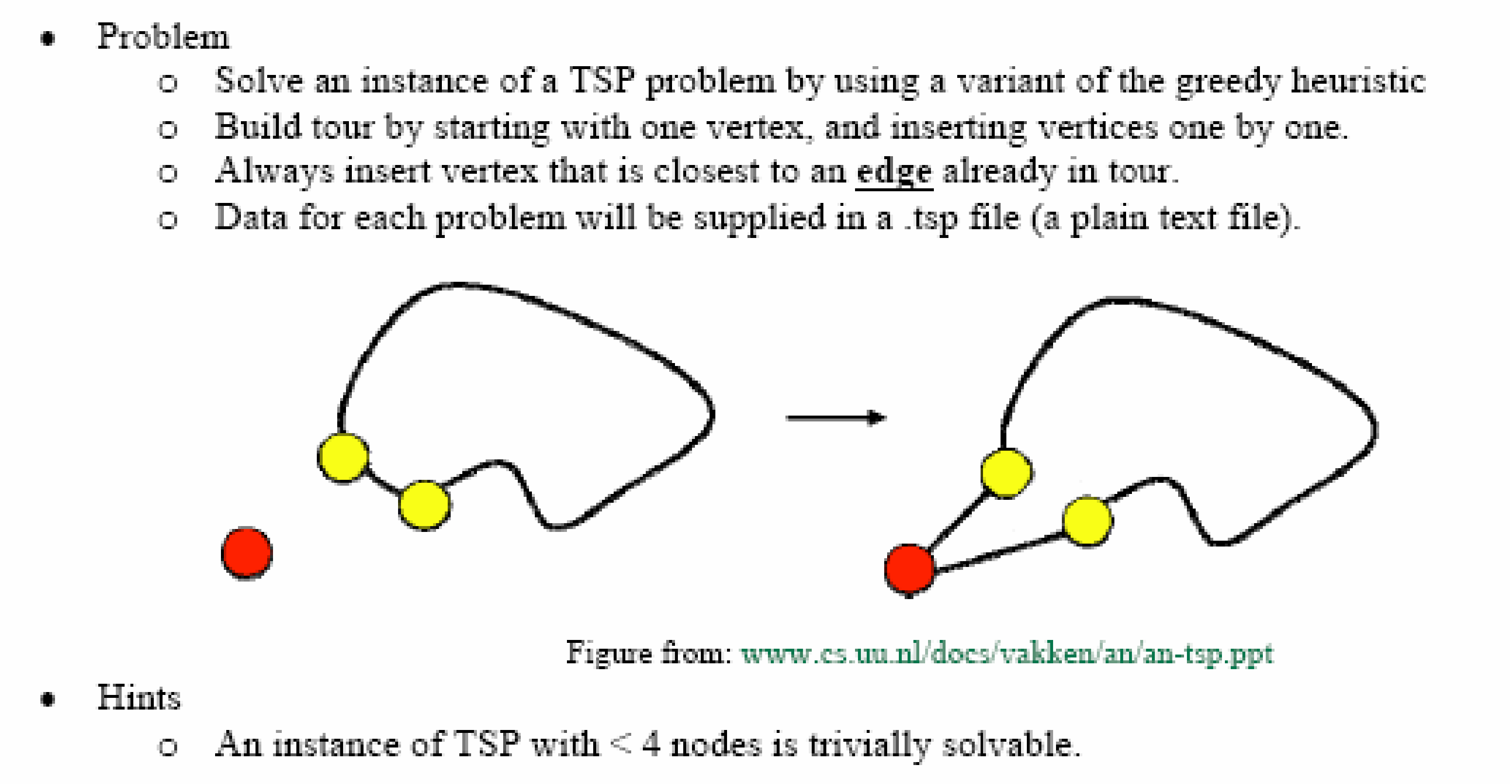


Project 3: TSP – Closest Edge Insertion Heuristic

- Learning objectives. At the completion of this project, you should be able to
  - Implement a greedy algorithm for solving TSP

- Background
  - A Traveling Salesperson Problem (TSP) is an NP-complete problem. A salesman is given a list of cities and a cost to travel between each pair of cities (or a list of city locations). The salesman must select a starting city and visit each city exactly one time and return to the starting city. His problem is to find the route (also known as a Hamiltonian Cycle) that will have the lowest cost. (See http://www.tsp.gatech.edu for more info)

Problem Description

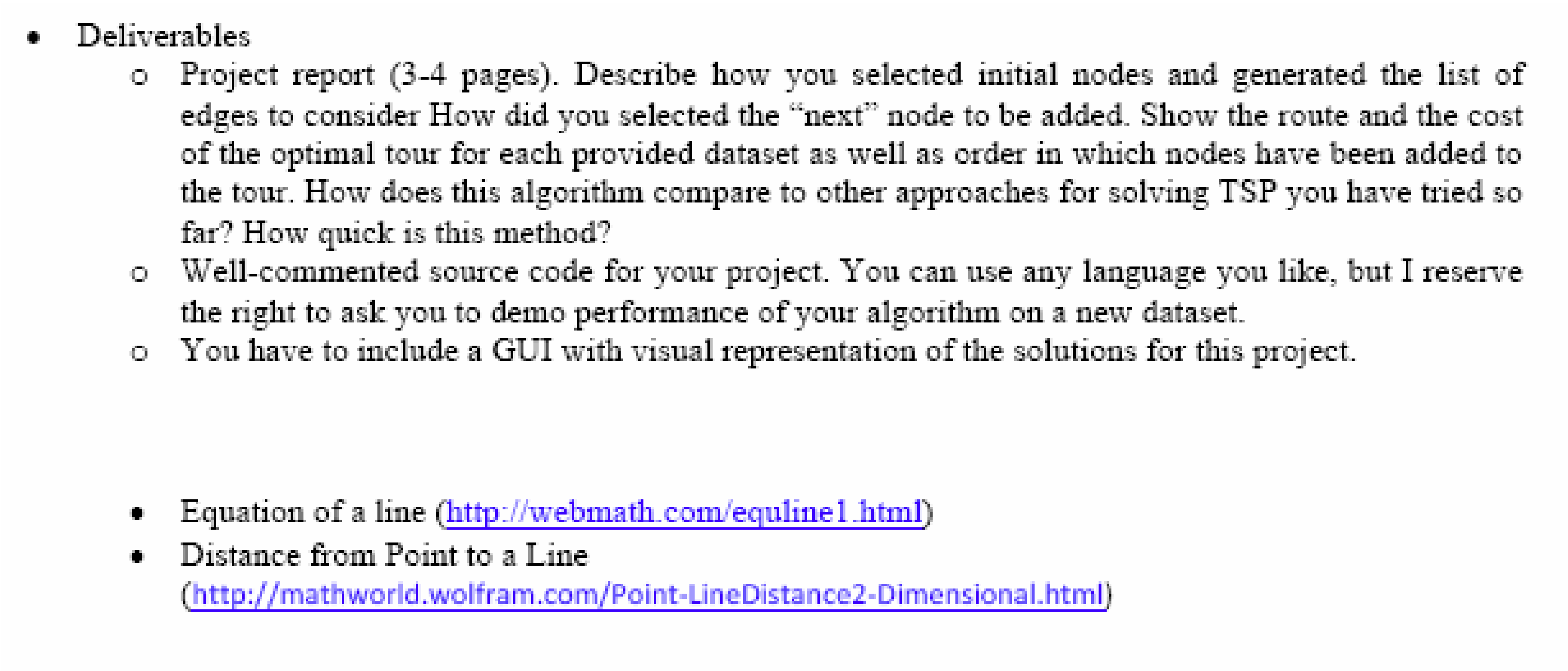


- Problem
  - Solve an instance of a TSP problem by using a variant of the greedy heuristic
  - Build tour by starting with one vertex, and inserting vertices one by one.
  - Always insert vertex that is closest to an edge already in tour.
  - Data for each problem will be supplied in a .tsp file (a plain text file).

Figure from: www.cs.uu.nl/docs/vakken/an/an-tsp.ppt

- Hints
  - An instance of TSP with < 4 nodes is trivially solvable.

Deliverables



- Deliverables
  - Project report (3-4 pages). Describe how you selected initial nodes and generated the list of edges to consider How did you selected the "next" node to be added. Show the route and the cost of the optimal tour for each provided dataset as well as order in which nodes have been added to the tour. How does this algorithm compare to other approaches for solving TSP you have tried so far? How quick is this method?
  - Well-commented source code for your project. You can use any language you like, but I reserve the right to ask you to demo performance of your algorithm on a new dataset.
  - You have to include a GUI with visual representation of the solutions for this project.

- Equation of a line (http://webmath.com/equline1.html)
- Distance from Point to a Line (http://mathworld.wolfram.com/Point-LineDistance2-Dimensional.html)

UNIVERSITY OF LOUISVILLE
J. B. SPEED SCHOOL OF ENGINEERING

Computer Engineering and Computer Science Department

CECS545-Artificial Intelligence

Beyond Classical Search

Dr. Roman Yampolskiy

Iterative improvement algorithms

In many optimization problems, path is irrelevant;
the goal state itself is the solution

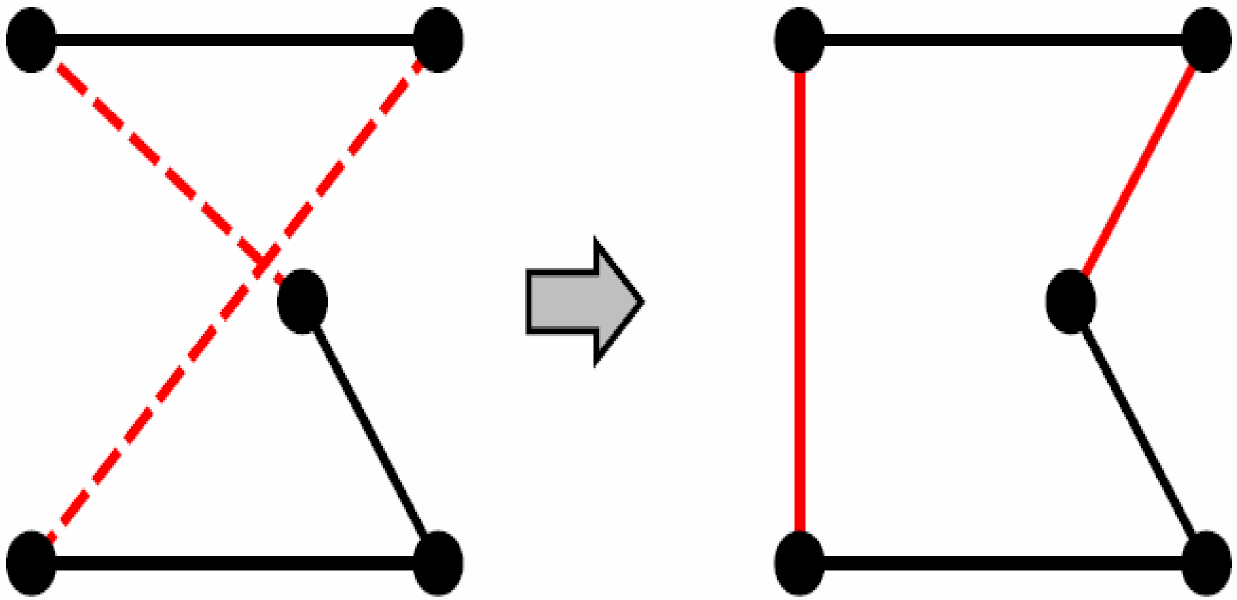
Then state space = set of "complete" configurations;
find optimal configuration, e.g., TSP
or, find configuration satisfying constraints, e.g., timetable

In such cases, can use iterative improvement algorithms;
keep a single "current" state, try to improve it

