Informed Search Methods

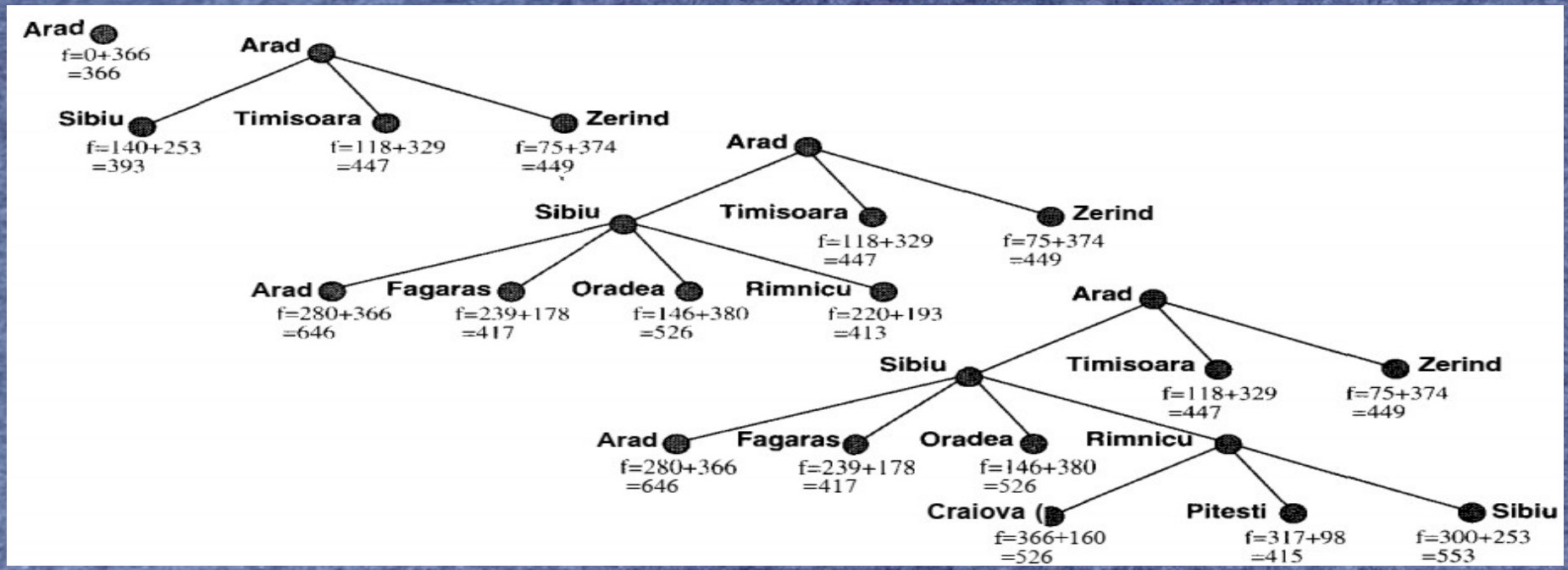
Sunil Surve
Fr. Conceicao Rodrigues College of Engineering

A* Algorithm

- Greedy search minimizes the estimated cost to the goal, *h(n),* and thereby cuts the search cost considerably
- Uniform-cost search, on the other hand, minimizes the cost of the path so far, *g(n);* it is optimal and complete, but can be very inefficient
- Combining the two evaluation functions simply by summing them:
  f(n) = g(n) + h(n)
  where *g(n)* is the path cost from the start node to node *n*, *h(n) is the estimated cost of the cheapest path from n to the goal* *and* f(n) is estimated cost of the cheapest solution through *n*
- **Admissible heuristic**
- ***If h is admissible, f (n) never overestimates the actual cost of* the best solution through n**

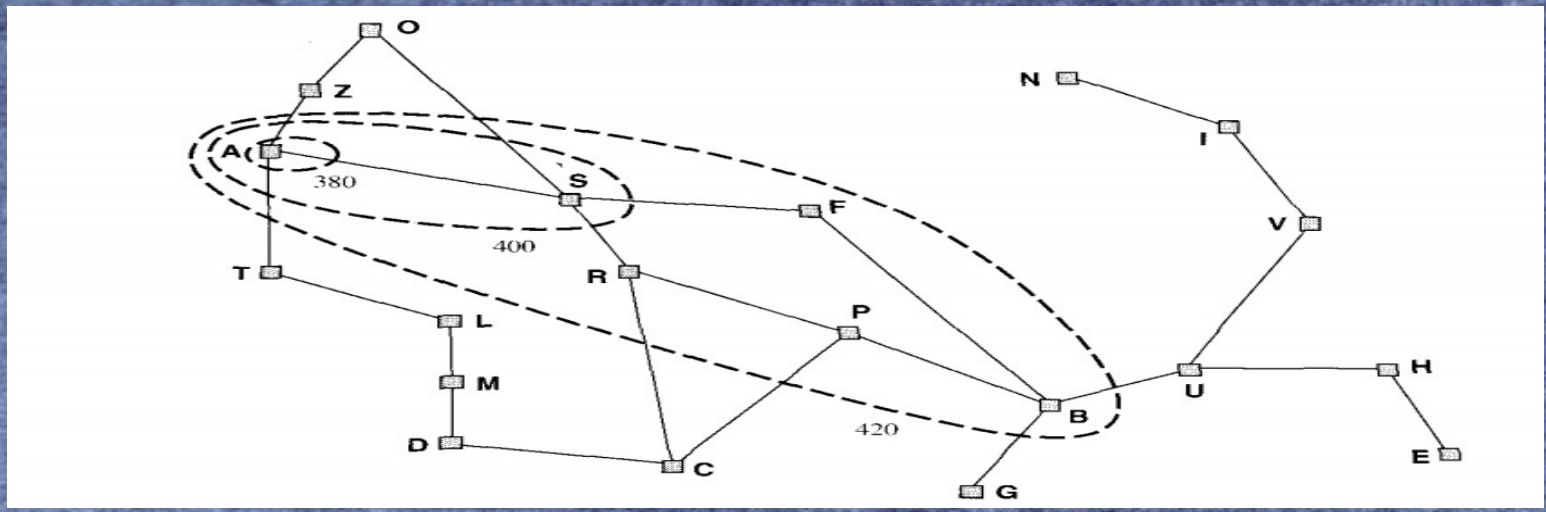
A* Algorithm

- Along any path from the root, the f-cost never decreases.
- It holds true for almost all admissible heuristics.
- A heuristic for which it holds is said to exhibit **monotonicity**



A* Algorithm

- If f-cost of child is less than parent's f-cost, then heuristic used is nonmonotonic.
- Pathmax equation f(n') = max(f(n), g(n') + h(n'))
- If/ never decreases along any path out from the root, we can conceptually draw contours in the state space



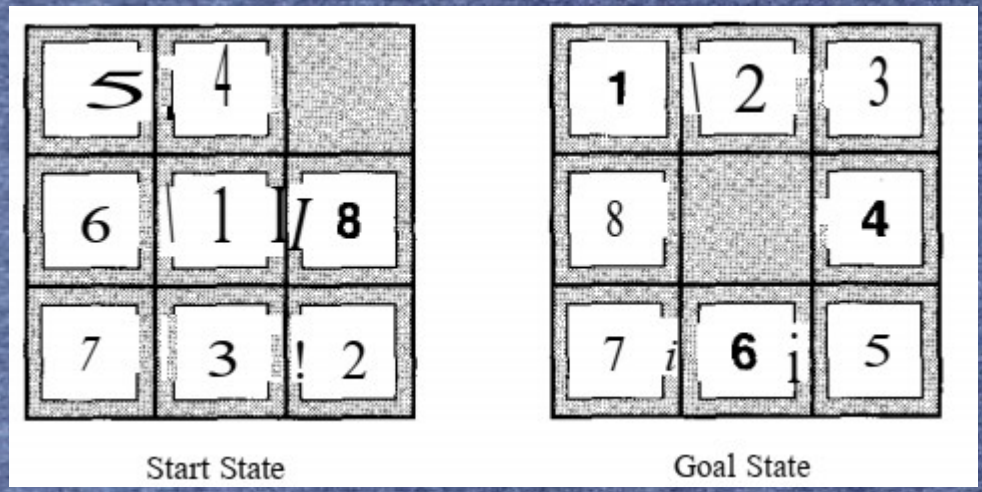
A* Algorithm

- If we define/* to be the cost of the optimal solution path, then we can say the following:
  - A* expands all nodes with f(n) < f*.
  - A* may then expand some of the nodes right on the "goal contour," for which f(n) = f*, before selecting a goal node
- First solution found must be the optimal one, because nodes in all subsequent contours will have higher/-cost, and thus higher g-cost
- A* is **optimally efficient** for any given heuristic function

function A*-SEARCH(*problem)* **returns** a solution or failure
return BEST-FIRST-SEARCH(problem,g + *h)*

Heuristic Functions

- 8-puzzle - the object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the initial configuration matches the goal configuration
- The branching factor is about 3, and 320 = 3.5 x 109 states and By keeping track of repeated states, there are only 9! = 362,880 states



Heuristic Functions

- h_1 = the number of tiles that are in the wrong position. For Figure 4.7, none of the 8 tiles is in the goal position, so the start state would have h_1 = 8. h_1 is an admissible heuristic
- h_2 = the sum of the distances of the tiles from their goal positions. This is sometimes called the **city block distance** or **Manhattan distance.** *h2* is also admissible h_2 = 2 + 3 + 2 + 1+2 + 2+ 1+2 =15
- Which heuristic function is better? h_1 or h_2?
- h_2 will expand lesser nodes than h_1.

Inventing heuristic functions

- **Relaxed problem**
- **If a collection of admissible heuristics $h_1,\ldots,h_m$ is available for a problem, and none of them dominates any of the others, which should we choose?**
  $h(n) = max(h_1(n),\ldots,h_m(n))$.
- ***Another way to invent a good heuristic is to use statistical information***

Iterative deepening A* search (IDA*)

- Use an f-cost limit
- Each iteration expands all nodes inside the contour for the current f-cost
- Once the search inside a given contour has been completed, a new iteration is started using a newf-cost for the next contour
- IDA* is complete and optimal with the same caveats as A* search IDA* is complete and optimal with the same caveats as A* search
- Requires space proportional to the longest path that it explores.
- If *b* is the smallest operator cost and f* the optimal solution cost, then in the worst case, IDA* will require *bf*/8* nodes of storage.
- In most cases, *bd* is a good estimate of the storage requirements

Iterative deepening A* search (IDA*)

- IDA* has difficulty in more complex domains
- In the travelling salesperson problem, for example, the heuristic value is different for every state. This means that each contour only includes one more state than the previous contour.
- If A* expands *N* nodes, IDA* will have to go through *N* iterations and will therefore expand $1 + 2 + \bullet \bullet - + N = O(N^2)$ nodes
- One way around this is to increase the f-cost limit by a fixed amount ε on each iteration, so that the total number of iterations is proportional to $1/\varepsilon$.
- This can reduce the search cost, at the expense of returning solutions that can be worse than optimal by at most e. Such an algorithm is called **ε-admissible**