Constant space, suitable for online as well as offline search

Example: Travelling Salesperson Problem

Start with any complete tour, perform pairwise exchanges

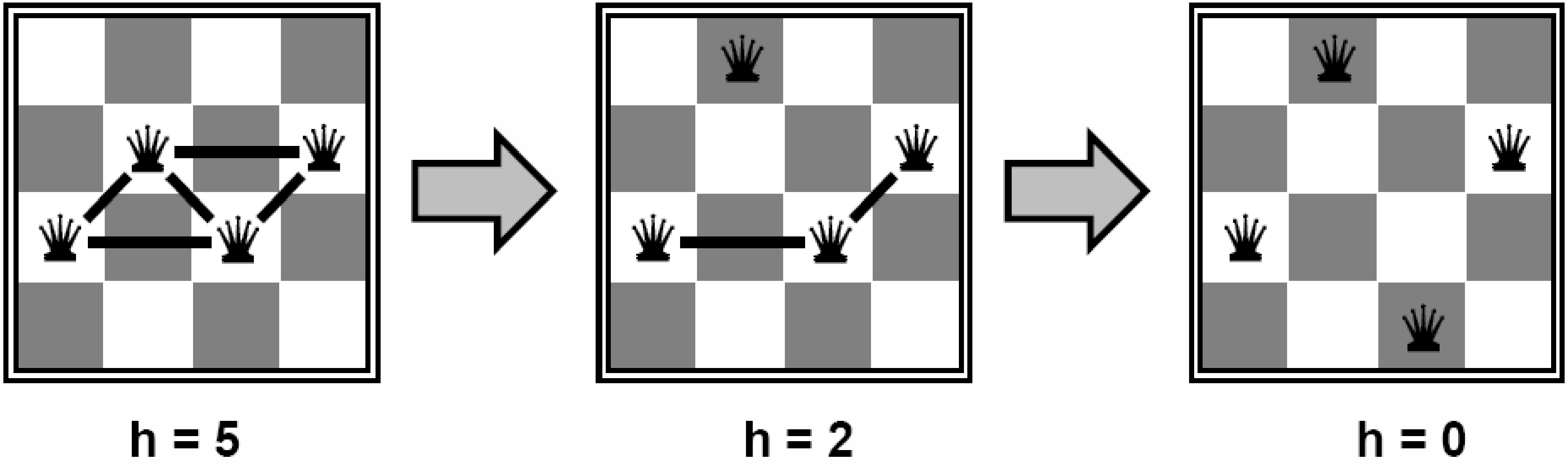


Variants of this approach get within 1% of optimal very quickly with thousands of cities

Example: n-queens

Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

Move a queen to reduce number of conflicts



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

Hill-climbing (or gradient ascent/descent)

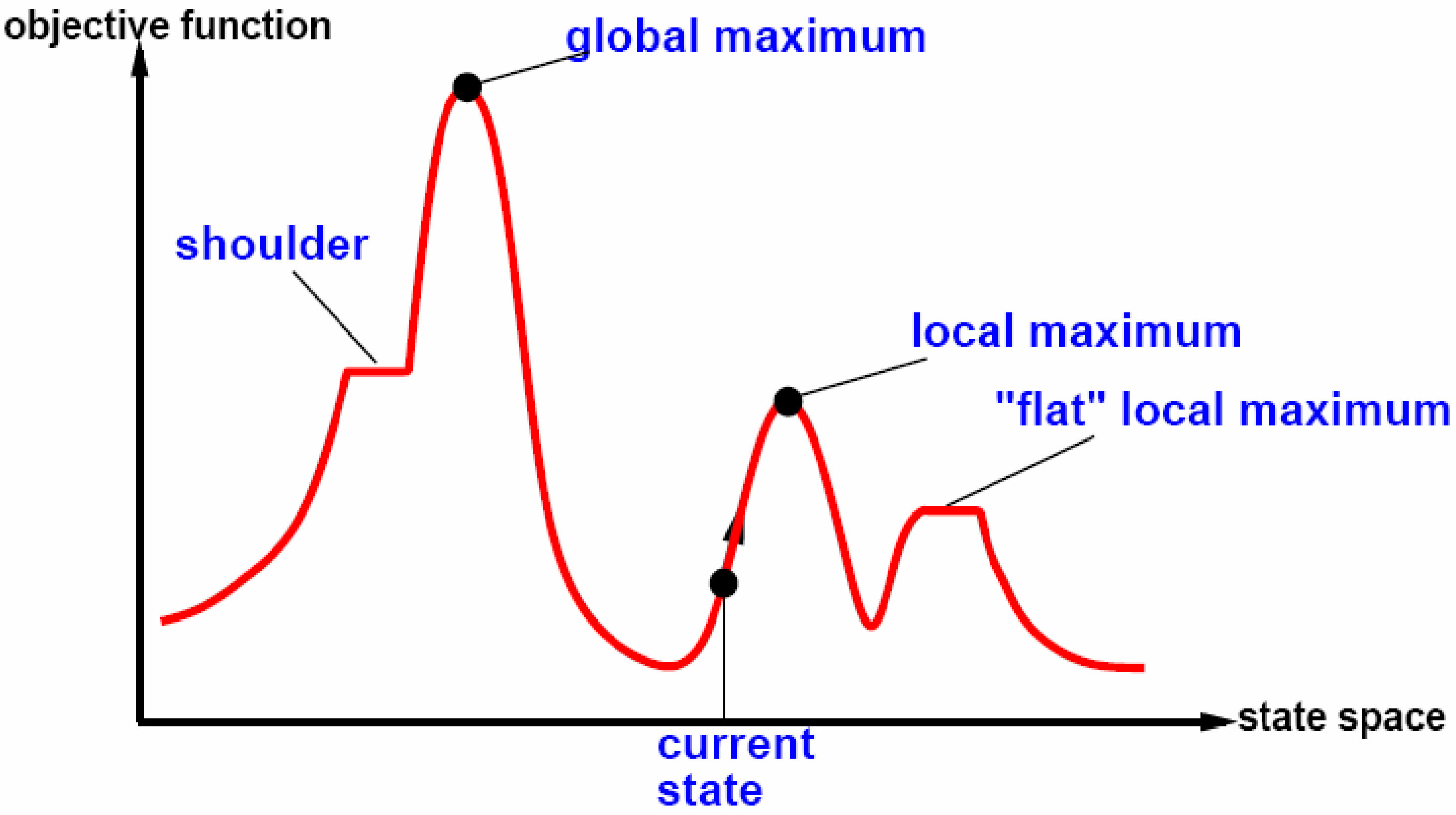
"Like climbing Everest in thick fog with amnesia"

```
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
    end
```

Hill-climbing contd.

Useful to consider state space landscape



Random-restart hill climbing overcomes local maxima—trivially complete

Random sideways moves 😊escape from shoulders ☹loop on flat maxima

Simulated annealing

Idea: escape local maxima by allowing some "bad" moves
but gradually decrease their size and frequency

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     $T$, a "temperature" controlling prob. of downward steps

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

Local beam search

Idea: keep k states instead of 1; choose top k of all their successors

Not the same as k searches run in parallel!
Searches that find good states recruit other searches to join them

Problem: quite often, all k states end up on same local hill

Idea: choose k successors randomly, biased towards good ones

Observe the close analogy to natural selection!

Can evolution be intelligent?

- Intelligence can be defined as the capability of a system to adapt its behaviour to ever-changing environment. According to Alan Turing, the form or appearance of a system is irrelevant to its intelligence.
- Evolutionary computation simulates evolution on a computer. The result of such a simulation is a series of optimisation algorithms, usually based on a simple set of rules. Optimisation iteratively improves the quality of solutions until an optimal, or at least feasible, found.

- The behaviour of an individual organism is an inductive inference about some yet unknown aspects of its environment. If, over successive generations, the organism survives, we can say that this organism is capable of learning to predict changes in its environment.
- The evolutionary approach is based on computational models of natural selection and genetics. We call them **evolutionary computation**, an umbrella term that combines **genetic algorithms**, **evolution strategies** and **genetic programming**.

How is a population with increasing fitness generated?

- Let us consider a population of rabbits. Some rabbits are faster than others, and we may say that these rabbits possess superior fitness, because they have a greater chance of avoiding foxes, surviving and then breeding.
- If two parents have superior fitness, there is a good chance that a combination of their genes will produce an offspring with even higher fitness. Over time the entire population of rabbits becomes faster to meet their environmental challenges in the face of foxes.

Simulation of natural evolution

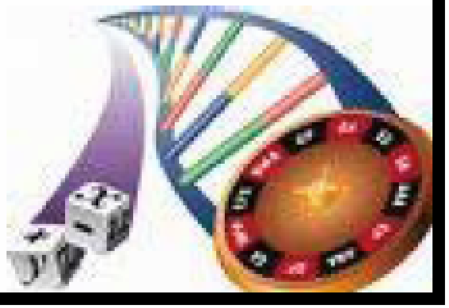
- All methods of evolutionary computation simulate natural evolution by creating a population of individuals, evaluating their fitness, generating a new population through genetic operations, and repeating this process a number of times.
- We will start with **Genetic Algorithms** (GAs) as most of the other evolutionary algorithms can be viewed as variations of genetic algorithms.

Genetic Algorithms

- In the early 1970s, John Holland introduced the concept of genetic algorithms.
- His aim was to make computers do what nature does. Holland was concerned with algorithms that manipulate strings of binary digits.
- Each artificial “chromosome” consists of a number of “genes”, and each gene is represented by 0 or 1:

1	0	1	1	0	1	0	0	0	0	0	1	0	1	0	1

- Nature has an ability to adapt and learn without being told what to do. In other words, nature finds good chromosomes blindly. GAs do the same. Two mechanisms link a GA to the problem it is solving: **encoding** and **evaluation**.
- The GA uses a measure of fitness of individual chromosomes to carry out reproduction. As reproduction takes place, the crossover operator exchanges parts of two single chromosomes, and the mutation operator changes the gene value in some randomly chosen location of the chromosome.



Basic genetic algorithms

Step 1: Represent the problem variable domain as a chromosome of a fixed length, choose the size of a chromosome population N, the crossover probability p_c and the mutation probability p_m.

Step 2: Define a fitness function to measure the performance, or fitness, of an individual chromosome in the problem domain. The fitness function establishes the basis for selecting chromosomes that will be mated during reproduction.

Step 3: Randomly generate an initial population of chromosomes of size N:

$x_1, x_2, \ldots, x_N$

Step 4: Calculate the fitness of each individual chromosome:

$f(x_1), f(x_2), \ldots, f(x_N)$

Step 5: Select a pair of chromosomes for mating from the current population. Parent chromosomes are selected with a probability related to their fitness.

Step 6: Create a pair of offspring chromosomes by applying the genetic operators – **crossover** and **mutation**.

Step 7: Place the created offspring chromosomes in the new population.

Step 8: Repeat *Step 5* until the size of the new chromosome population becomes equal to the size of the initial population, *N*.

Step 9: Replace the initial (parent) chromosome population with the new (offspring) population.

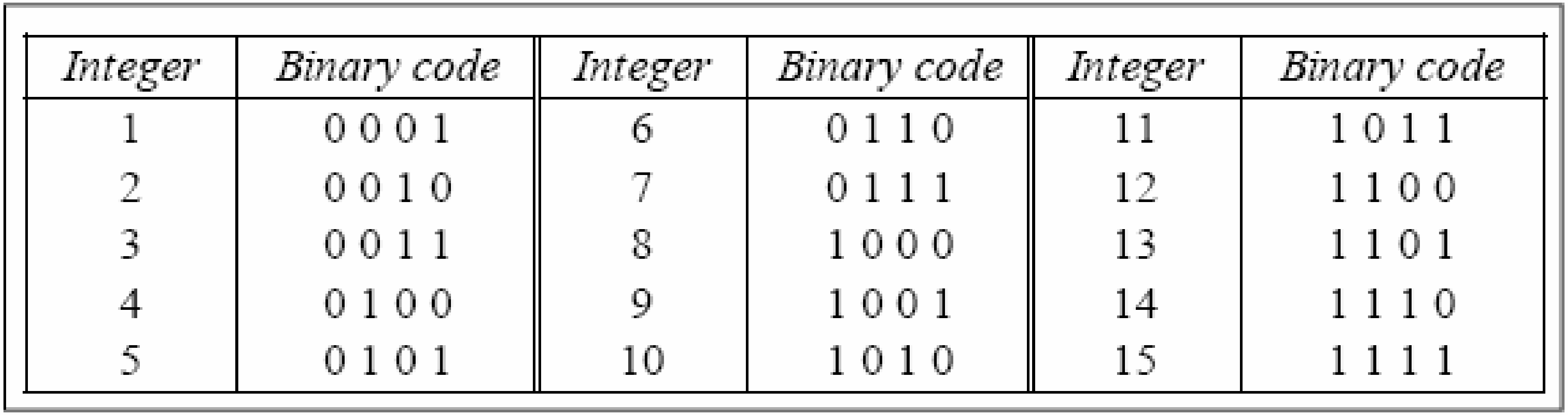
Step 10: Go to *Step 4*, and repeat the process until the termination criterion is satisfied.

Genetic algorithms

- GA represents an iterative process. Each iteration is called a **generation**. A typical number of generations for a simple GA can range from 50 to over 500. The entire set of generations is called a **run**.
- Because GAs use a stochastic search method, the fitness of a population may remain stable for a number of generations before a superior chromosome appears.
- A common practice is to terminate a GA after a specified number of generations and then examine the best chromosomes in the population. If no satisfactory solution is found, the GA is restarted.

Genetic algorithms: case study

A simple example will help us to understand how a GA works. Let us find the maximum value of the function $(15x - x^2)$ where parameter x varies between 0 and 15. For simplicity, we may assume that x takes only integer values. Thus, chromosomes can be built with only four genes:



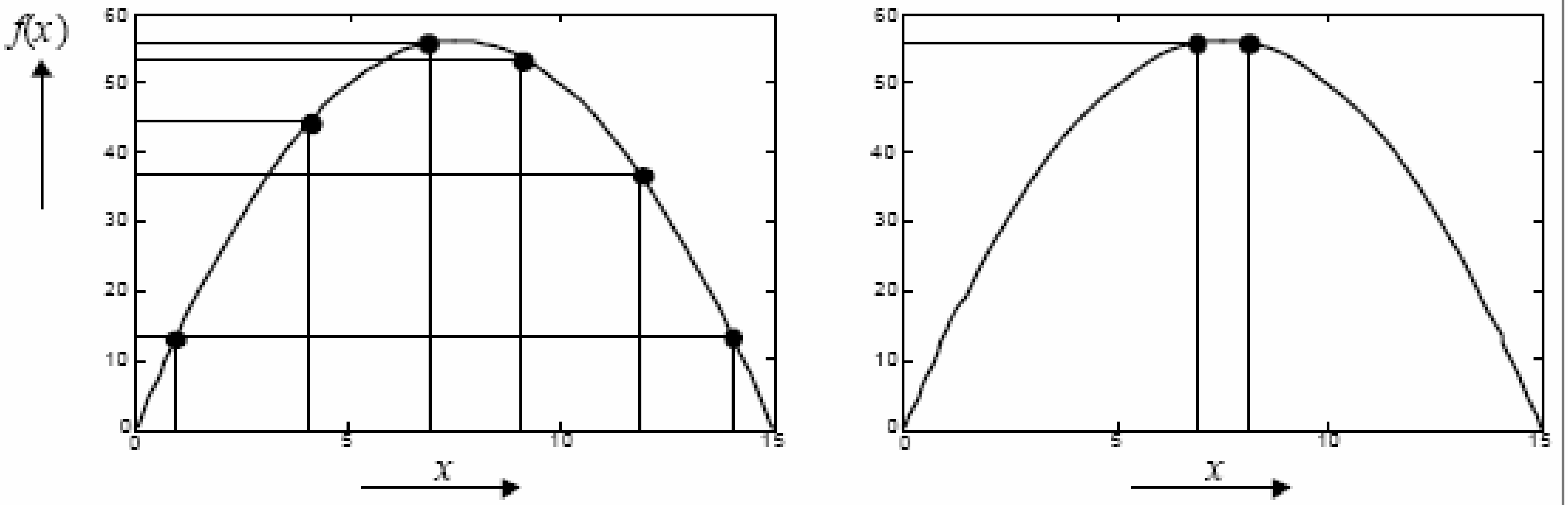
Integer	*Binary code*	*Integer*	*Binary code*	*Integer*	*Binary code*
1	0 0 0 1	6	0 1 1 0	11	1 0 1 1
2	0 0 1 0	7	0 1 1 1	12	1 1 0 0
3	0 0 1 1	8	1 0 0 0	13	1 1 0 1
4	0 1 0 0	9	1 0 0 1	14	1 1 1 0
5	0 1 0 1	10	1 0 1 0	15	1 1 1 1

Suppose that the size of the chromosome population N is 6, the crossover probability p_c equals 0.7, and the mutation probability p_m equals 0.001. The fitness function in our example is defined by

$$f(x) = 15\,x - x^2$$

The fitness function and chromosome locations

Chromosome label	*Chromosome string*	*Decoded integer*	*Chromosome fitness*	*Fitness ratio, %*
X1	1 1 0 0	12	36	16.5
X2	0 1 0 0	4	44	20.2
X3	0 0 0 1	1	14	6.4
X4	1 1 1 0	14	14	6.4
X5	0 1 1 1	7	56	25.7
X6	1 0 0 1	9	54	24.8



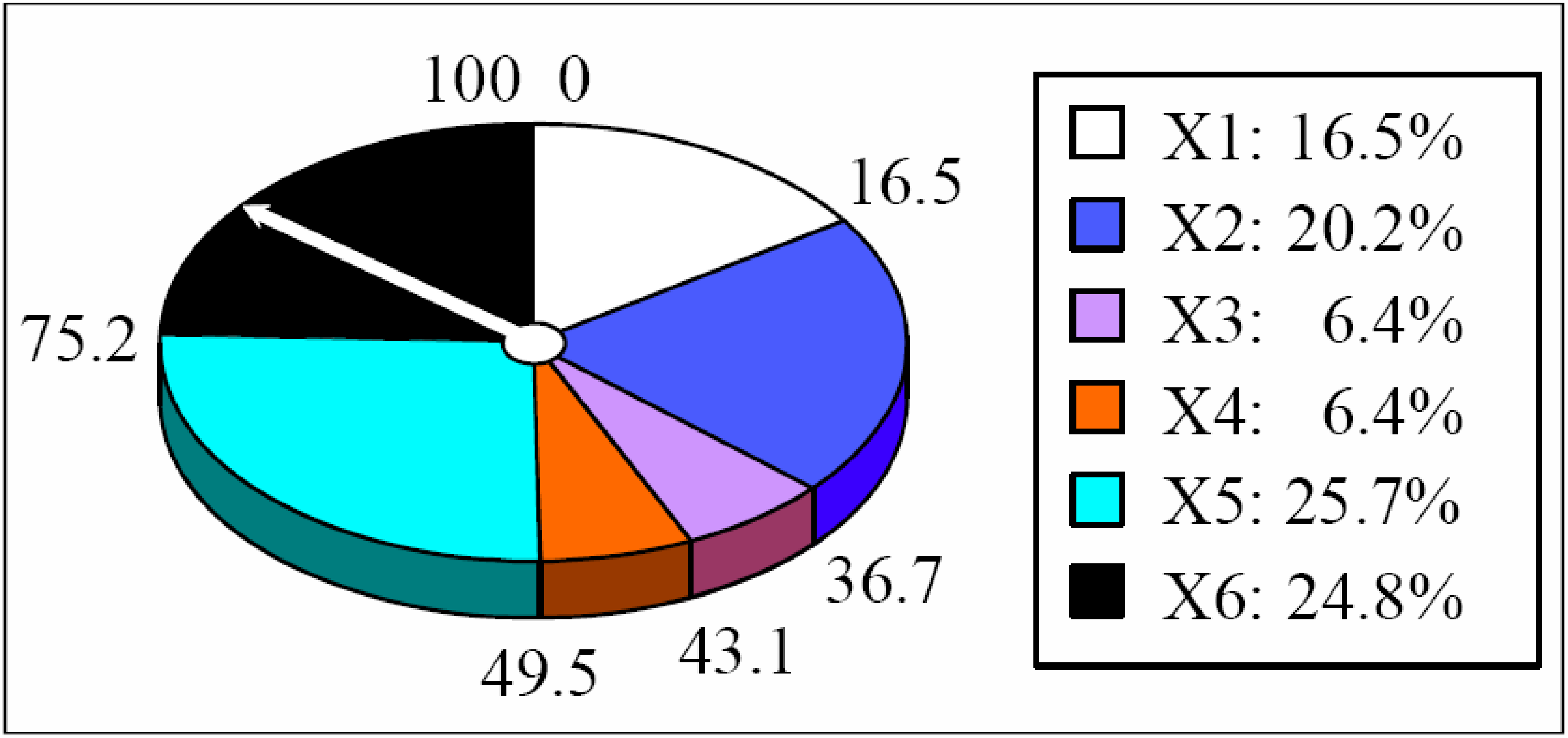
(*a*) Chromosome initial locations.

(*b*) Chromosome final locations.

- In natural selection, only the fittest species can survive, breed, and thereby pass their genes on to the next generation. GAs use a similar approach, but unlike nature, the size of the chromosome population remains unchanged from one generation to the next.
- The last column in Table shows the ratio of the individual chromosome’s fitness to the population’s total fitness. This ratio determines the chromosome’s chance of being selected for mating. The chromosome’s average fitness improves from one generation to the next.

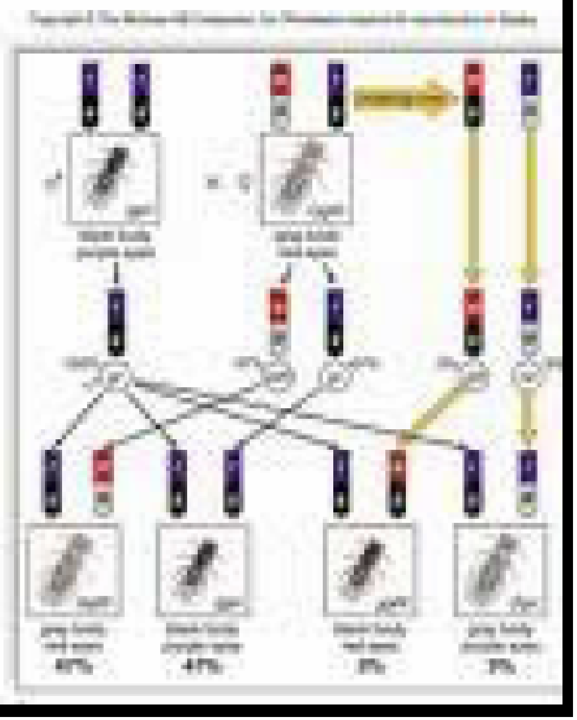
Roulette wheel selection

The most commonly used chromosome selection techniques is the **roulette wheel selection**.



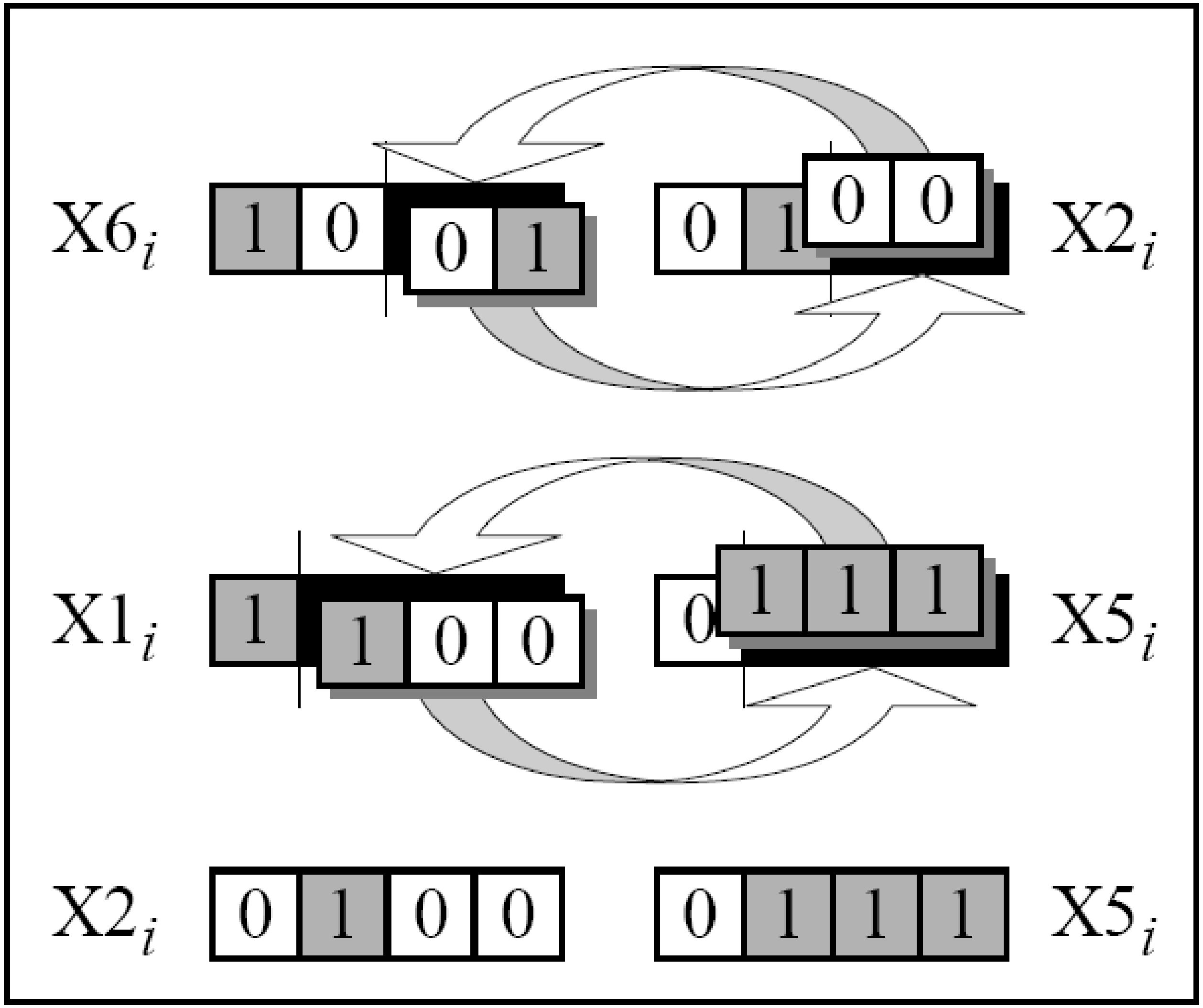
Crossover operator

- In our example, we have an initial population of 6 chromosomes. Thus, to establish the same population in the next generation, the roulette wheel would be spun six times.
- Once a pair of parent chromosomes is selected, the **crossover** operator is applied.