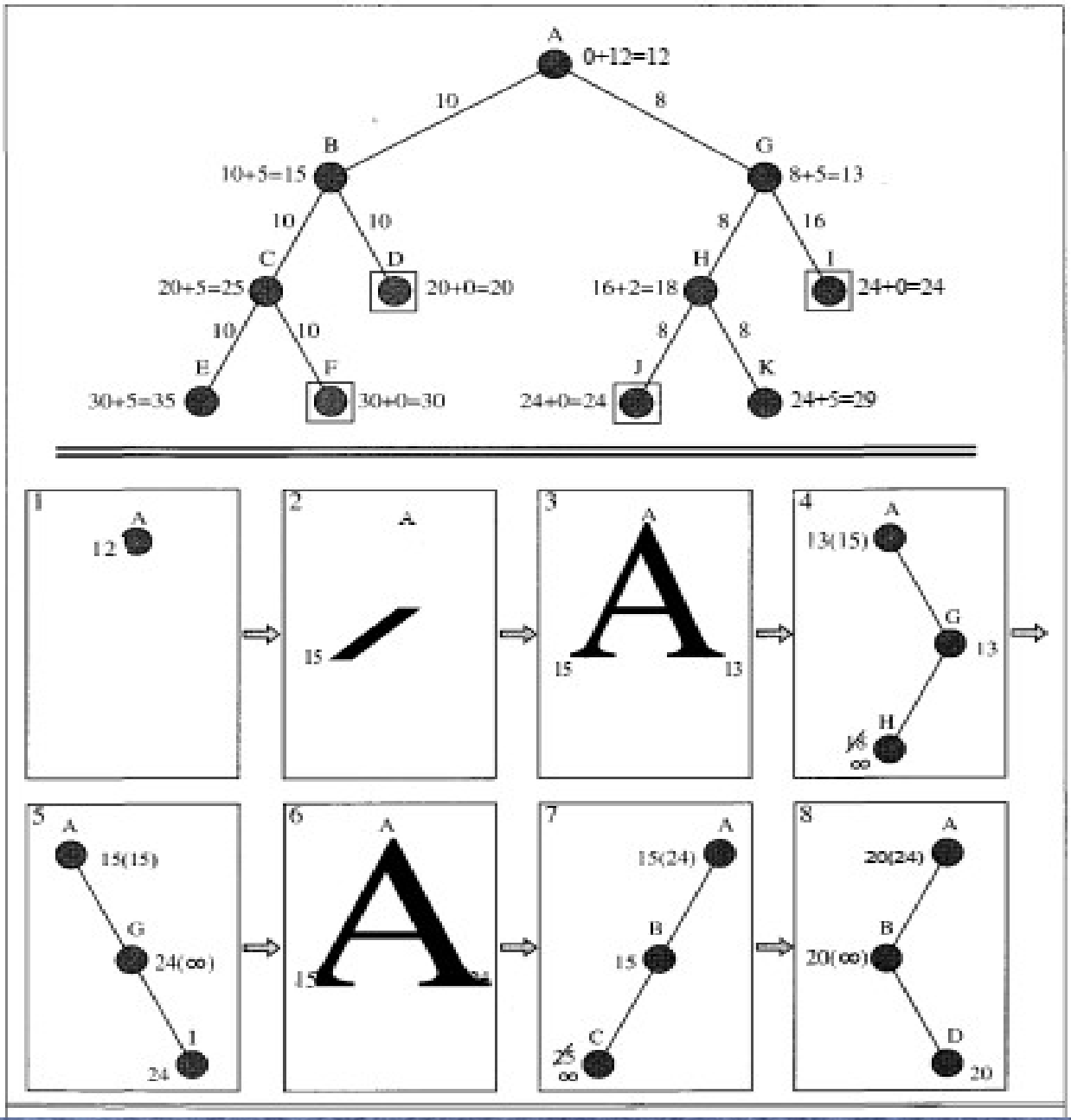
Iterative deepening A* search (IDA*)

```
function IDA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: f-limit, the current f-COST limit
          mot, a node

  root ← MAKE-NODE(INITIAL-STATE[problem])
  f-limit ← f-COST(root)
  loop do
      solution, f-limit ← DFS-CONTOUR(root, f-limit)
      if solution is non-null then return solution
      if f-limit = ∞ then return failure; end
```

```
function DFS-CONTOUR(node, f-limit) returns a solution sequence and a new f-COST limit
  inputs: node, a node
          f-limit, the current f-COST limit
  static: next-f, the f-COST limit for the next contour, initially ∞

  if f-COST[node] > f-limit then return null, f-COST[node]
  if GOAL-TEST[problem](STATE[node]) then return node, f-limit
  for each node s in SUCCESSORS(node) do
      solution, new-f ← DFS-CONTOUR(s, f-limit)
      if solution is non-null then return solution, f-limit
      next-f ← MIN(next-f, new-f); end
  return null, next-f
```

Simplified Memory-Bounded A*

- SMA* has the following properties:
  - It will utilize whatever memory is made available to it.
  - It avoids repeated states as far as its memory allows.
  - It is complete if the available memory is sufficient to store the shallowest solution path.
  - It is optimal if enough memory is available to store the shallowest optimal solution path.
  - Otherwise, it returns the best solution that can be reached with the available memory.
  - When enough memory is available for the entire search tree, the search is optimally efficien

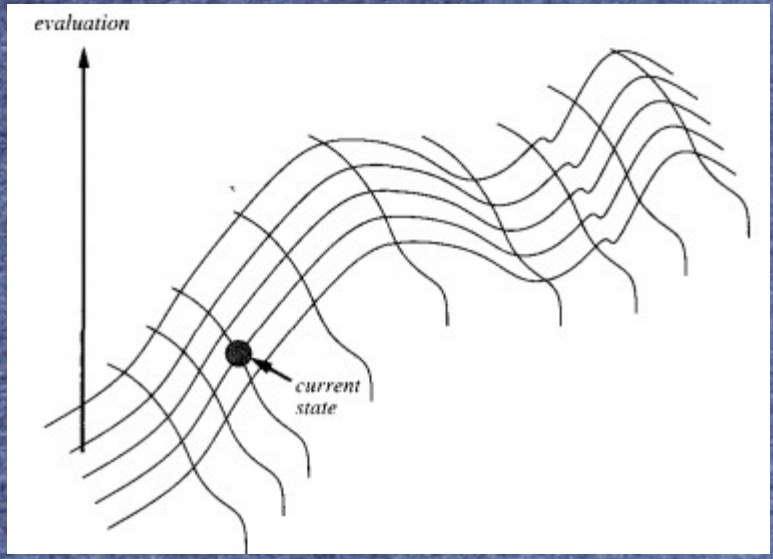

A
0+12=12
10
8
B
10+5=15
G
8+5=13
10
10
8
16
C
20+5=25
D
20+0=20
H
16+2=18
I
24+0=24
10
10
8
8
E
30+5=35
F
30+0=30
J
24+0=24
K
24+5=29
1
A
12
2
A
15
3
A
15
13
4
A
13(15)
G
13
H
∞
5
A
15(15)
G
24(∞)
I
24
6
A
15
7
A
15(24)
B
15
C
∞
8
A
20(24)
B
20(∞)
D
20

Simplified Memory-Bounded A*

```
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: Queue, a queue of nodes ordered by f-cost

  Queue ← MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])})
  loop do
    if Queue is empty then return failure
    n ← deepest least-f-cost node in Queue
    if GOAL-TEST(n) then return success
    s ← NEXT-SUCCESSOR(n)
    if s is not a goal and is at maximum depth then
      f(s) ← ∞
    else
      f(s) ← MAX(f(n), g(s)+h(s))
    if all of n's successors have been generated then
      update n's f-cost and those of its ancestors if necessary
    if SUCCESSORS(n) all in memory then remove n from Queue
    if memory is full then
      delete shallowest, highest-f-cost node in Queue
      remove it from its parent's successor list
      insert its parent on Queue if necessary
    insert s on Queue
  end
```

Hill-climbing search

- **A loop that continually moves in the direction of increasing value.**
- **The algorithm does not maintain a search tree**
- **Local maxima: a local maximum is a peak that is lower than the highest peak in the state space. Once on a local maximum, the algorithm will halt even though the solution may be far from satisfactory**
- **Plateaux: a plateau is an area of the state space where the evaluation function is essentially flat.**
- **Ridges:** a ridge may have steeply sloping sides, so that the search reaches the top of the ridge with ease, but the top may slope only very gently toward a peak



Hill-climbing search

function HILL-CLIMBING(*problem)* **returns** a solution state

inputs: *problem,* a problem

static: *current,* a node

next, a node

current <— MAKE-NODE(INITIAL-STATE[proW(?m])

loop do

next'— a highest-valued successor of *current*

if VALUE[next] < VALUE[current] **then return** *current*

*current *—next*

end

Simulated annealing

- Instead of starting again randomly when stuck on a local maximum, allow the search to take some downhill steps to escape the local maximum picks a *random* move
- If the move actually improves the situation, it is always executed.
- Otherwise, the algorithm assigns some probability less than 1 to move. The probability decreases exponentially with the "badness" of the move
- At higher values of parameter *T,* "bad" moves are more likely to be allowed.
- As *T* tends to zero, they become more and more unlikely, until the algorithm behaves more or less like hill-climbing.
- The *schedule* input determines the value of *T* as a function of how many cycles already have been completed

Simulated annealing

- "Annealing" refers to an analogy with thermodynamics, specifically with the way that metals cool and anneal.
- It involves heating a material above its recrystallization temperature, maintaining a suitable temperature for an appropriate amount of time and then cooling
- Simulated annealing uses the objective function of an optimization problem instead of the energy of a material
- The algorithm is basically hill-climbing except instead of picking the best move, it picks a random move.
- If the selected move improves the solution, then it is always accepted.
- Otherwise, the algorithm makes the move anyway with some probability less than 1.

Simulated annealing

- The probability decreases exponentially with the “badness” of the move, which is the amount deltaE by which the solution is worsened (i.e., energy is increased.)
- Prob(accepting uphill move) ~ 1 - exp(deltaE / kT))
- A parameter T is also used to determine this probability. It is analogous to temperature in an annealing system.
- At higher values of T, uphill moves are more likely to occur. As T tends to zero, they become more and more unlikely, until the algorithm behaves more or less like hill-climbing.
- In a typical SA optimization, T starts high and is gradually decreased according to an “annealing schedule”.
- The parameter k is some constant that relates temperature to energy

Simulated annealing

function SIMULATED-ANNEALiNCXproWewi, *schedule)* **returns** a solution
state
inputs: *problem,* a problem
schedule, a mapping from time to "temperature"
static: *current,* a node
next, a node
T, a "temperature" controlling the probability of downward steps
Current <-- MAKE-NODE(lNITIAL-STATE[/>roWeTO])
for *t* <— 1 to oo **do**
T'<— *schedule[t]*
if *T=0* **then return** *current*
next <— a randomly selected successor of *current*
A.E <-- VALUE(next) - *VALUE[current]*
if **A£ > 0 then** *current^ next*
else *current <-- next only* with probability $e^{\Delta E/T}$

Genetic Algorithms - Introduction

- Genetic Algorithm (GA) is a search-based optimization technique based on the principles of Genetics and Natural Selection
- The set of all possible solutions or valus which the inputs can take make up the search space.
- In this search space, lies a point or a set of points which gives the optimal solution.
- The aim of optimization is to find that point or set of points in the search space.
- Genetic Algorithms (GAs) are search based algorithms based on the concepts of natural selection and genetics.
- GAs are a subset of a much larger branch of computation known as Evolutionary Computation.

Advantages of GAs

- Does not require any derivative information (which may not be available for many real-world problems).
- Is faster and more efficient as compared to the traditional methods.
- Has very good parallel capabilities.
- Optimizes both continuous and discrete functions and also multi-objective problems.
- Provides a list of “good” solutions and not just a single solution.
- Always gets an answer to the problem, which gets better over the time.
- Useful when the search space is very large and there are a large number of parameters involved.