- First, the crossover operator randomly chooses a crossover point where two parent chromosomes “break”, and then exchanges the chromosome parts after that point. As a result, two new offspring are created.
- If a pair of chromosomes does not cross over, then the chromosome cloning takes place, and the offspring are created as exact copies of each parent.

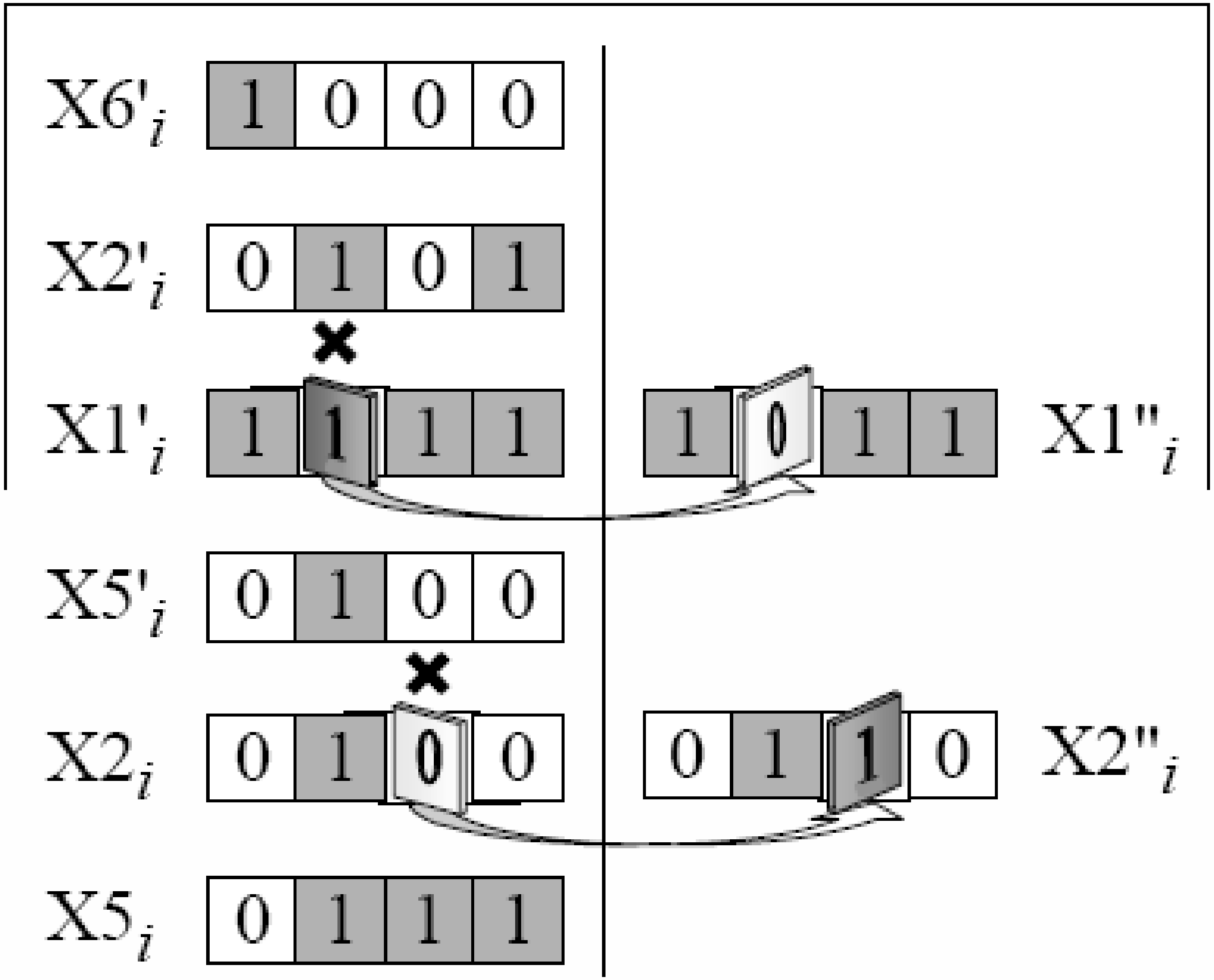
Crossover



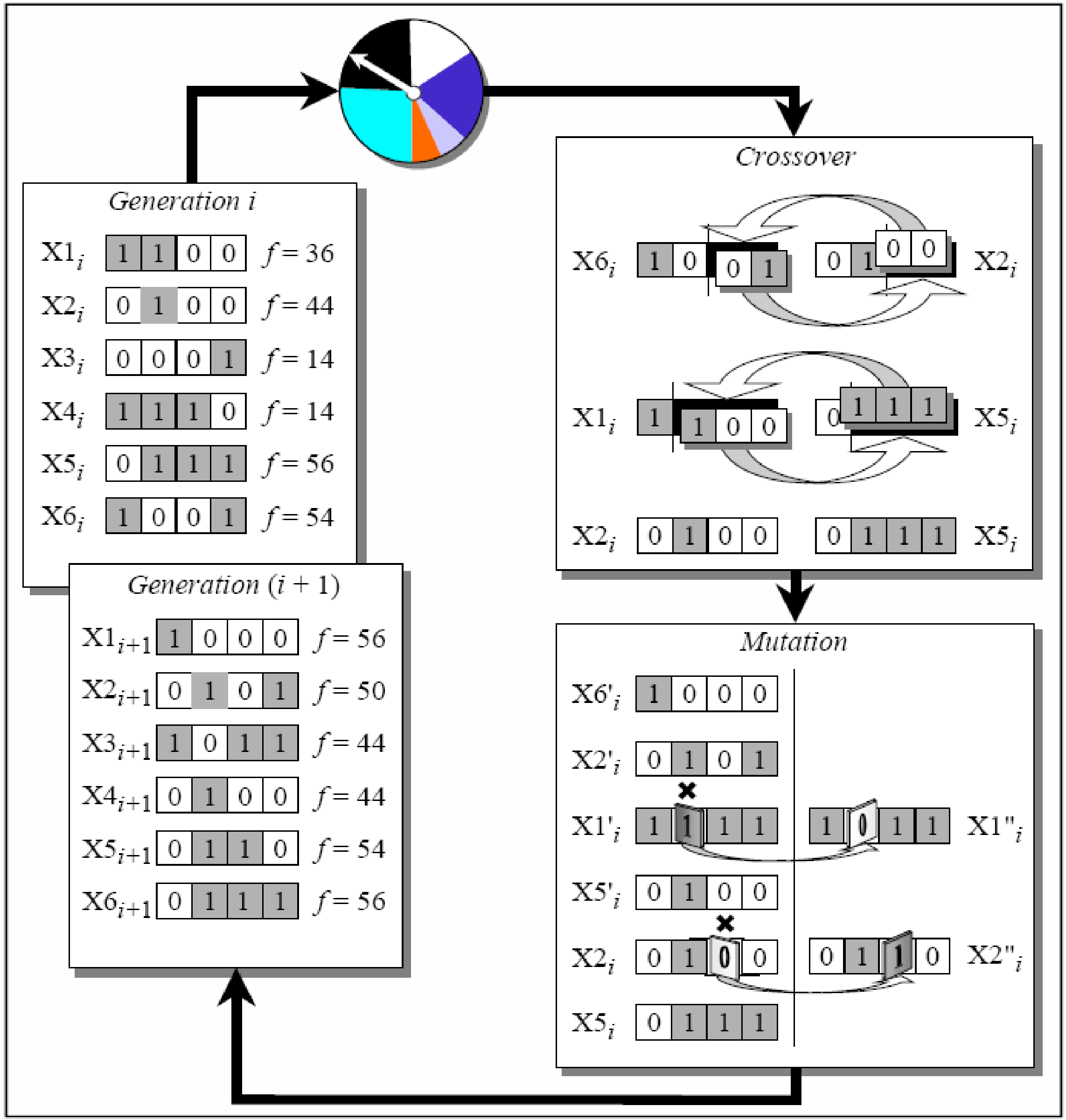
Mutation operator

- Mutation represents a change in the gene.
- Mutation is a background operator. Its role is to provide a guarantee that the search algorithm is not trapped on a local optimum.
- The mutation operator flips a randomly selected gene in a chromosome.
- The mutation probability is quite small in nature, and is kept low for GAs, typically in the range between 0.001 and 0.01.

Mutation


X6'ᵢ 1 0 0 0
X2'ᵢ 0 1 0 1
X1'ᵢ 1 1 1 1
1 0 1 1 X1"ᵢ
X5'ᵢ 0 1 0 0
X2ᵢ 0 1 0 0
0 1 1 0 X2"ᵢ
X5ᵢ 0 1 1 1

The genetic algorithm cycle



Generation i

$X1_i$	1	1	0	0	$f = 36$
$X2_i$	0	1	0	0	$f = 44$
$X3_i$	0	0	0	1	$f = 14$
$X4_i$	1	1	1	0	$f = 14$
$X5_i$	0	1	1	1	$f = 56$
$X6_i$	1	0	0	1	$f = 54$

Crossover

$X6_i$ 1 0 0 1 — 0 1 0 0 $X2_i$

$X1_i$ 1 1 0 0 — 0 1 1 1 $X5_i$

$X2_i$ 0 1 0 0 — 0 1 1 1 $X5_i$

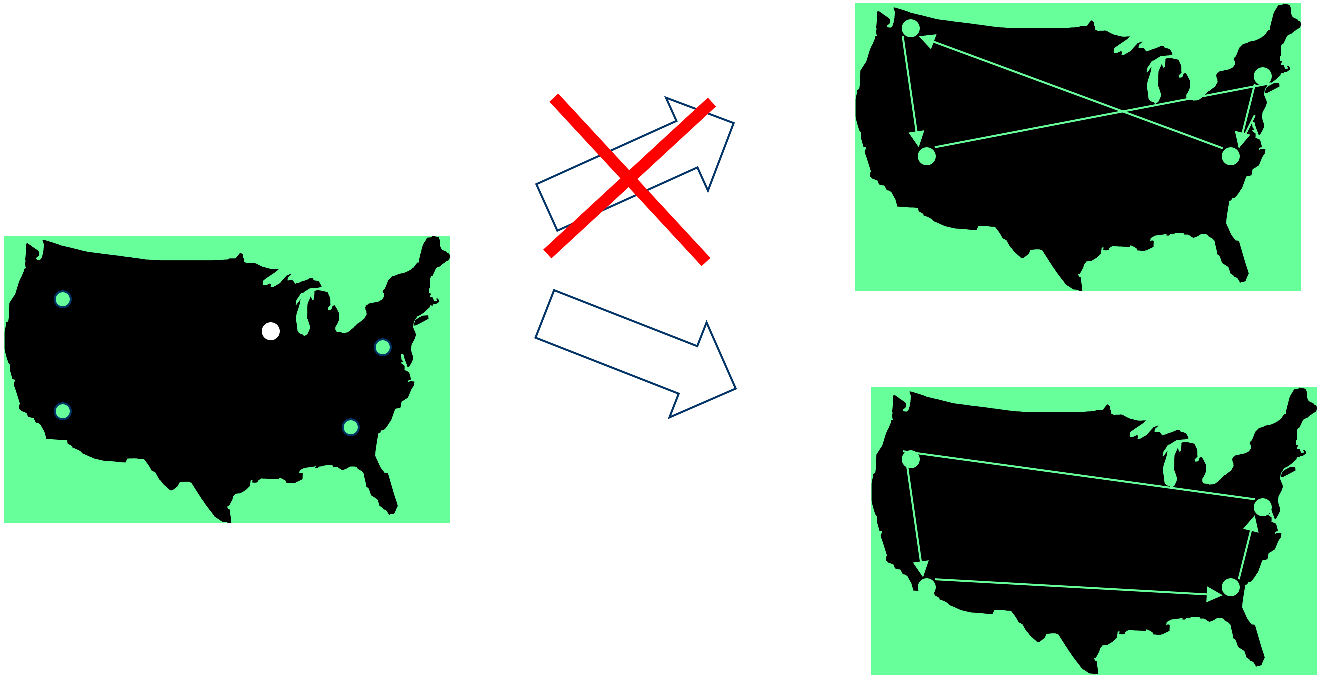
Mutation

$X6'_i$	1	0	0	0						
$X2'_i$	0	1	0	1						
$X1'_i$	1	1	1	1	1	0	1	1	$X1''_i$
$X5'_i$	0	1	0	0						
$X2_i$	0	1	0	0	0	1	1	0	$X2''_i$
$X5_i$	0	1	1	1						

Generation (i + 1)

$X1_{i+1}$	1	0	0	0	$f = 56$
$X2_{i+1}$	0	1	0	1	$f = 50$
$X3_{i+1}$	1	0	1	1	$f = 44$
$X4_{i+1}$	0	1	0	0	$f = 44$
$X5_{i+1}$	0	1	1	0	$f = 54$
$X6_{i+1}$	0	1	1	1	$f = 56$

TSP Example



Representation

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

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

CityList1 (3 5 7 2 1 6 4 8)
CityList2 (2 5 7 6 8 1 3 4)

Initial Population for TSP

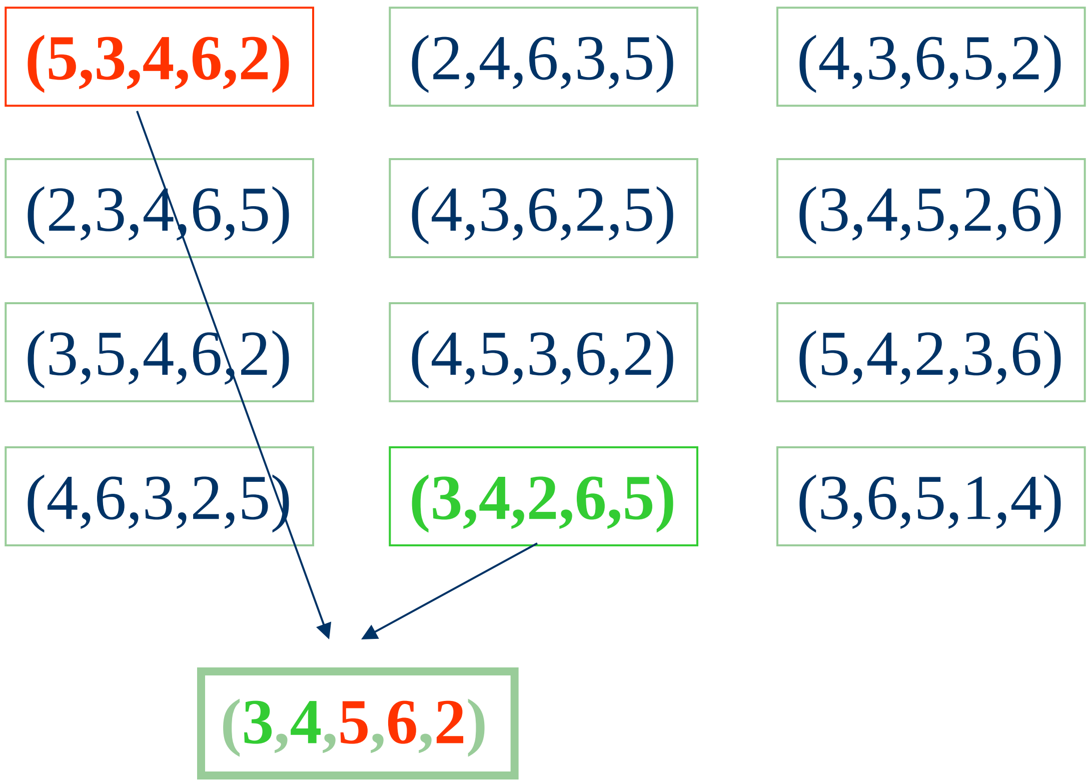
(5,3,4,6,2)	(2,4,6,3,5)	(4,3,6,5,2)
(2,3,4,6,5)	(4,3,6,2,5)	(3,4,5,2,6)
(3,5,4,6,2)	(4,5,3,6,2)	(5,4,2,3,6)
(4,6,3,2,5)	(3,4,2,6,5)	(3,6,5,1,4)

Select Parents

(5,3,4,6,2)	(2,4,6,3,5)	(4,3,6,5,2)
(2,3,4,6,5)	(4,3,6,2,5)	(3,4,5,2,6)
(3,5,4,6,2)	(4,5,3,6,2)	(5,4,2,3,6)
(4,6,3,2,5)	**(3,4,2,6,5)**	(3,6,5,1,4)

Try to pick the better ones.

Create Off-Spring – 1 point



(5,3,4,6,2) (2,4,6,3,5) (4,3,6,5,2)

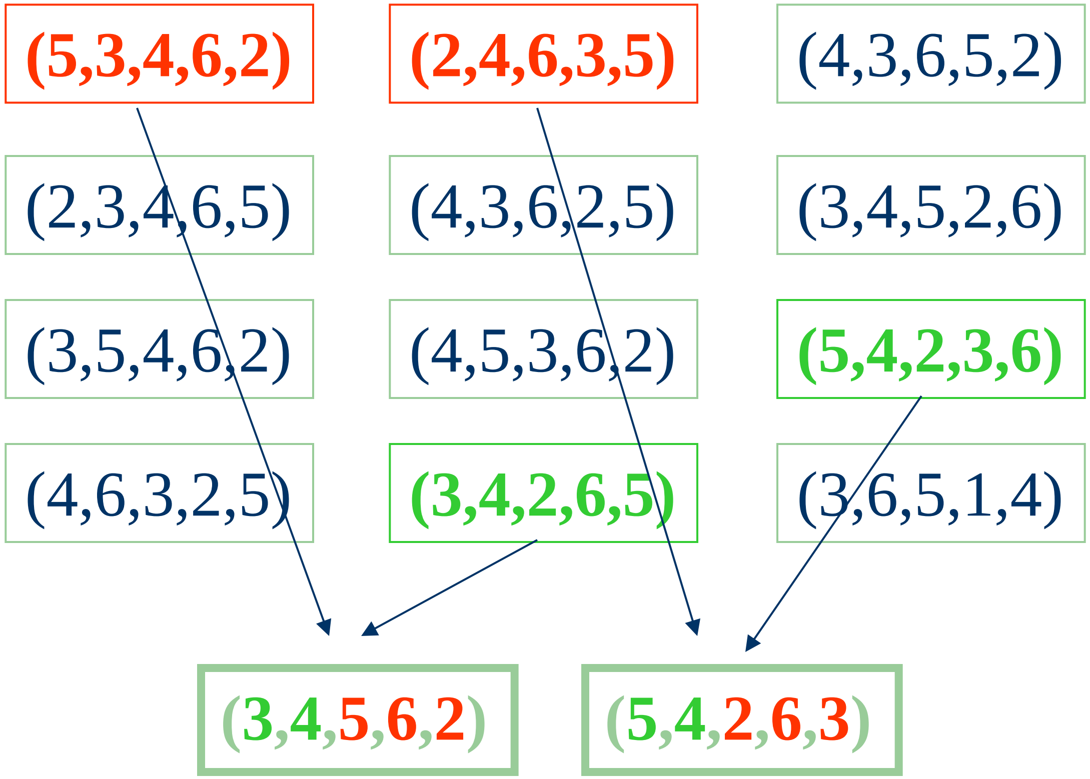
(2,3,4,6,5) (4,3,6,2,5) (3,4,5,2,6)

(3,5,4,6,2) (4,5,3,6,2) (5,4,2,3,6)

(4,6,3,2,5) **(3,4,2,6,5)** (3,6,5,1,4)

(3,4,5,6,2)

Create More Offspring



(5,3,4,6,2)	**(2,4,6,3,5)**	(4,3,6,5,2)
(2,3,4,6,5)	(4,3,6,2,5)	(3,4,5,2,6)
(3,5,4,6,2)	(4,5,3,6,2)	**(5,4,2,3,6)**
(4,6,3,2,5)	**(3,4,2,6,5)**	(3,6,5,1,4)

(3,4,5,6,2) **(5,4,2,6,3)**

Mutate

(5,3,4,6,2)	(2,4,6,3,5)	(4,3,6,5,2)
(2,3,4,6,5)	(4,3,6,2,5)	(3,4,5,2,6)
(3,5,4,6,2)	(4,5,3,6,2)	(5,4,2,3,6)
(4,6,3,2,5)	(3,4,2,6,5)	(3,6,5,1,4)

(3,4,5,6,2)

(5,4,2,6,3)

Mutate

(5,3,4,6,2)	(2,4,6,3,5)	(4,3,6,5,2)
(2,3,4,6,5)	(2,3,6,4,5)	(3,4,5,2,6)
(3,5,4,6,2)	(4,5,3,6,2)	(5,4,2,3,6)
(4,6,3,2,5)	(3,4,2,6,5)	(3,6,5,1,4)

(3,4,5,6,2) (5,4,2,6,3)

Eliminate

(5,3,4,6,2)	**(2,4,6,3,5)**	~~(4,3,6,5,2)~~
~~(2,3,4,6,5)~~	**(2,3,6,4,5)**	(3,4,5,2,6)
(3,5,4,6,2)	(4,5,3,6,2)	**(5,4,2,3,6)**
(4,6,3,2,5)	**(3,4,2,6,5)**	(3,6,5,1,4)

(3,4,5,6,2) **(5,4,2,6,3)**

Tend to kill off the worst ones.