Limitations of GAs

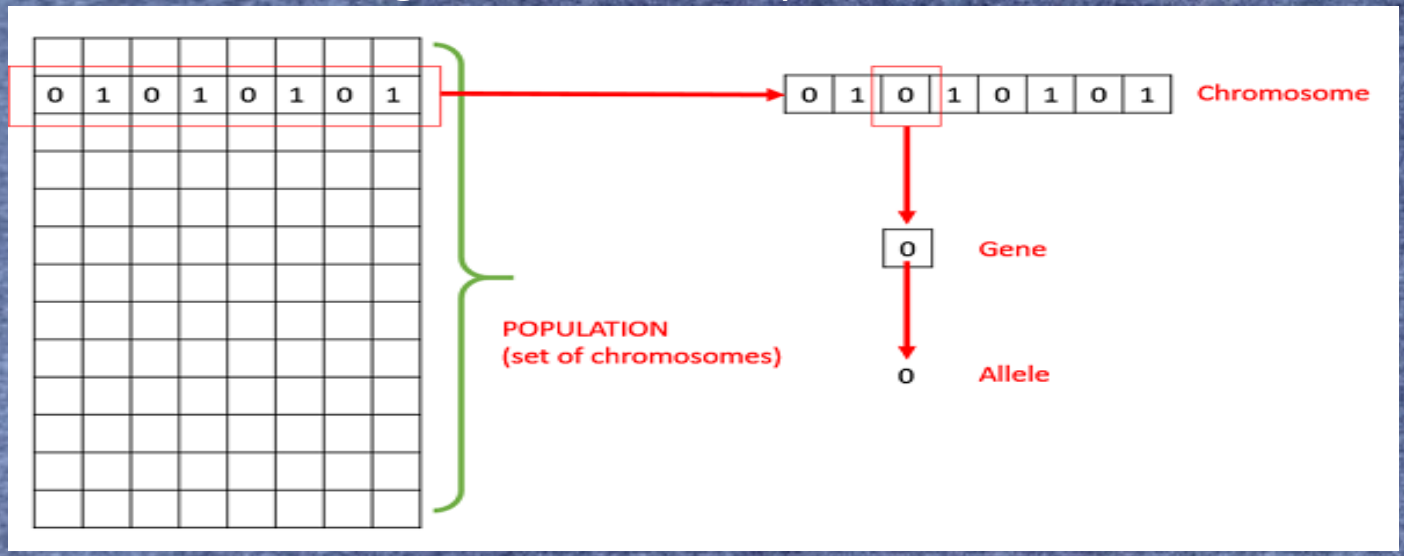
- GAs are not suited for all problems, especially problems which are simple and for which derivative information is available.
- Fitness value is calculated repeatedly which might be computationally expensive for some problems.
- Being stochastic, there are no guarantees on the optimality or the quality of the solution.
- If not implemented properly, the GA may not converge to the optimal solution.

GA – Motivation

- Genetic Algorithms have the ability to deliver a “good-enough” solution “fast-enough”. This makes genetic algorithms attractive for use in solving optimization problems
- Solving Difficult Problems
  - NP-Hard.
  - GAs prove to be an efficient tool to provide usable near-optimal solutions in a short amount of time.
- Getting a Good Solution Fast
- Travelling Salesperson Problem (TSP), real-world applications like path finding and VLSI Design.
  - GPS Navigation system takes a few minutes (or even a few hours) to compute the “optimal” path from the source to destination.
  - Delay in such real world applications is not acceptable and therefore a “good-enough” solution, which is delivered “fast” is what is required.

Basic Terminology

- Population – It is a subset of all the possible (encoded) solutions to the given problem.
- Chromosomes – A chromosome is one such solution to the given problem.
- Gene – A gene is one element position of a chromosome.
- Allele – It is the value a gene takes for a particular chromosome.



Basic Terminology

- Genotype – Genotype is the population in the computation space. In the computation space, the solutions are represented in a way which can be easily understood and manipulated using a computing system.
- Phenotype – Phenotype is the population in the actual real world solution space in which solutions are represented in a way they are represented in real world situations.
- Decoding and Encoding – For simple problems, the phenotype and genotype spaces are the same. However, in most of the cases, the phenotype and genotype spaces are different. Decoding is a process of transforming a solution from the genotype to the phenotype space, while encoding is a process of transforming from the phenotype to genotype space.

Basic Terminology

- For example, consider the 0/1 Knapsack Problem. The Phenotype space consists of solutions which just contain the item numbers of the items to be picked.
- However, in the genotype space it can be represented as a binary string of length n (where n is the number of items). A 0 at position x represents that xth item is picked while a 1 represents the reverse. This is a case where genotype and phenotype spaces are different.

Basic Terminology

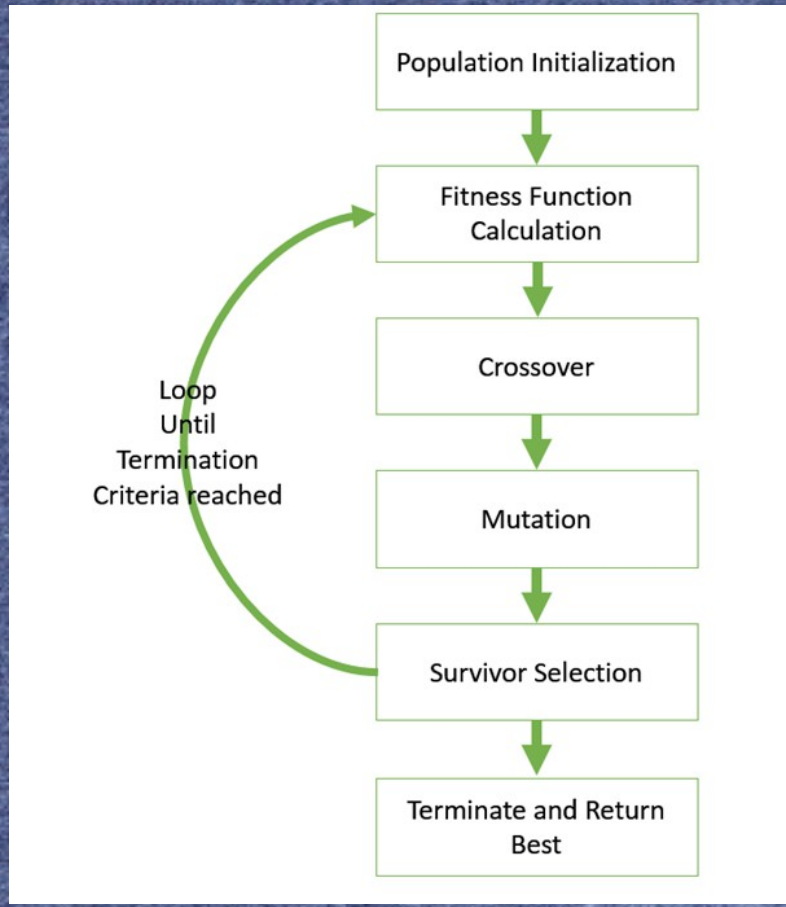
- Fitness Function – A fitness function simply defined is a function which takes the solution as input and produces the suitability of the solution as the output. In some cases, the fitness function and the objective function may be the same, while in others it might be different based on the problem.
- Genetic Operators – These alter the genetic composition of the offspring. These include crossover, mutation, selection, etc.

Basic Terminology

- Crossover. Swaping parts of the solution with another in chromosomes or solution representations. The main role is to provide mixing of the solutions and convergence in a subspace.
- Mutation. The change of parts of one solution randomly, which increases the diversity of the population and provides a mechanism for escaping from a local optimum.
- Selection of the fittest, or elitism. The use of the solutions with high fitness to pass on to next generations, which is often carried out in terms of some form of selection of the best solutions.

Basic Structure

- Start with an initial population (which may be generated at random or seeded by other heuristics), select parents from this population for mating.
- Apply crossover and mutation operators on the parents to generate new off-springs.
- And finally these off-springs replace the existing individuals in the population and the process repeats.



Genotype Representation

- One of the most important decisions to make while implementing a genetic algorithm is deciding the representation that we will use to represent solutions
- A proper definition of the mappings between the phenotype and genotype spaces is essential for the success of a GA
- Binary Representation:
- For some problems when the solution space consists of Boolean decision variables – yes or no, the binary representation is natural
- For other problems, specifically those dealing with numbers, we can represent the numbers with their gray code representation

Genotype Representation

- Real Valued Representation: For problems where genes uses continuous rather than discrete variables, the real valued representation is the most natural

0.5	0.2	0.6	0.8	0.7	0.4	0.3	0.2	0.1	0.9

- Integer Representation: If we want to encode the four distances – North, South, East and West, we can encode them as {0,1,2,3}. In such cases, integer representation is desirable

1	2	3	4	3	2	4	1	2	1

Genotype Representation

- Permutation Representation: In many problems, the solution is represented by an order of elements.
- In such cases permutation representation is the most suited.
- A classic example of this representation is the travelling salesman problem (TSP). In this the salesman has to take a tour of all the cities, visiting each city exactly once and come back to the starting city. The total distance of the tour has to be minimized

Genetic Algorithms - Population

- Population is a subset of solutions in the current generation. It can also be defined as a set of chromosomes.
- There are several things to be kept in mind when dealing with GA population –
  - The diversity of the population should be maintained otherwise it might lead to premature convergence.
  - The population size should not be kept very large as it can cause a GA to slow down, while a smaller population might not be enough for a good mating pool. Therefore, an optimal population size needs to be decided by trial and error.

Population Initialization

- There are two primary methods to initialize a population in a GA.
- Random Initialization – Populate the initial population with completely random solutions.
- Heuristic initialization – Populate the initial population using a known heuristic for the problem
- It has been observed that the entire population should not be initialized using a heuristic, as it can result in the population having similar solutions and very little diversity.
- It has been experimentally observed that the random solutions are the ones to drive the population to optimality.
- It has also been observed that heuristic initialization in some cases, only effects the initial fitness of the population, but in the end, it is the diversity of the solutions which lead to optimality

Population Models

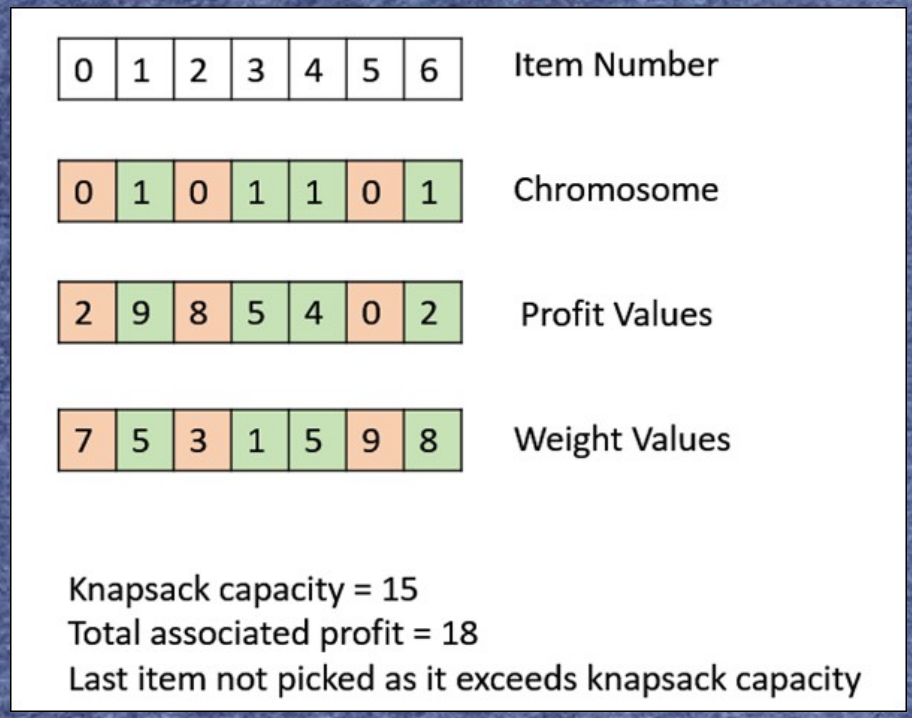
- There are two population models widely in use –
  - Steady State: In steady state GA, we generate one or two off-springs in each iteration and they replace one or two individuals from the population. A steady state GA is also known as Incremental GA.
  - Generational: In a generational model, we generate ‘n’ off-springs, where n is the population size, and the entire population is replaced by the new one at the end of the iteration.

Genetic Algorithms - Fitness Function

- Is a function which takes a candidate solution to the problem as input and produces as output how “fit” our how “good” the solution is with respect to the problem in consideration
- Calculation of fitness value is done repeatedly in a GA
  - sufficiently fast.
  - A slow computation of the fitness value can adversely affect a GA and make it exceptionally slow
- In most cases the fitness function and the objective function are the same as the objectivefunction.
- However, for more complex problems with multiple objectives and constraints, an Algorithm Designer might choose to have a different fitness function

Genetic Algorithms - Fitness Function

- A fitness function should possess the following characteristics –
  - The fitness function should be sufficiently fast to compute.
  - It must quantitatively measure how fit a given solution is or how fit individuals can be produced from the given solution
- In some cases, calculating the fitness function directly might not be possible due to the inherent complexities of the problem at hand. In such cases, we do fitness approximation to suit our needs



Genetic Algorithms - Parent Selection

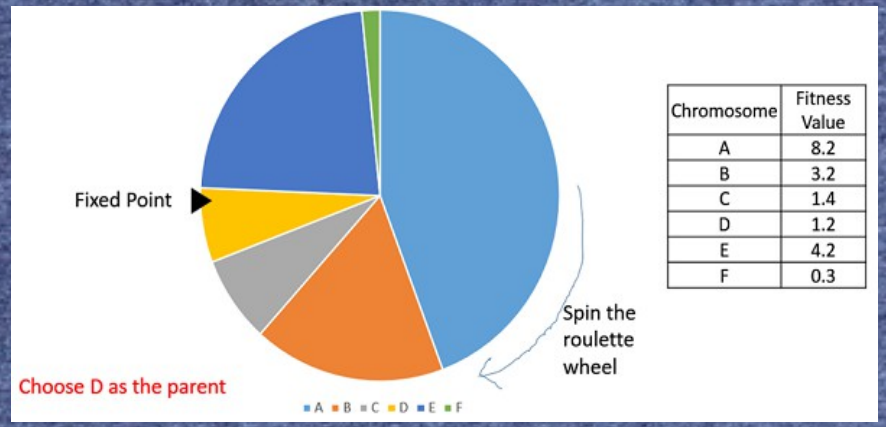
- Parent Selection is the process of selecting parents which mate and recombine to create off-springs for the next generation
- Maintaining good diversity in the population is extremely crucial for the success of a GA.
- This taking up of the entire population by one extremely fit solution is known as **premature convergence** and is an undesirable condition in a GA.

Genetic Algorithms - Parent Selection

- Fitness Proportionate Selection:
  - Every individual can become a parent with a probability which is proportional to its fitness
  - Fitter individuals have a higher chance of mating and propagating their features to the next generation.
  - Therefore, such a selection strategy applies a selection pressure to the more fit individuals in the population, evolving better individuals over time
- Two implementations of fitness proportionate selection are possible –
  - Roulette Wheel Selection
  - Stochastic Universal Sampling (SUS)

Genetic Algorithms - Parent Selection

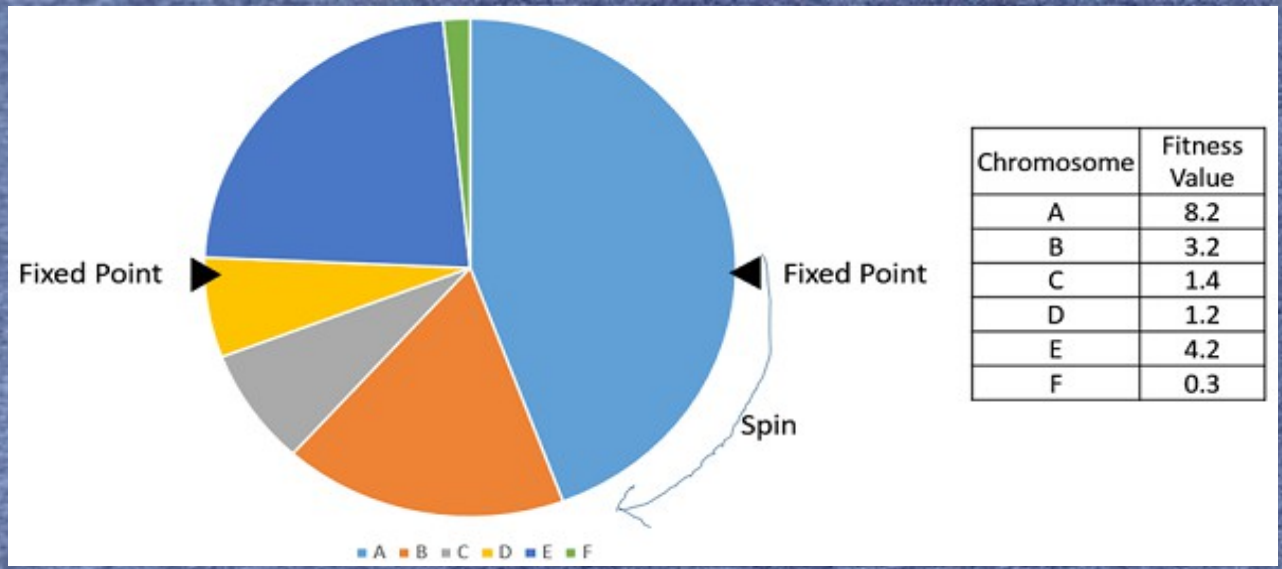
- Roulette Wheel Selection
- Implementation wise, we use the following steps –
- Calculate S = the sum of a finesses.
- Generate a random number between 0 and S.
- Starting from the top of the population, keep adding the finesses to the partial sum P, till P<S.
- The individual for which P exceeds S is the chosen individual.



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Genetic Algorithms - Parent Selection

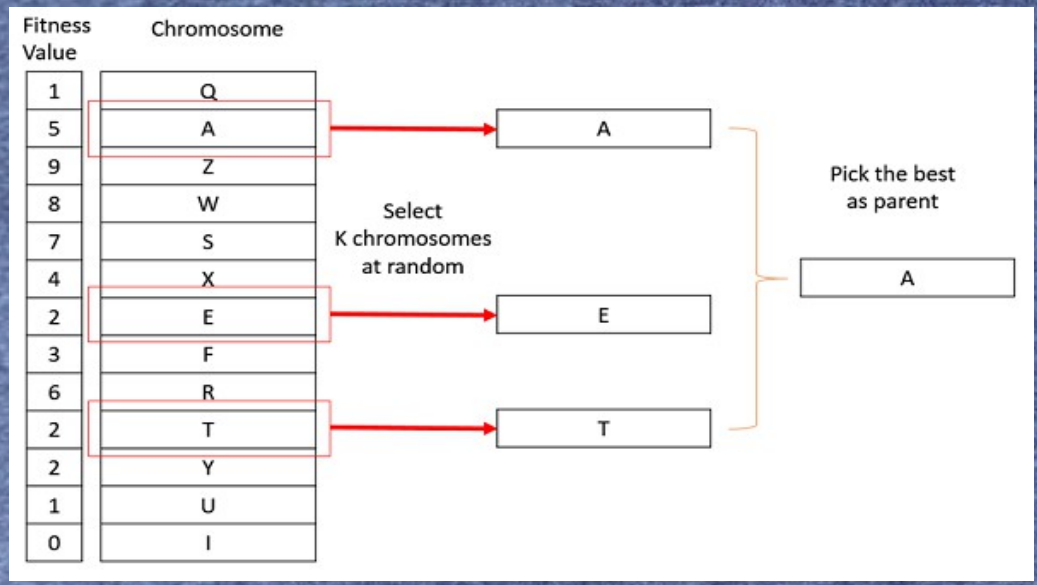
- Stochastic Universal Sampling (SUS):
- multiple fixed points
- parents are chosen in just one spin of the wheel
- fitness proportionate selection methods don't work for cases where the fitness can take a negative value



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

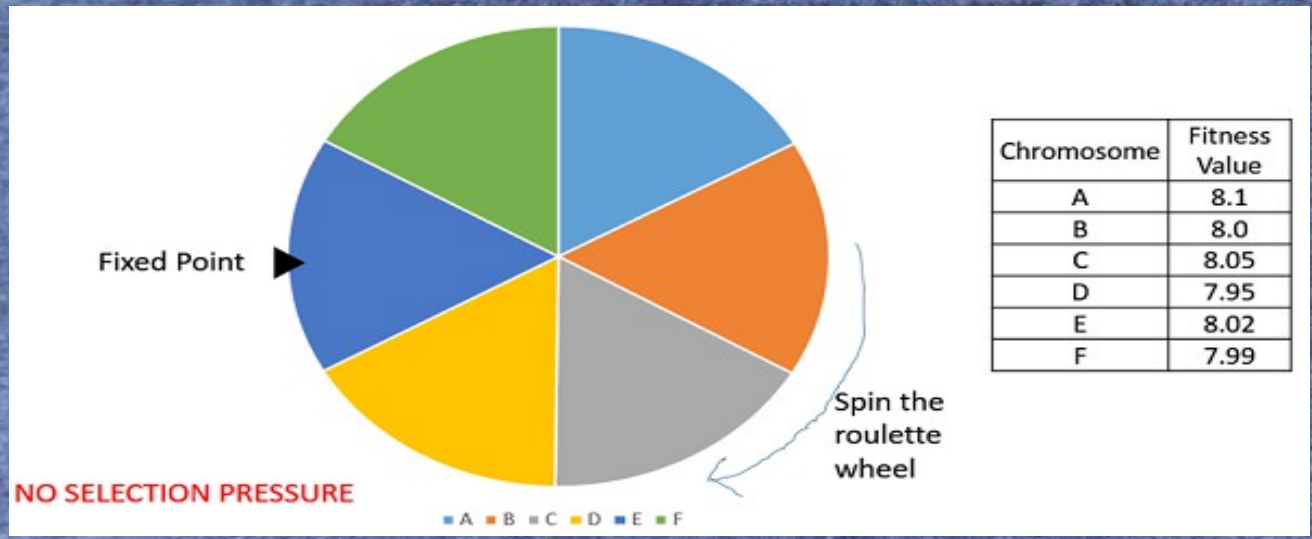
Genetic Algorithms - Parent Selection

- Tournament Selection
- In K-Way tournament selection, we select K individuals from the population at random and select the best out of these to become a parent.
- The same process is repeated for selecting the next parent.



Genetic Algorithms - Parent Selection

- Rank Selection also works with negative fitness values and is mostly used when the individuals in the population have very close fitness values
- Leads to a loss in the selection pressure towards fitter individuals, making the GA to make poor parent selections in such situations.



Chromosome	Fitness Value
A	8.1
B	8.0
C	8.05
D	7.95
E	8.02
F	7.99

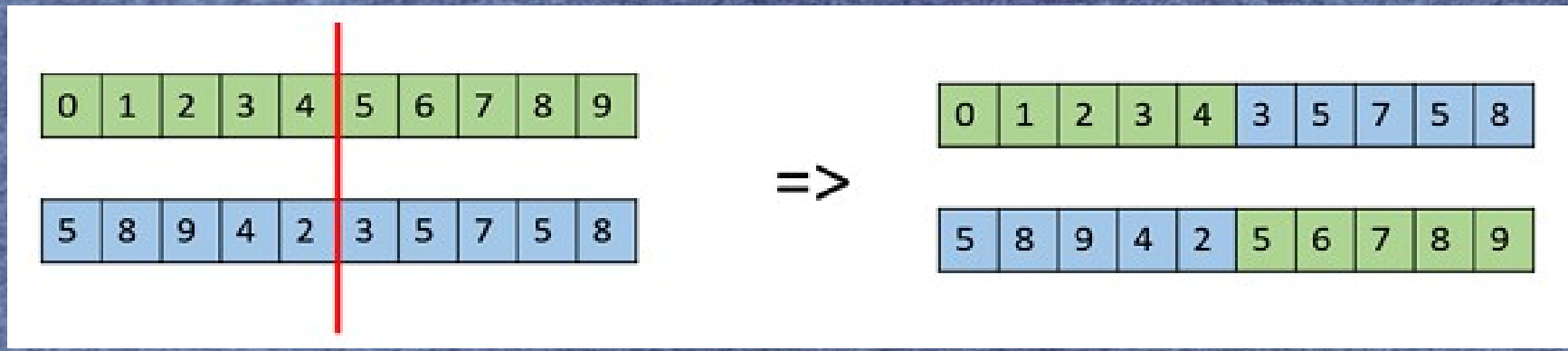
Genetic Algorithms - Parent Selection

Chromosome	Fitness Value	Rank
A	8.1	1
B	8.0	4
C	8.05	2
D	7.95	6
E	8.02	3
F	7.99	5

- Random Selection
- In this strategy we randomly select parents from the existing population.