Integrate

(5,3,4,6,2)	**(2,4,6,3,5)**	**(5,4,2,6,3)**
(3,4,5,6,2)	**(2,3,6,4,5)**	(3,4,5,2,6)
(3,5,4,6,2)	(4,5,3,6,2)	**(5,4,2,3,6)**
(4,6,3,2,5)	**(3,4,2,6,5)**	(3,6,5,1,4)

Restart

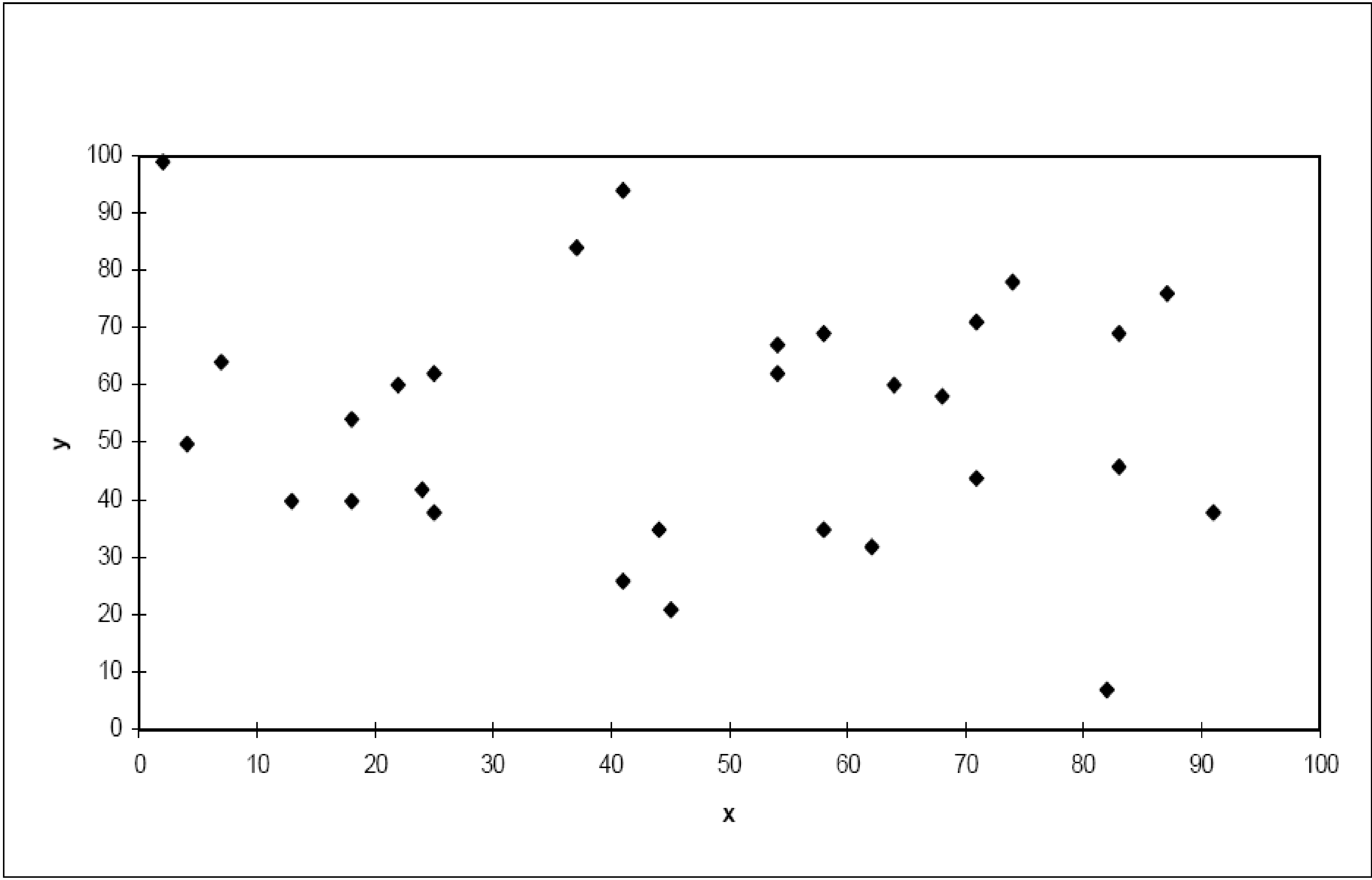
(5,3,4,6,2) (2,4,6,3,5) (5,4,2,6,3)

(3,4,5,6,2) (2,3,6,4,5) (3,4,5,2,6)

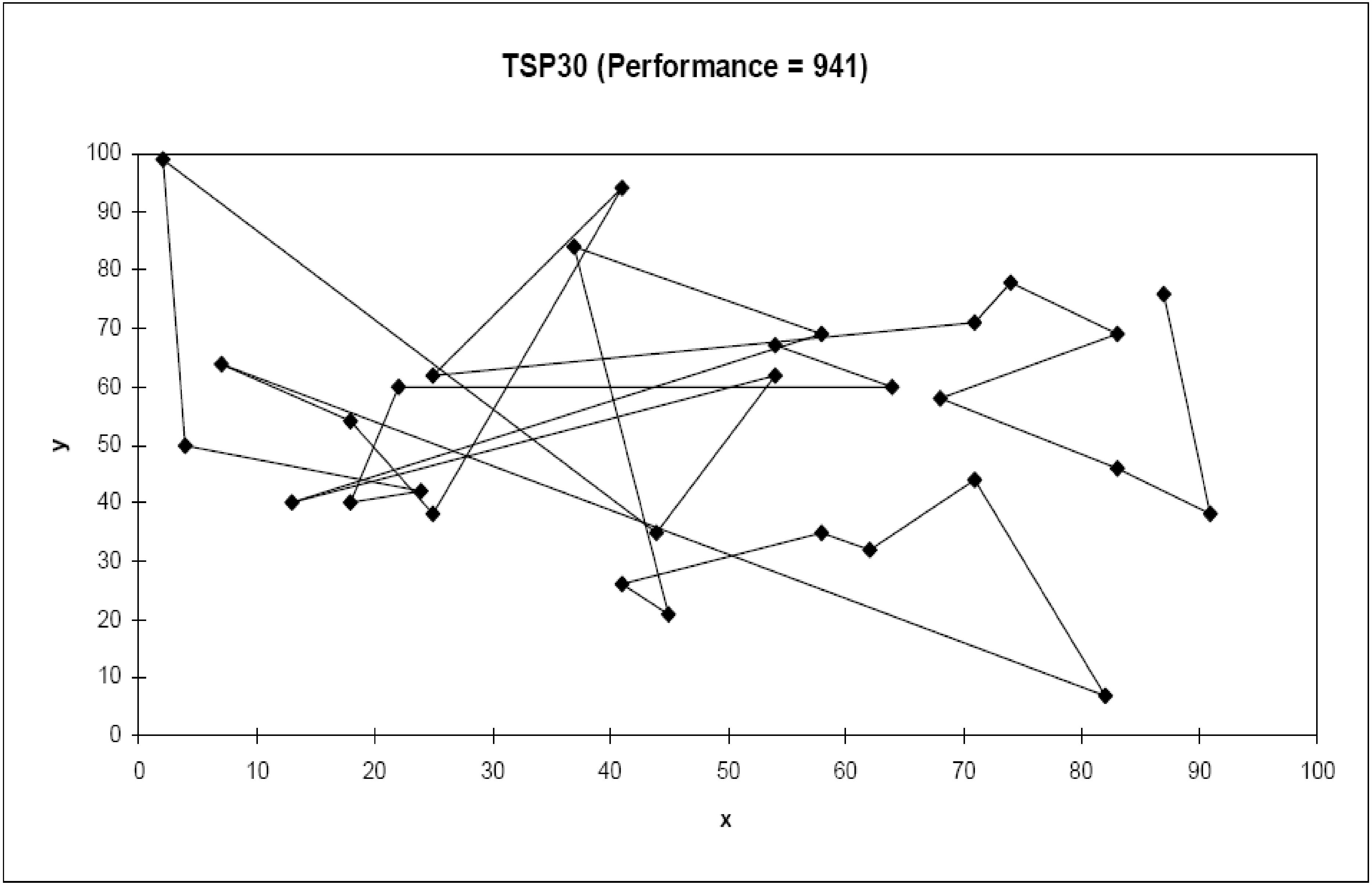
(3,5,4,6,2) (4,5,3,6,2) (5,4,2,3,6)

(4,6,3,2,5) (3,4,2,6,5) (3,6,5,1,4)