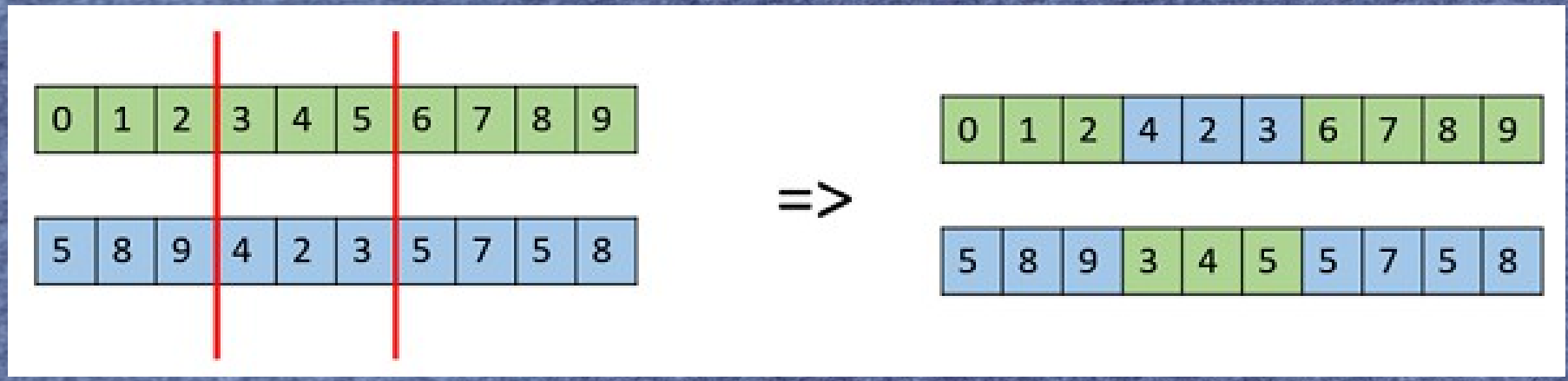
Genetic Algorithms - Crossover

- The crossover operator is analogous to reproduction and biological crossover.
- In this more than one parent is selected and one or more off-springs are produced using the genetic material of the parents.
- Crossover is usually applied in a GA with a high probability – p_c
- **One Point Crossover:** In this one-point crossover, a random crossover point is selected and the tails of its two parents are swapped to get new off-springs.



Genetic Algorithms - Crossover

- Multi Point Crossover: Multi point crossover is a generalization of the one-point crossover wherein alternating segments are swapped to get new off-springs.



Genetic Algorithms - Crossover

- Uniform Crossover:
  - Treat each gene separately.
  - In this, flip a coin for each chromosome to decide whether or not it'll be included in the off-spring.
  - Can also bias the coin to one parent, to have more genetic material in the child from that parent.

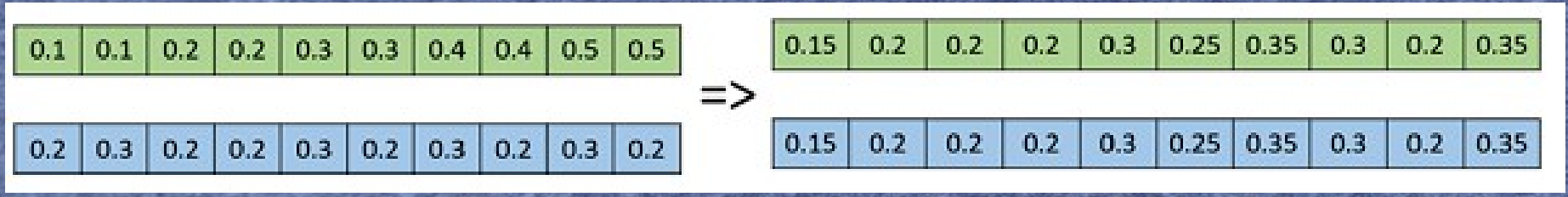
Genetic Algorithms - Crossover

- Whole Arithmetic Recombination

  Child1 = $\alpha.x + (1-\alpha).y$

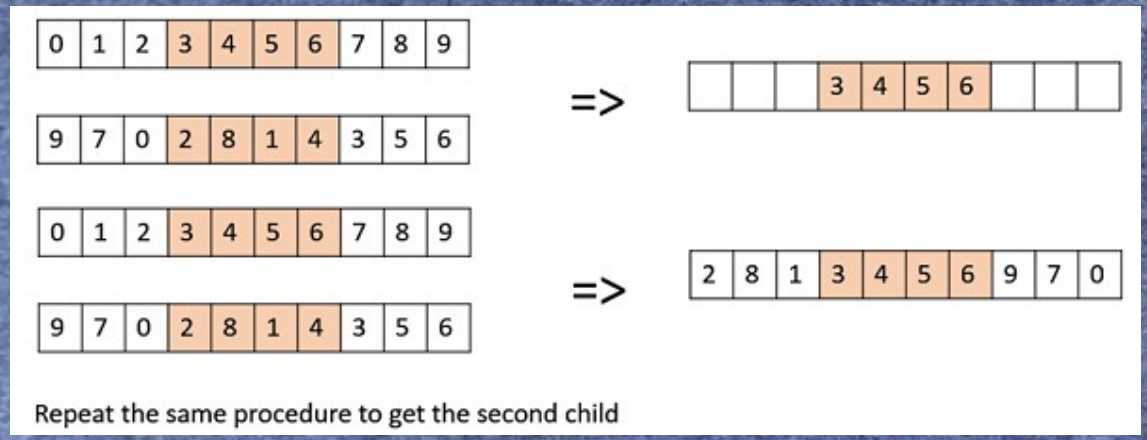
  Child2 = $\alpha.x + (1-\alpha).y$

- Obviously, if $\alpha = 0.5$, then both the children will be identical



Genetic Algorithms - Crossover

- Davis' Order Crossover (OX1): OX1 is used for permutation based crossovers with the intention of transmitting information about relative ordering to the off-springs. It works as follows –
  - Create two random crossover points in the parent and copy the segment between them from the first parent to the first offspring.
  - Now, starting from the second crossover point in the second parent, copy the remaining unused numbers from the second parent to the first child, wrapping around the list.
  - Repeat for the second child with the parent's role reversed.

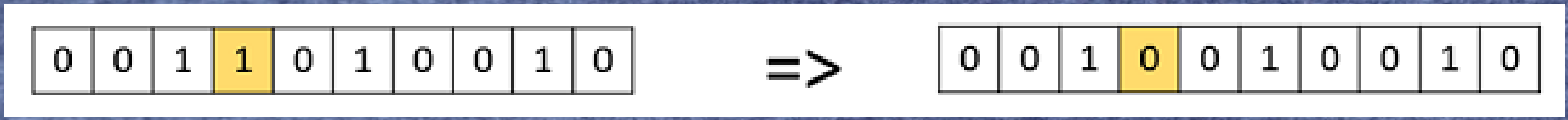


Genetic Algorithms - Mutation

- Mutation may be defined as a small random tweak in the chromosome, to get a new solution.
- It is used to maintain and introduce diversity in the genetic population and is usually applied with a low probability – p_m.
- If the probability is very high, the GA gets reduced to a random search.
- Mutation is the part of the GA which is related to the “exploration” of the search space.
- It has been observed that mutation is essential to the convergence of the GA while crossover is not.

Genetic Algorithms - Mutation

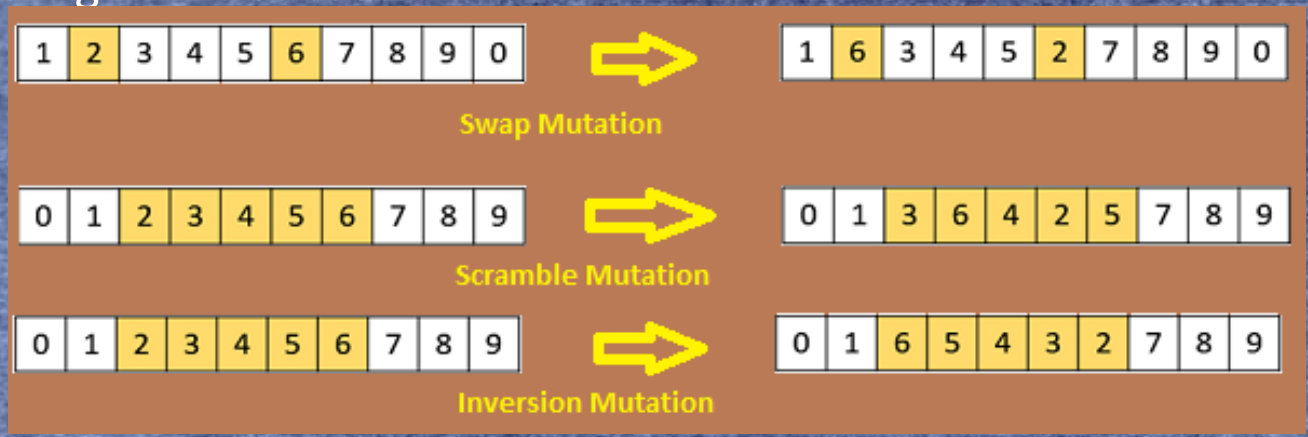
- Bit Flip Mutation: In this bit flip mutation, we select one or more random bits and flip them. This is used for binary encoded Gas.



- Random Resetting: Random Resetting is an extension of the bit flip for the integer representation. In this, a random value from the set of permissible values is assigned to a randomly chosen gene.
- Swap Mutation: In swap mutation, we select two positions on the chromosome at random, and interchange the values. This is common in permutation based encodings.

Genetic Algorithms - Mutation

- Swap Mutation: In swap mutation, we select two positions on the chromosome at random, and interchange the values. This is common in permutation based encodings.
- Scramble Mutation: Scramble mutation is also popular with permutation representations. In this, from the entire chromosome, a subset of genes is chosen and their values are scrambled or shuffled randomly.
- Inversion Mutation: In inversion mutation, we select a subset of genes like in scramble mutation, but instead of shuffling the subset, we merely invert the entire string in the subset.

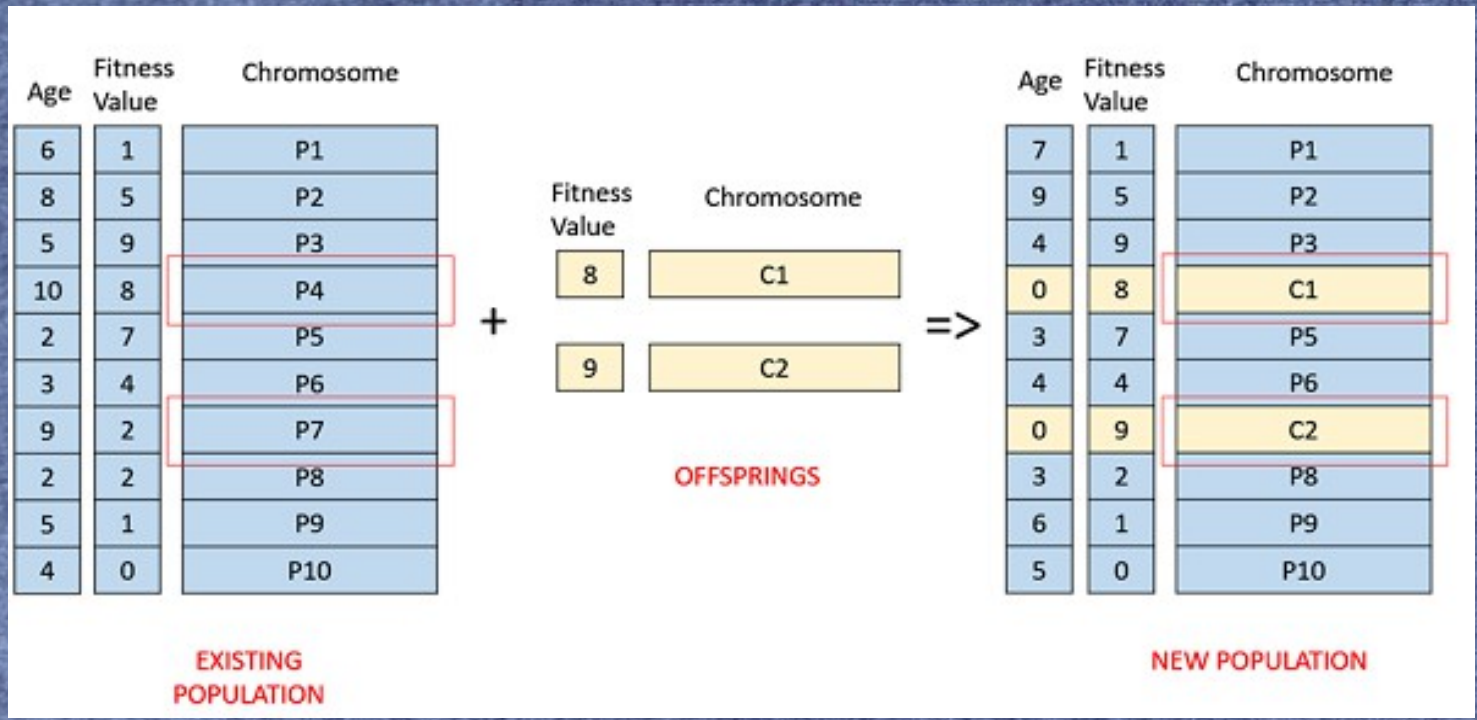


Genetic Algorithms - Survivor Selection

- The Survivor Selection Policy determines which individuals are to be kicked out and which are to be kept in the next generation.
- It is crucial as it should ensure that the fitter individuals are not kicked out of the population, while at the same time diversity should be maintained in the population.
- Some GAs employ Elitism. In simple terms, it means the current fittest member of the population is always propagated to the next generation.
- The easiest policy is to kick random members out of the population, but such an approach frequently has convergence issues

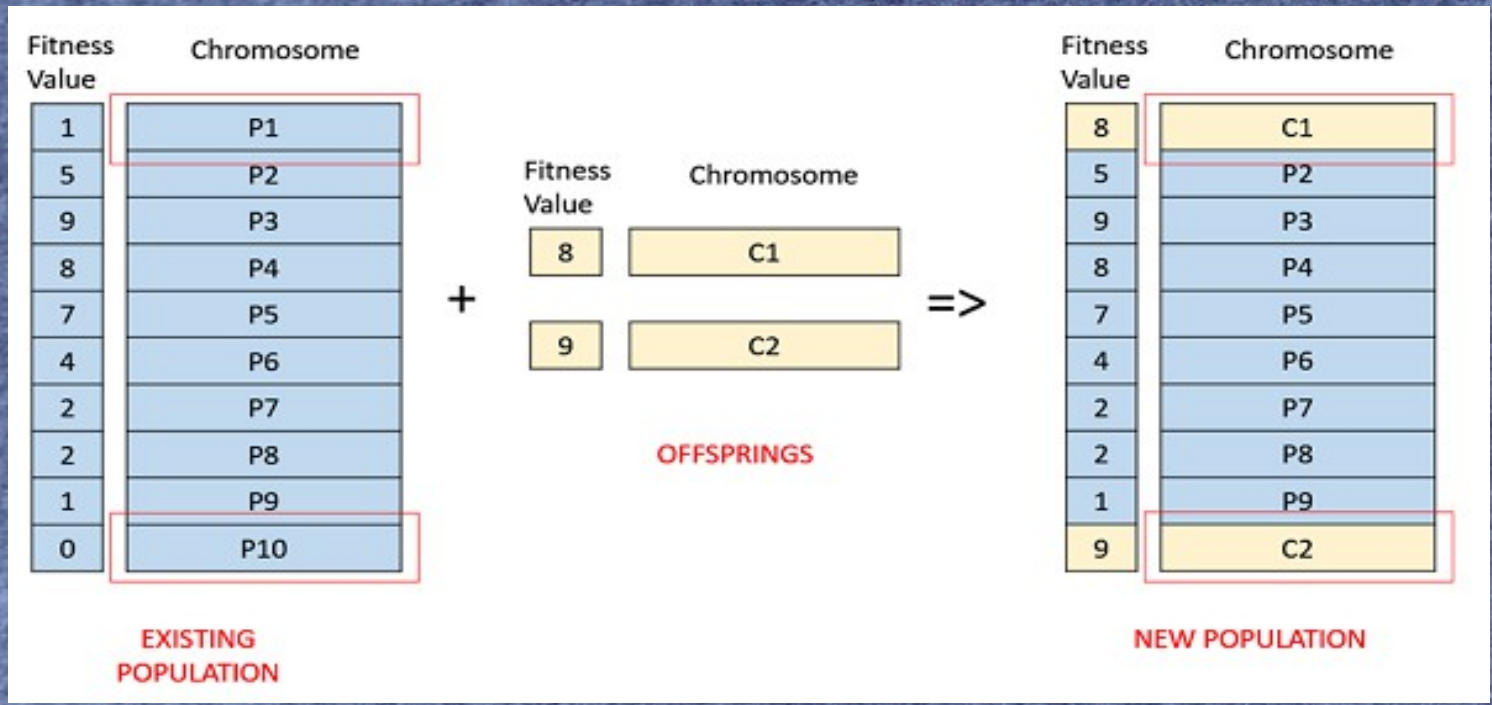
Genetic Algorithms - Survivor Selection

- **Age Based Selection:** It is based on the premise that each individual is allowed in the population for a finite generation where it is allowed to reproduce, after that, it is kicked out of the population no matter how good its fitness is



Genetic Algorithms - Survivor Selection

- **Fitness Based Selection:** the children tend to replace the least fit individuals in the population. The selection of the least fit individuals may be done using a variation of any of the selection policies described before – tournament selection, fitness proportionate selection, etc.

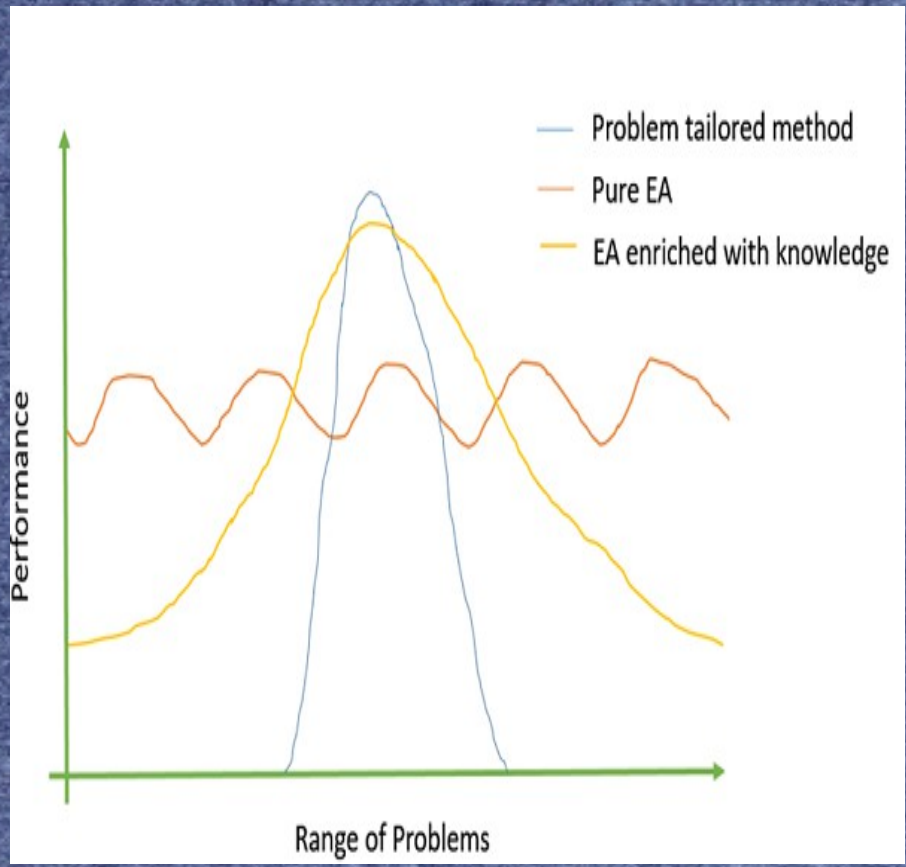


Genetic Algorithms - Termination Condition

- The termination condition of a Genetic Algorithm is important in determining when a GA run will end. It has been observed that initially, the GA progresses very fast with better solutions coming in every few iterations, but this tends to saturate in the later stages where the improvements are very small.
- When there has been no improvement in the population for X iterations.
- When we reach an absolute number of generations.
- When the objective function value has reached a certain pre-defined value.

Effective Implementation

- **Introduce problem-specific domain knowledge:**
  - It has been observed that the more problem-specific domain knowledge we incorporate into the GA; the better objective values we get.
  - Adding problem specific information can be done by either using problem specific crossover or mutation operators, custom representations, etc.



Effective Implementation

- **Reduce Crowding:** Crowding happens when a highly fit chromosome gets to reproduce a lot, and in a few generations, the entire population is filled with similar solutions having similar fitness. This reduces diversity which is a very crucial element to ensure the success of a GA.
- Mutation to introduce diversity
- Switching to rank selection and tournament selection which have more selection pressure than fitness proportionate selection for individuals with similar fitness.
- Fitness Sharing – In this an individual's fitness is reduced if the population already contains similar individuals.

Effective Implementation

- **Randomization:** It has been experimentally observed that the best solutions are driven by randomized chromosomes as they impart diversity to the population. The GA implementer should be careful to keep sufficient amount of randomization and diversity in the population for the best results.
- **Variation of parameters and techniques**: In genetic algorithms, there is no “one size fits all” or a magic formula which works for all problems. Even after the initial GA is ready, it takes a lot of time and effort to play around with the parameters like population size, mutation and crossover probability etc. to find the ones which suit the particular problem.

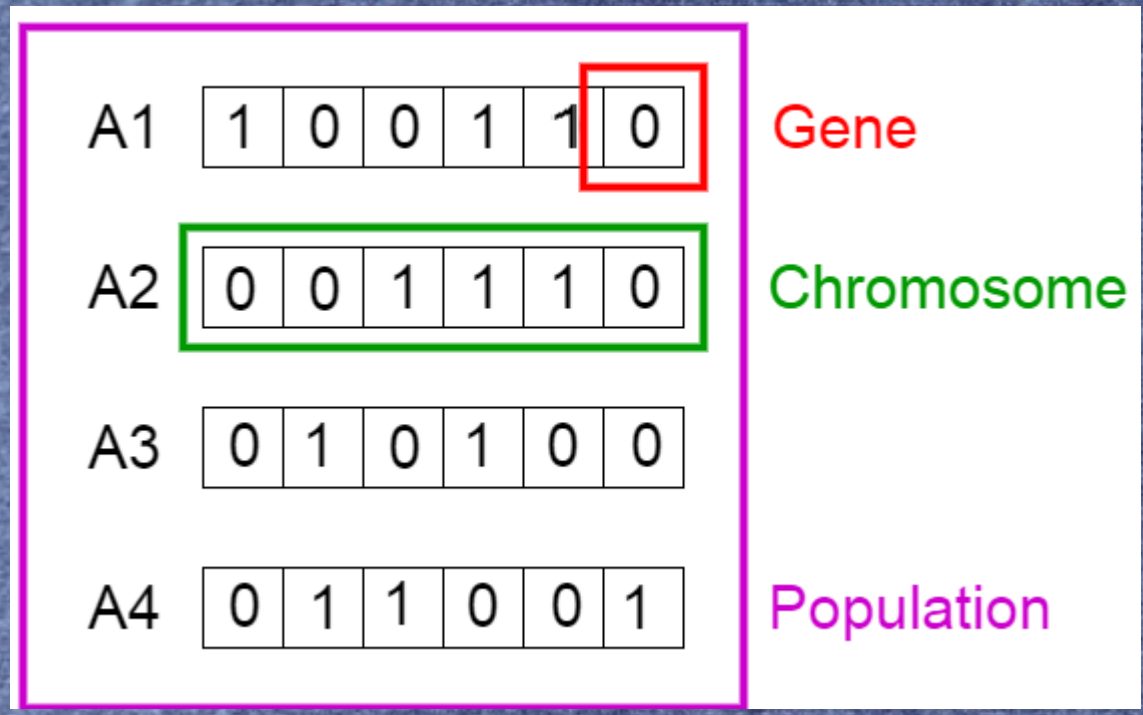
Example 1

Let's say, you are going to spend a month in the wilderness. Only thing you are carrying is the backpack which can hold a maximum weight of 30 kg. Now you have different survival items, each having its own "Survival Points" (which are given for each item in the table). So, your objective is maximise the survival points.