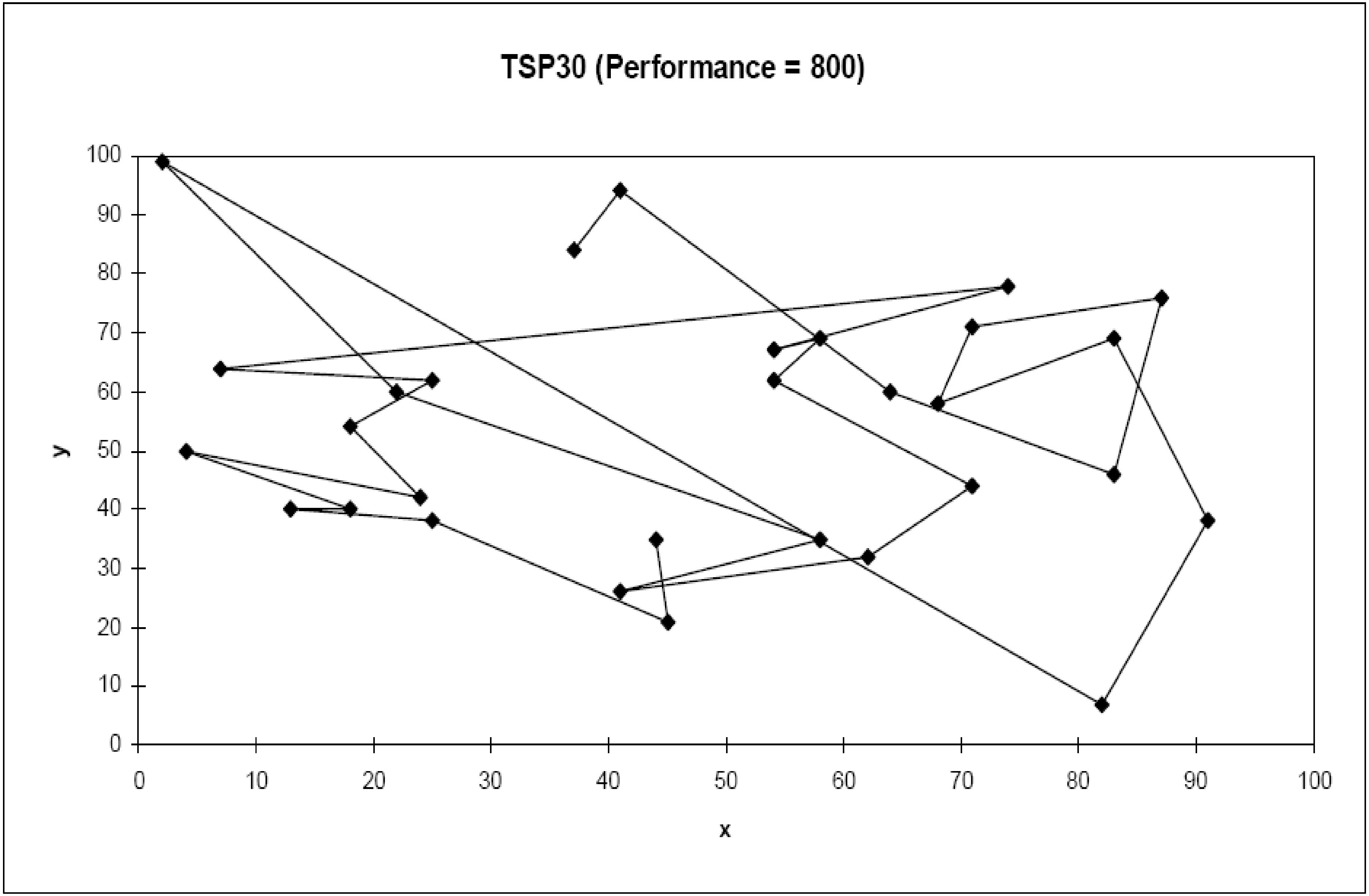
TSP Example: 30 Cities



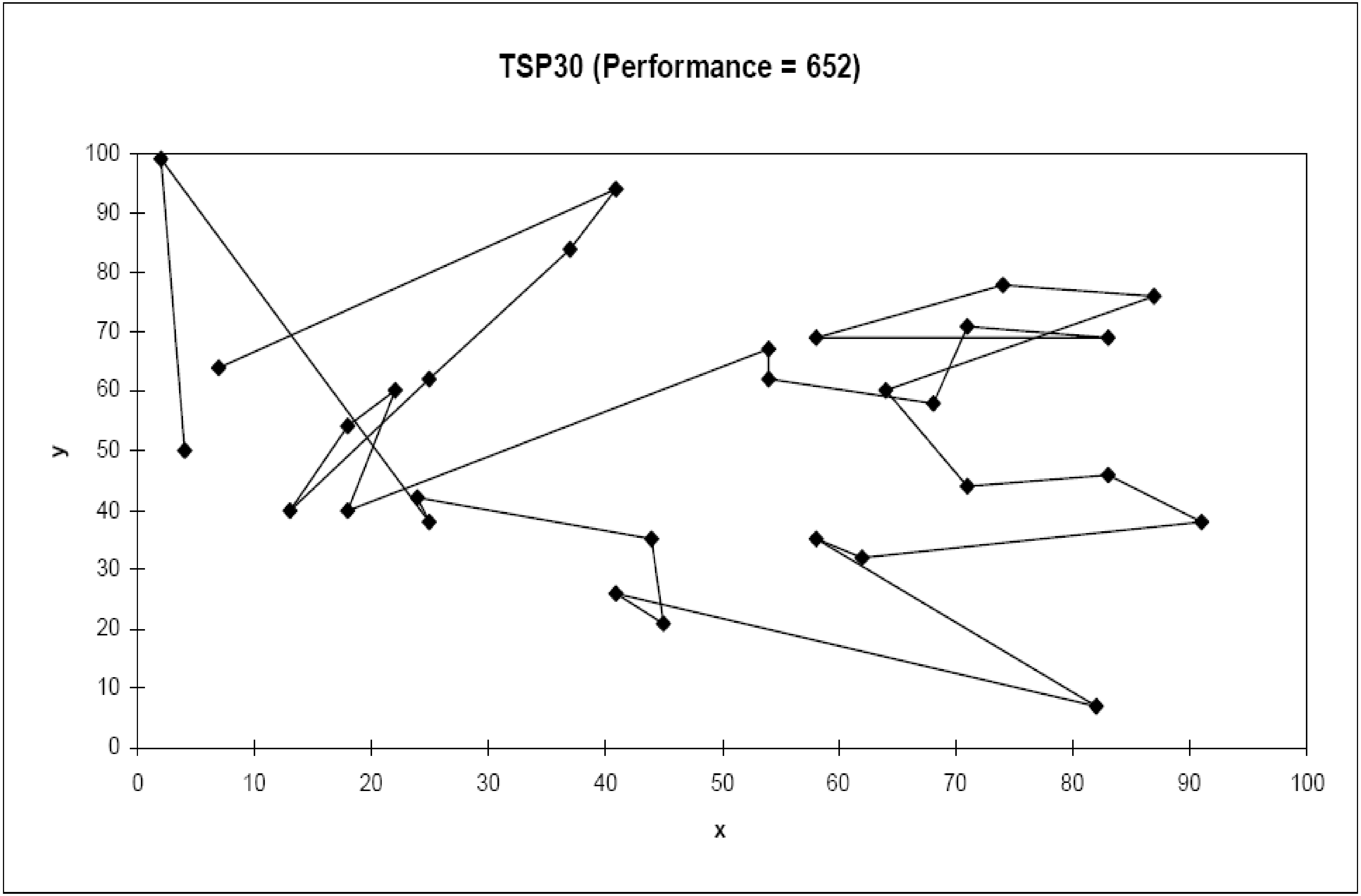
Solution $_i$ (Distance = 941)



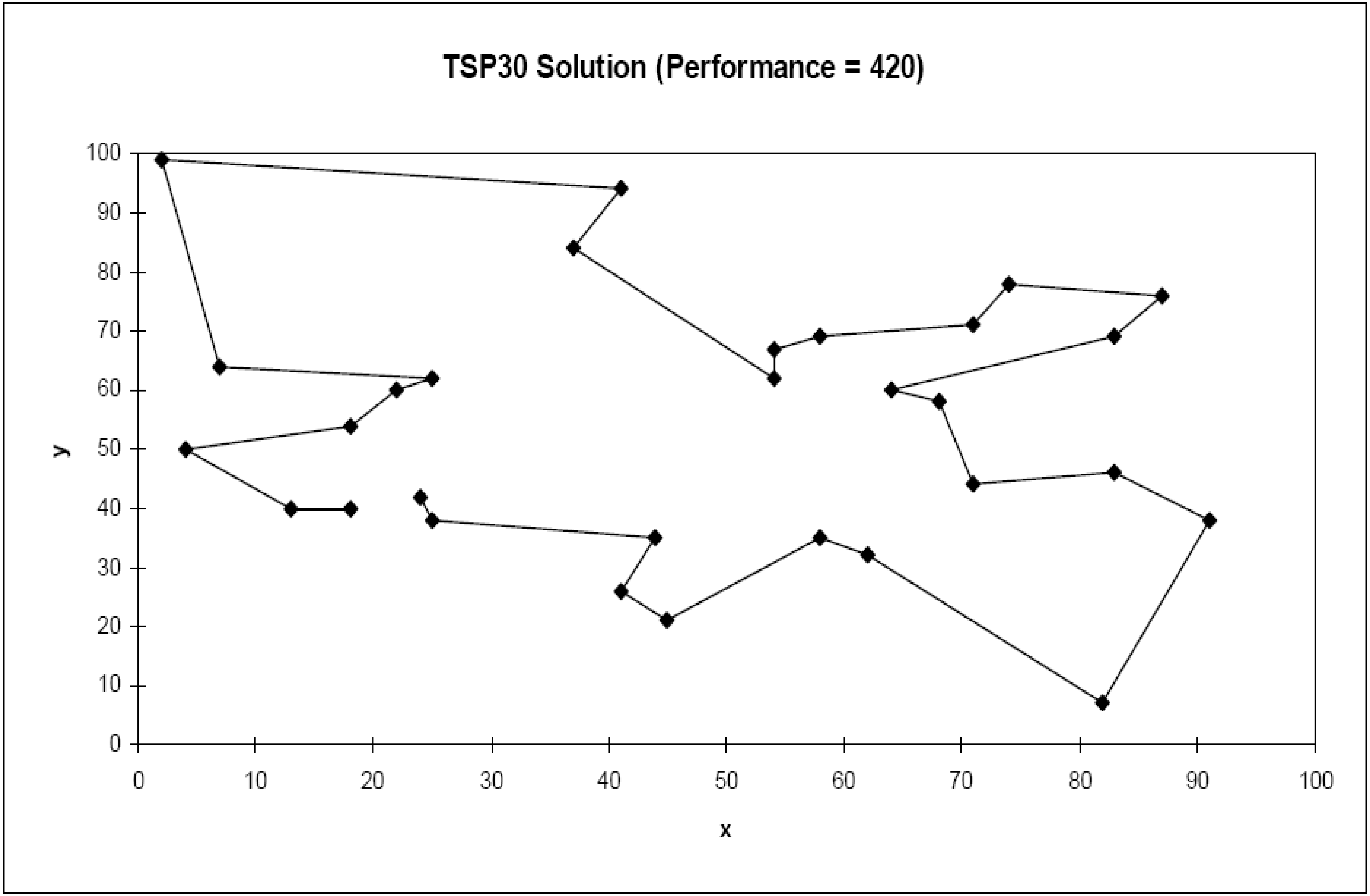
Solution $_j$(Distance = 800)

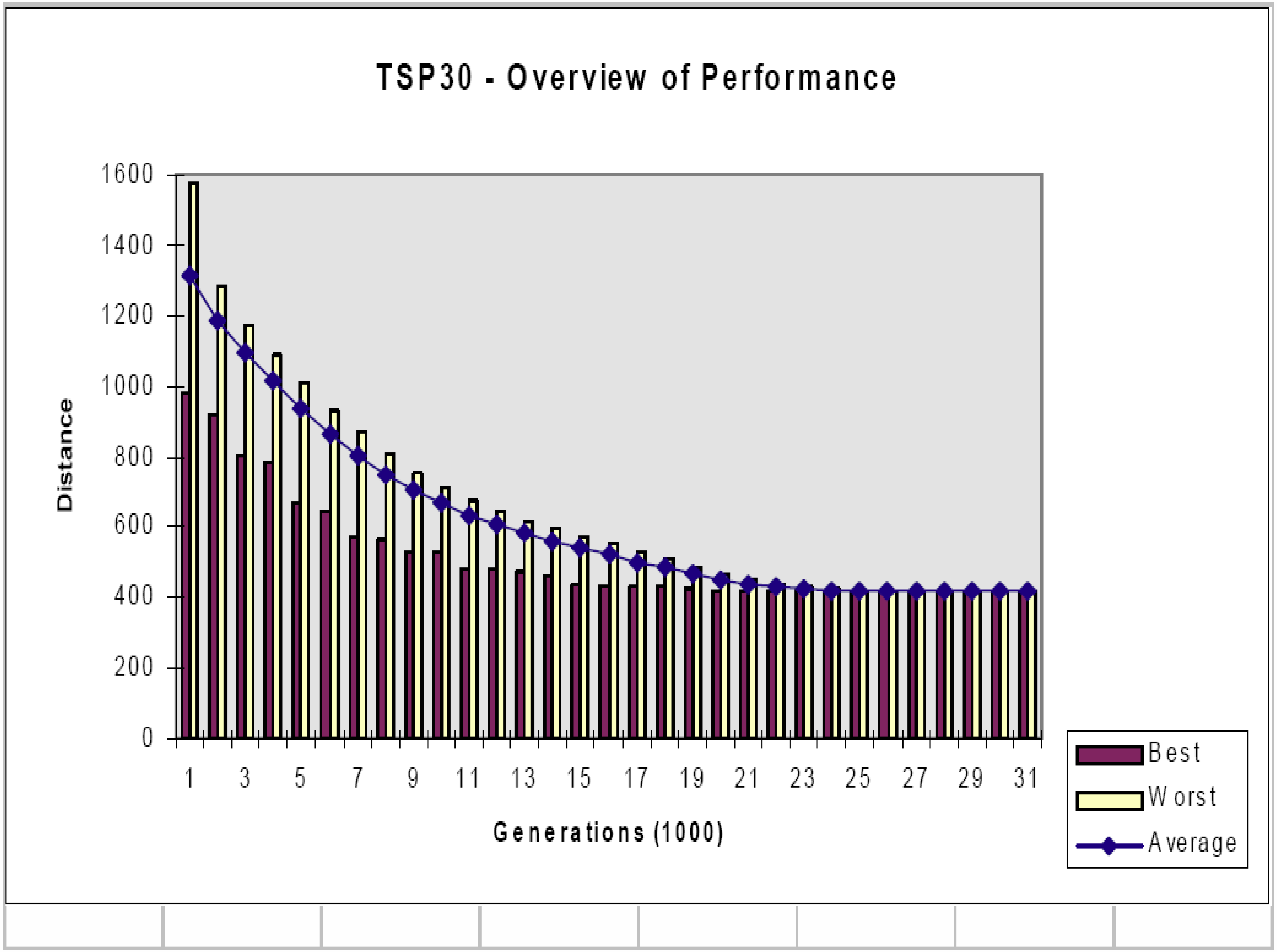


Solution $_k$(Distance = 652)



Best Solution (Distance = 420)




TSP30 - Overview of Performance
Distance
1600
1400
1200
1000
800
600
400
200
0
1
3
5
7
9
11
13
15
17
19
21
23
25
27
29
31
Generations (1000)
Best
Worst
Average

Components of a GA

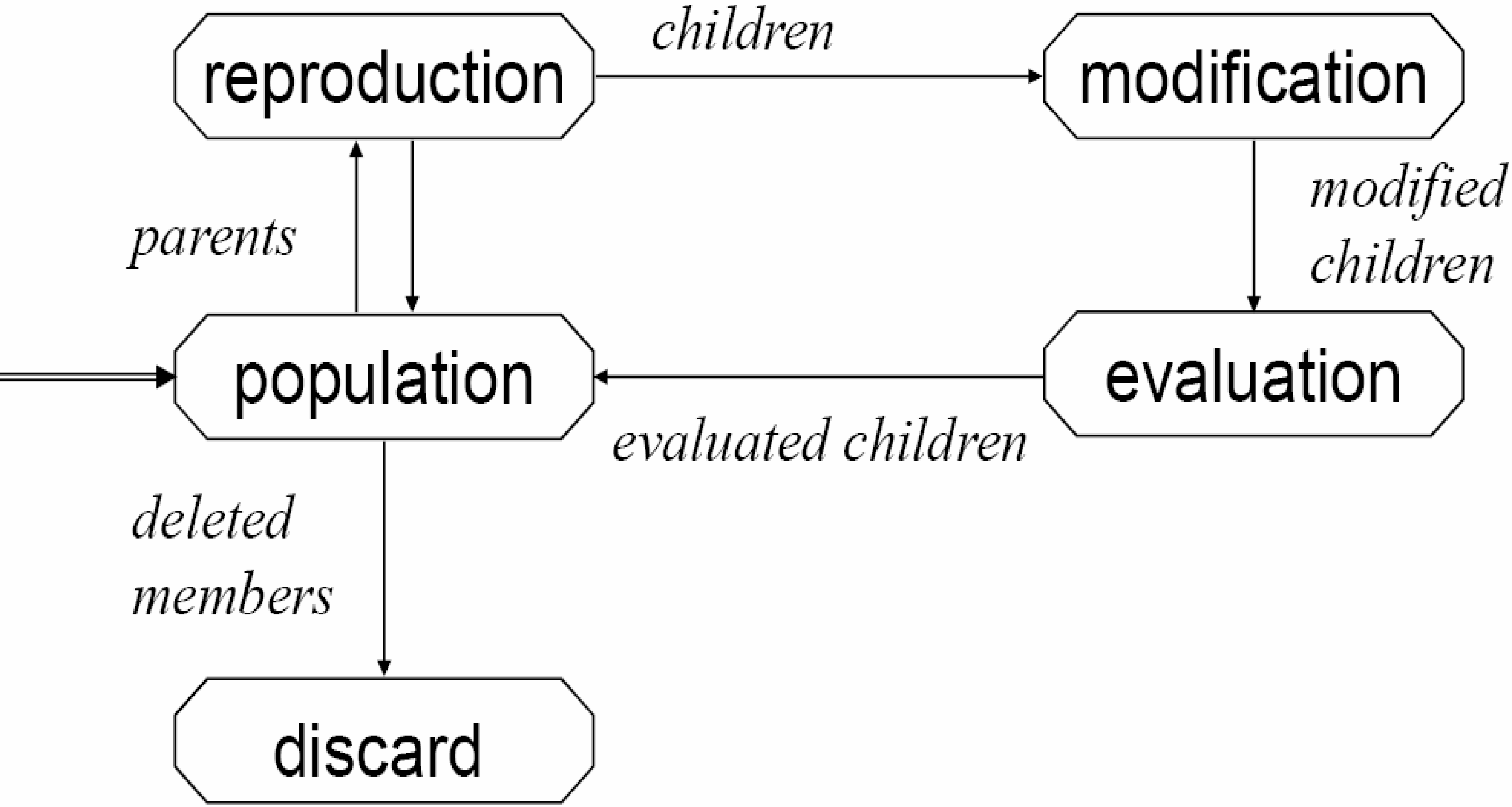
A problem to solve, and ...

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

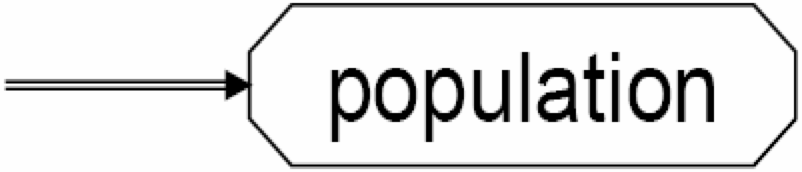
Simple Genetic Algorithm

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

The GA Cycle of Reproduction



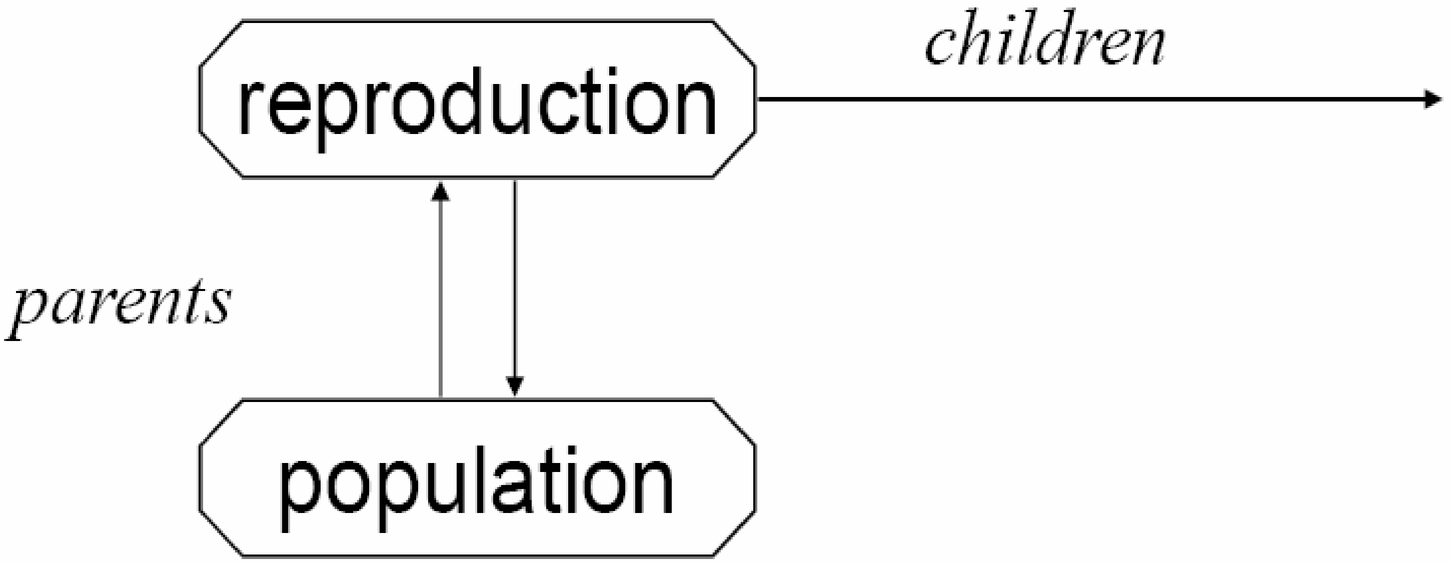
Population



Chromosomes could be:

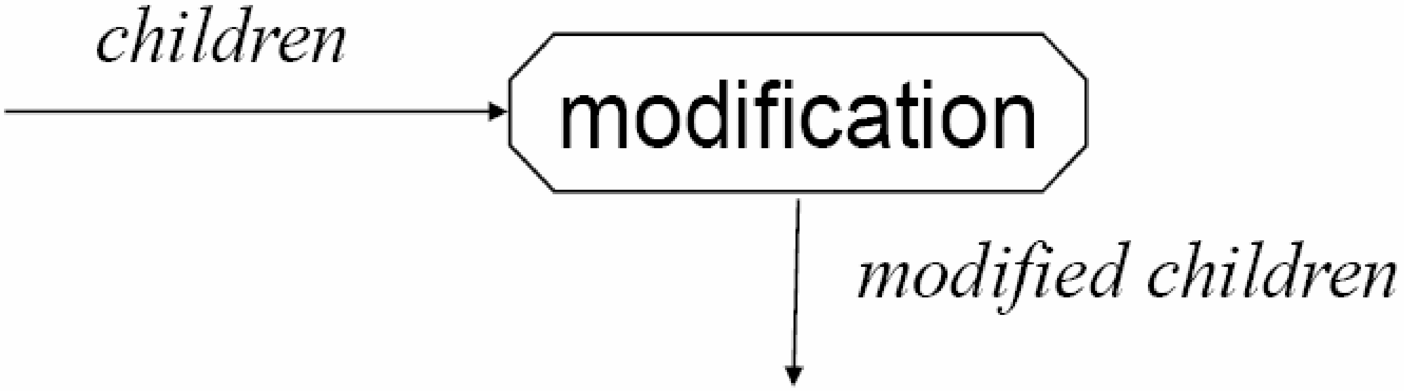
♦ Bit strings	(0101 ... 1100)
♦ Real numbers	(43.2 -33.1 ... 0.0 89.2)
♦ Permutations of element	(E11 E3 E7 ... E1 E15)
♦ Lists of rules	(R1 R2 R3 ... R22 R23)
♦ Program elements	(genetic programming)
♦ ... any data structure ...	

Reproduction



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

Chromosome Modification



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

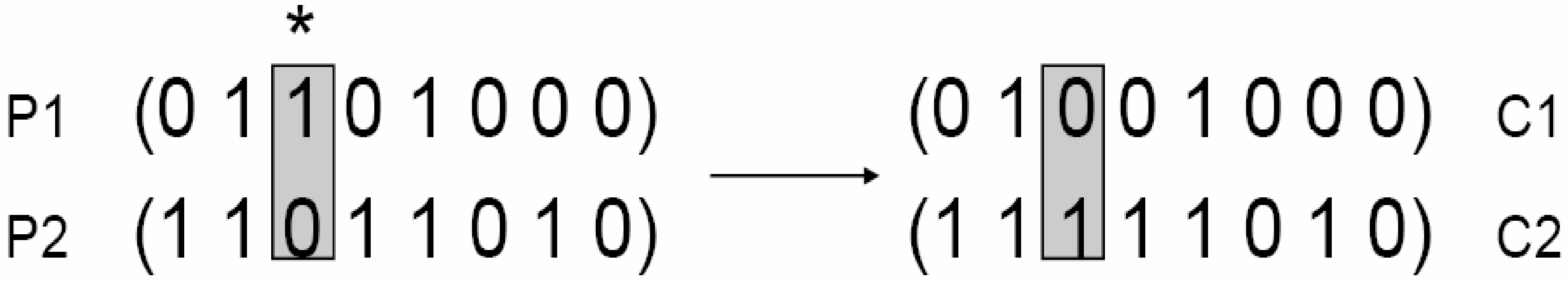
Mutation: Local Modification

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

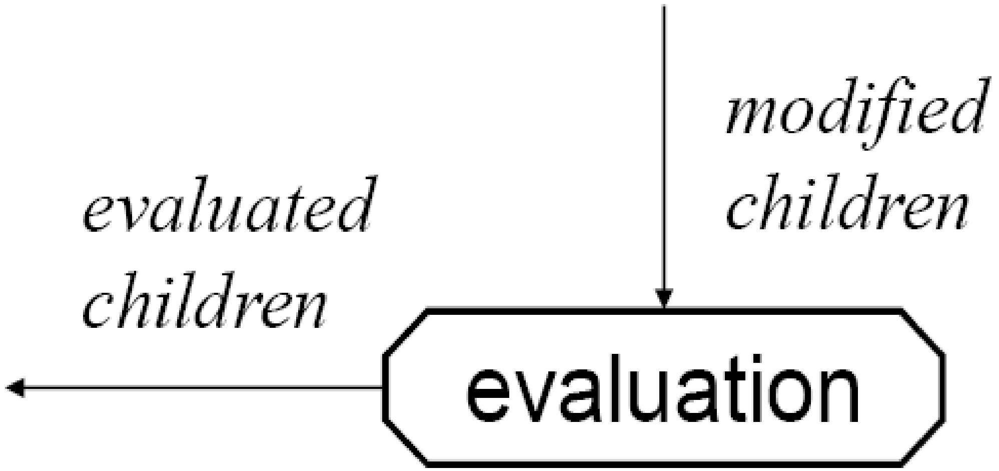
Crossover: Recombination



Crossover is a critical feature of genetic algorithms:

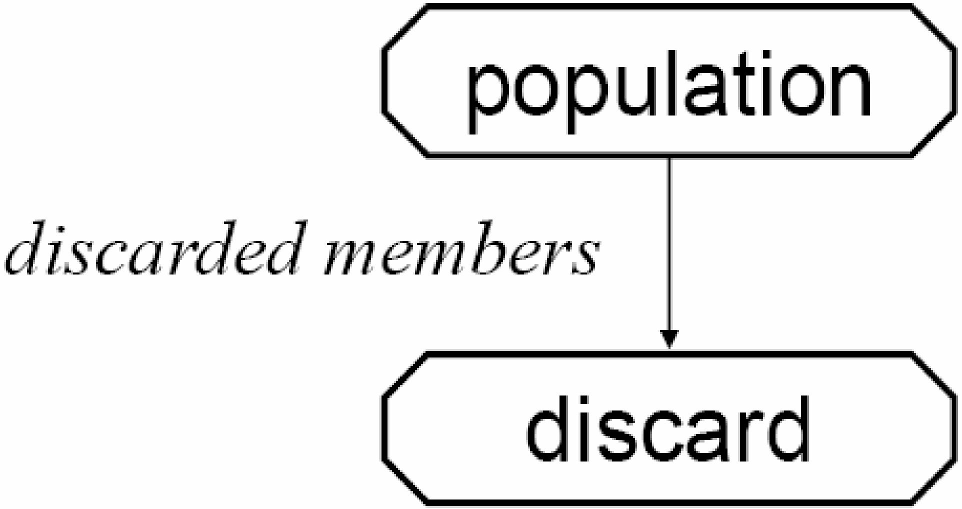
- It greatly accelerates search early in evolution of a population
- It leads to effective combination of schemata (subsolutions on different chromosomes)

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

Deletion



population

discarded members

discard

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

Steps in the GA development

1. Specify the problem, define constraints and optimum criteria;
2. Represent the problem domain as a chromosome;
3. Define a fitness function to evaluate the chromosome performance;
4. Construct the genetic operators;
5. Run the GA and tune its parameters.

The End!