ITEM	WEIGHT	SURVIVAL POINTS
SLEEPING BAG	15	15
ROPE	3	7
POCKET KNIFE	2	10
TORCH	5	5
BOTTLE	9	8
GLUCOSE	20	17

Example 1

Initialisation: chromosomes are binary strings, where for this problem 1 would mean that the following item is taken and 0 meaning that it is dropped



Example 1

Fitness Function: Let us calculate fitness points for our first two chromosomes

For A1 chromosome [100110]

ITEMS	WEIGHT	SURVIVAL POINTS
Sleeping bag	15	15
Torch	5	5
Bottle	9	8
TOTAL	29	28

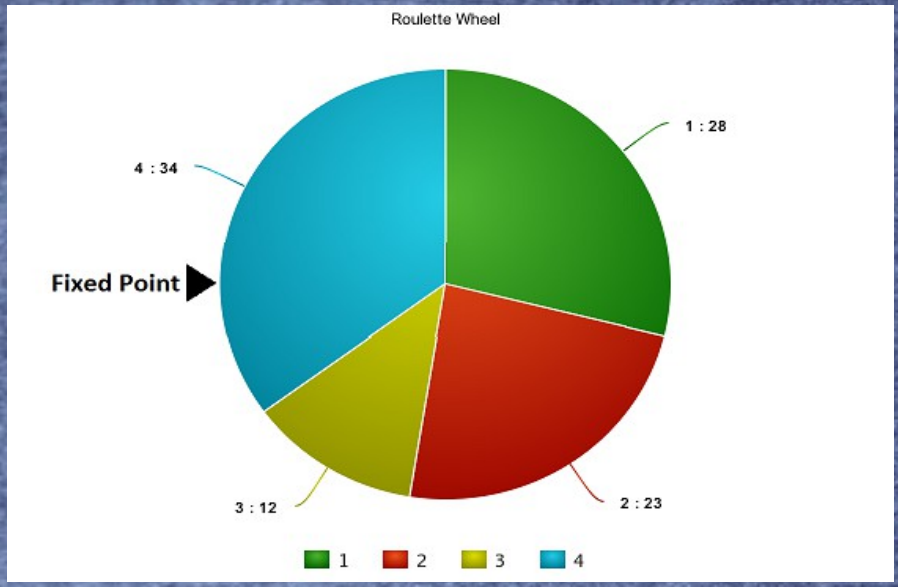
Similarly for A2 chromosome [001110]

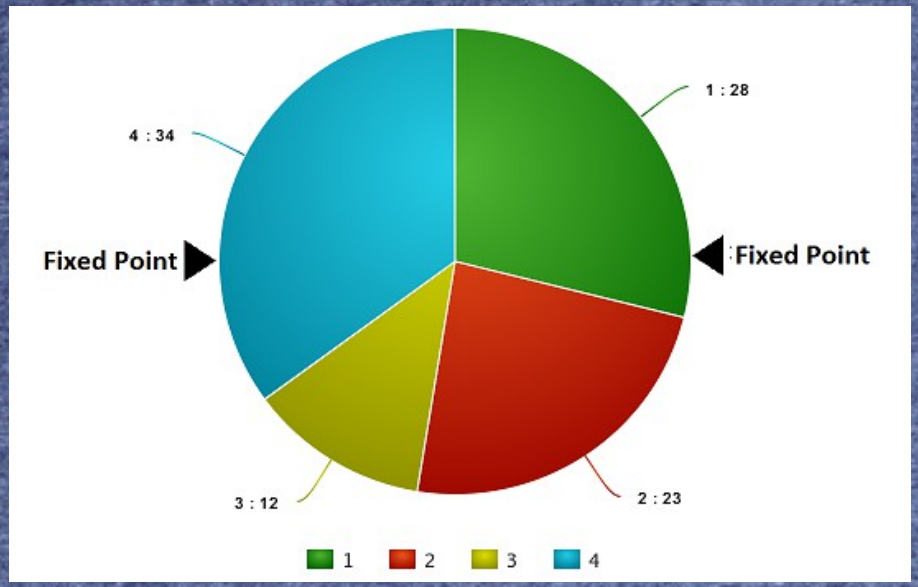
ITEMS	WEIGHT	SURVIVAL POINTS
Pocket Knife	2	10
Torch	5	5
Bottle	9	8
TOTAL	16	23

Example 1

Selection

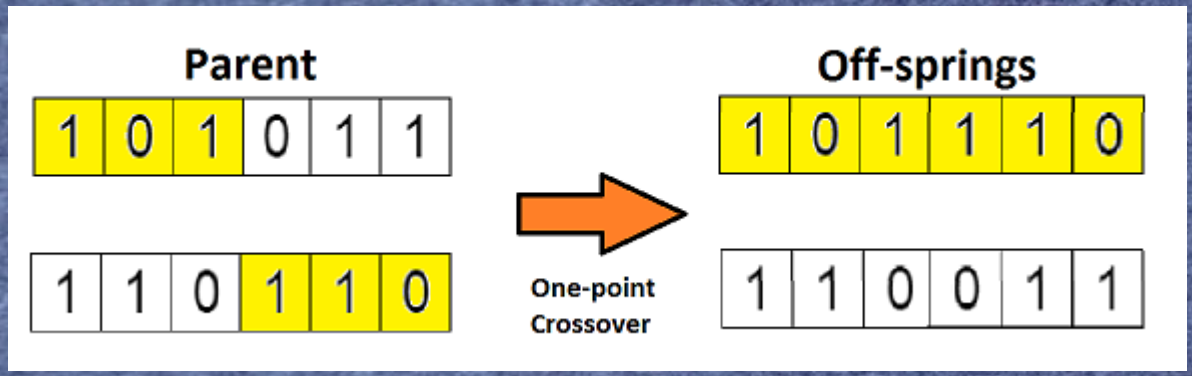
	Survival Points	Percentage
Chromosome 1	28	28.9%
Chromosome 2	23	23.7%
Chromosome 3	12	12.4%
Chromosome 4	34	35.1%

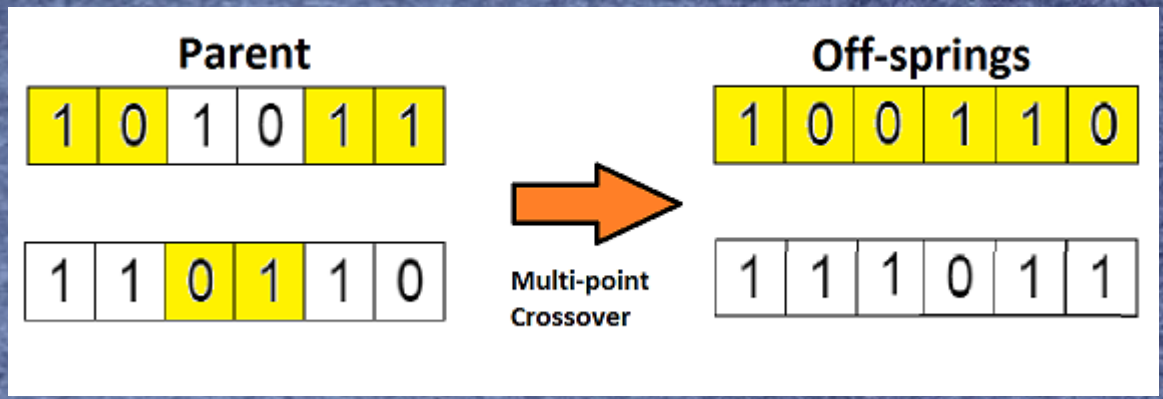




Example 1

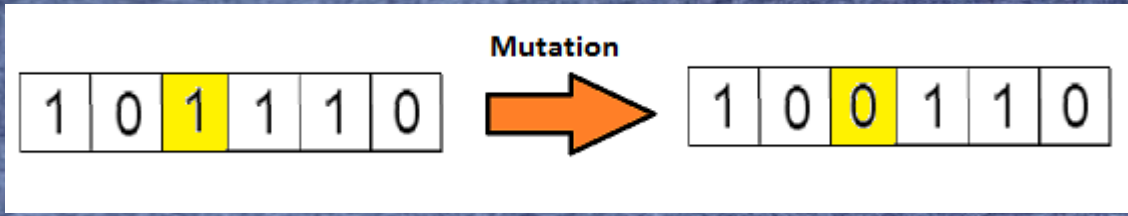
Crossover

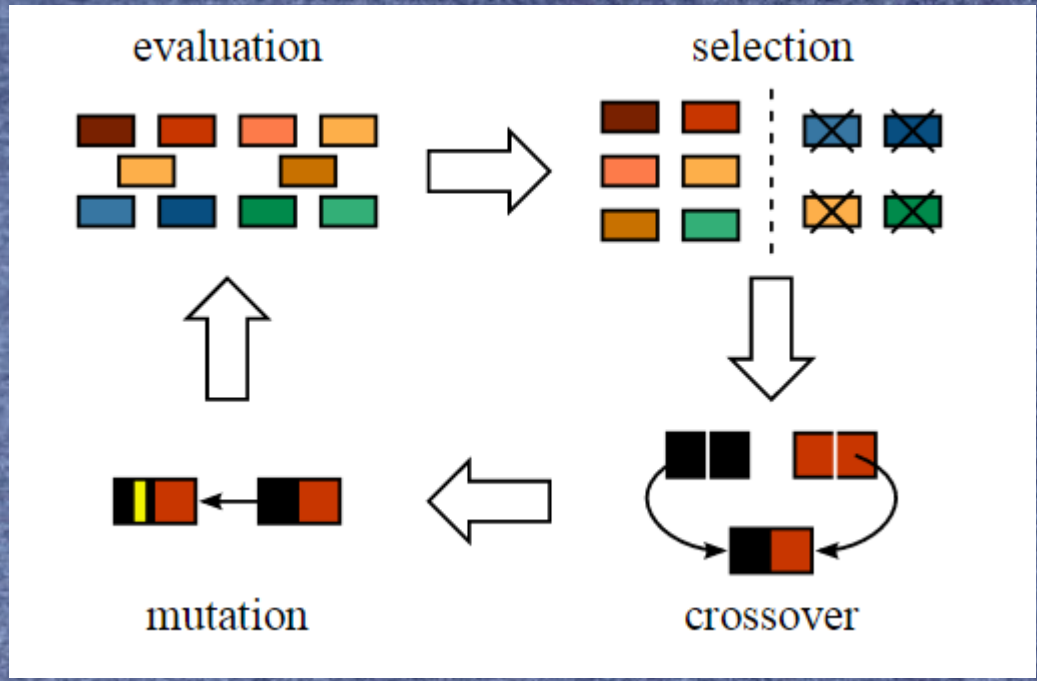




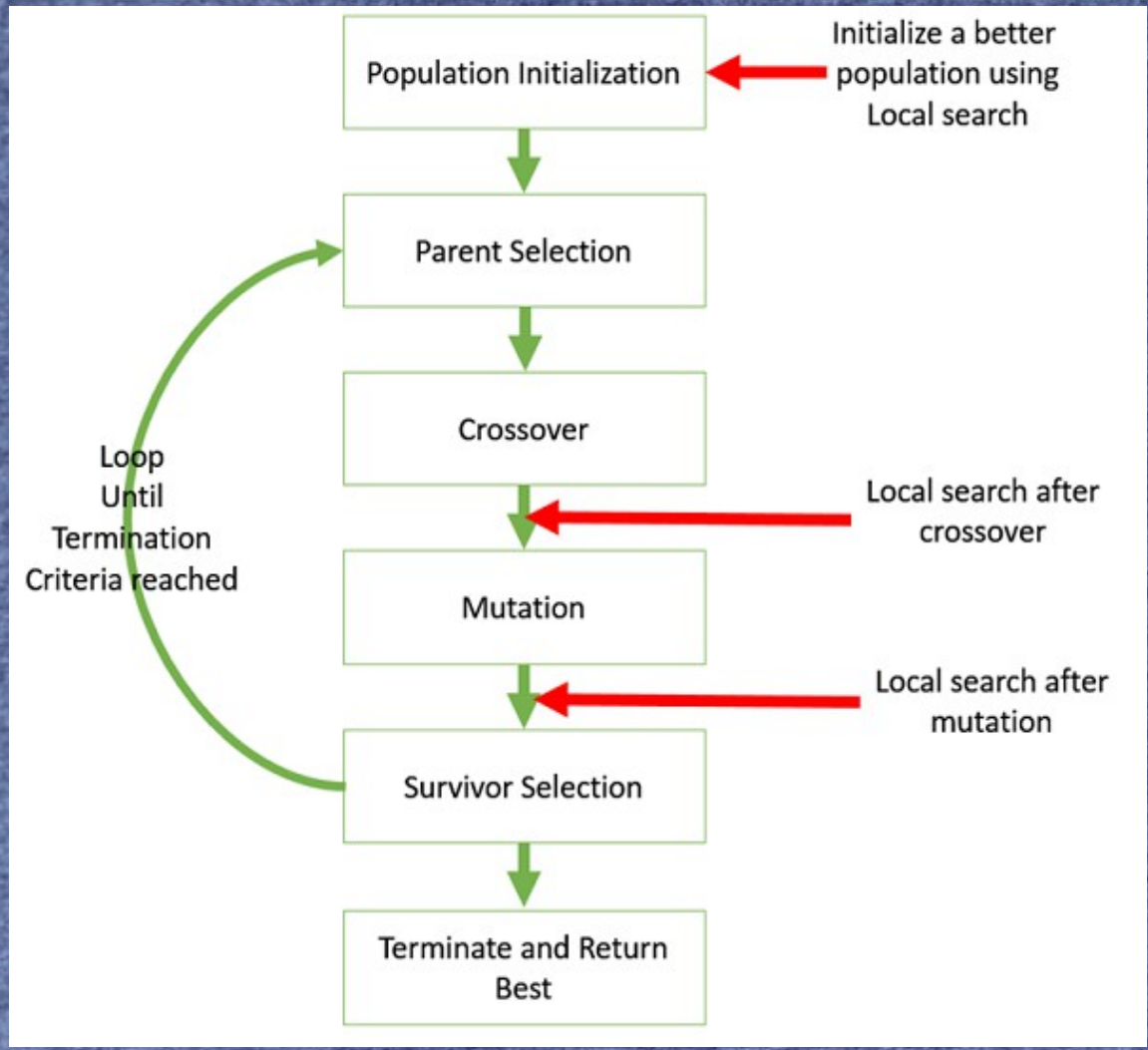
Example 1

Mutation



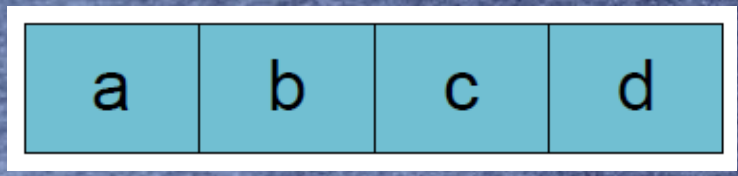


Hybridize GA with Local Search



Example 2

- Use genetic algorithms to solve $a + 2b + 3c + 4d = 30$
- Objective is minimizing the value of function $f(x) = a + 2b + 3c + 4d - 30$
- Since there are four variables in the equation, namely **a**, **b**, **c**, and **d**, we can compose the chromosome as follow:
  - To speed up the computation, we can restrict that the values of variables **a**, **b**, **c**, and **d** are integers between 0 and 30



Example 2

Step 1. Initialization

For example we define the number of chromosomes in population are 6, then we generate random value of gene **a**, **b**, **c**, **d** for 6 chromosomes

Chromosome[1] = [**a**;**b**;**c**;**d**] = [12;05;23;08]

Chromosome[2] = [**a**;**b**;**c**;**d**] = [02;21;18;03]

Chromosome[3] = [**a**;**b**;**c**;**d**] = [10;04;13;14]

Chromosome[4] = [**a**;**b**;**c**;**d**] = [20;01;10;06]

Chromosome[5] = [**a**;**b**;**c**;**d**] = [01;04;13;19]

Chromosome[6] = [**a**;**b**;**c**;**d**] = [20;05;17;01]

Example 2

Step 2. Evaluation

We compute the objective function value for each chromosome produced in initialization step:

F_obj[1] = Abs((12 + 2*05 + 3*23 + 4*08) - 30)
= Abs((12 + 10 + 69 + 32) - 30) = Abs(123 - 30) = 93

F_obj[2] = Abs((02 + 2*21 + 3*18 + 4*03) - 30)
= Abs((02 + 42 + 54 + 12) - 30) = Abs(110 - 30) = 80

F_obj[3] = Abs((10 + 2*04 + 3*13 + 4*14) - 30)
= Abs((10 + 08 + 39 + 56) - 30) = Abs(113 - 30) = 83

Example 2

Step 2. Evaluation

F_obj[4] = Abs((20 + 2*01 + 3*10 + 4*06) - 30)
= Abs((20 + 02 + 30 + 24) - 30) = Abs(76 - 30) = 46

F_obj[5] = Abs((01 + 2*04 + 3*13 + 4*19) - 30)
= Abs((01 + 08 + 39 + 76) - 30) = Abs(124 – 30) = 94

F_obj[6] = Abs((20 + 2*05 + 3*17 + 4*01) - 30)
= Abs((20 + 10 + 51 + 04) - 30) = Abs(85 - 30) = 55

Example 2

Step 3. Selection

To compute fitness probability we must compute the fitness of each chromosome.

Fitness[1] = 1 / (1+**F_obj**[1]) = 1 / 94 = 0.0106

Fitness[2] = 1 / (1+**F_obj**[2]) = 1 / 81 = 0.0123

Fitness[3] = 1 / (1+**F_obj**[3]) = 1 / 84 = 0.0119

Fitness[4] = 1 / (1+**F_obj**[4]) = 1 / 47 = 0.0213

Fitness[5] = 1 / (1+**F_obj**[5]) = 1 / 95 = 0.0105

Fitness[6] = 1 / (1+**F_obj**[6]) = 1 / 56 = 0.0179

Total = 0.0106 + 0.0123 + 0.0119 + 0.0213 + 0.0105 + 0.0179 = 0.0845

Example 2

Step 3. Selection

The probability for each chromosomes is: **P**[i] = **Fitness**[i] / **Total**

P[1] = 0.0106 / 0.0845 = 0.1254

P[2] = 0.0123 / 0.0845 = 0.1456

P[3] = 0.0119 / 0.0845 = 0.1408

P[4] = 0.0213 / 0.0845 = 0.2521

P[5] = 0.0105 / 0.0845 = 0.1243

P[6] = 0.0179 / 0.0845 = 0.2118

Example 2

Step 3. Selection

To use roulette wheel, compute the cumulative probability values:

C[1] = 0.1254

C[2] = 0.1254 + 0.1456 = 0.2710

C[3] = 0.1254 + 0.1456 + 0.1408 = 0.4118

C[4] = 0.1254 + 0.1456 + 0.1408 + 0.2521 = 0.6639

C[5] = 0.1254 + 0.1456 + 0.1408 + 0.2521 + 0.1243 = 0.7882

C[6] = 0.1254 + 0.1456 + 0.1408 + 0.2521 + 0.1243 + 0.2118 = 1.0

Example 2

Step 3. Selection

Generate random number **R** in the range 0-1 :

R[1] = 0.201 **R**[2] = 0.284 **R**[3] = 0.099 **R**[4] = 0.822 **R**[5] = 0.398

R[6] = 0.501

If random number **R**[1] is greater than **C**[1] and smaller than **C**[2] then select **Chromosome**[2] as a chromosome in the new population for next generation:

NewChromosome[1] = **Chromosome**[2]

NewChromosome[2] = **Chromosome**[3]

NewChromosome[3] = **Chromosome**[1]

NewChromosome[4] = **Chromosome**[6]

NewChromosome[5] = **Chromosome**[3]

NewChromosome[6] = **Chromosome**[4]

Example 2

Chromosomes in the population thus became:

Chromosome[1] = [02;21;18;03]

Chromosome[2] = [10;04;13;14]

Chromosome[3] = [12;05;23;08]

Chromosome[4] = [20;05;17;01]

Chromosome[5] = [10;04;13;14]

Chromosome[6] = [20;01;10;06]

Example 2

Step 4: Crossover

Parent chromosome which will mate is randomly selected and the number of mate Chromosomes is controlled using **crossover_rate** (**ρc**) parameters.

Pseudo-code for the crossover process is as follows:

```
begin
    k← 0;
    while(k<population) do
            R[k] = random(0-1);
            if(R[k]< ρc) then
            select Chromosome[k] as parent;
    end;
    k = k + 1;
    end;
    end;
```

Example 2

Step 4: Crossover

Chromosome k will be selected as a parent if R[k]<ρc.

Suppose we set that the crossover rate is 25%, then Chromosome number k will be selected for crossover if random generated value for Chromosome k below 0.25.

The process is as follows: First we generate a random number R as the number of population.

R[1] = 0.191	**R[2] = 0.259**	**R[3] = 0.760**
R[4] = 0.006	**R[5] = 0.159**	**R[6] = 0.340**

Example 2

Step 4: Crossover

For random number **R** above, parents are **Chromosome**[1], **Chromosome**[4] and **Chromosome**[5] will be selected for crossover.

Chromosome[1] >< **Chromosome**[4]

Chromosome[4] >< **Chromosome**[5]

Chromosome[5] >< **Chromosome**[1]

Determine the position of the crossover point. This is done by generating random numbers between 1 to (length of Chromosome – 1).

C[1] = 1

C[2] = 1

C[3] = 2

Example 2

Then for first crossover, second crossover and third crossover, parent's gens will be cut at gen number 1, gen number 1 and gen number 3 respectively, e.g.

Chromosome[1] = **Chromosome**[1] >< **Chromosome**[4]

= [02;21;18;03] >< [20;05;17;01] = [02;05;17;01]

Chromosome[4] = **Chromosome**[4] >< **Chromosome**[5]

= [20;05;17;01] >< [10;04;13;14] = [20;04;13;14]

Chromosome[5] = **Chromosome**[5] >< **Chromosome**[1]

= [10;04;13;14] >< [02;21;18;03] = [10;04;18;03]

Example 2

Thus Chromosome population after experiencing a crossover process:

Chromosome[1] = [02;05;17;01]

Chromosome[2] = [10;04;13;14]

Chromosome[3] = [12;05;23;08]

Chromosome[4] = [20;04;13;14]

Chromosome[5] = [10;04;18;03]

Chromosome[6] = [20;01;10;06]

Example 2

Step 5. Mutation

Number of chromosomes that have mutations in a population is determined by the mutation_rate parameter.

First calculate the total length of gen in the population.

Total length of gen is total_gen = number_of_gen_in_Chromosome * number of population = 4 * 6 = 24

Mutation process is done by generating a random integer between 1 and total_gen (1 to 24).

If generated random number is smaller than mutation_rate(ρm) variable then marked the position of gen in chromosomes.

Suppose we define ρm 10%, it is expected that 10% (0.1) of total_gen in the population that will be mutated: number of mutations = 0.1 * 24 = 2.4 ≈ 2

Example 2

Step 5. Mutation

Suppose generation of random number yield 12 and 18 then the chromosome which have mutation are Chromosome number 3 gen number 4 and Chromosome 5 gen number 2. The value of mutated gens at mutation point is replaced by random number between 0-30. Suppose generated random number are 2 and 5 then Chromosome composition after mutation are:

Chromosome[1] = [02;05;17;01]

Chromosome[2] = [10;04;13;14]

Chromosome[3] = [12;05;23;02]

Chromosome[4] = [20;04;13;14]

Chromosome[5] = [10;05;18;03]

Chromosome[6] = [20;01;10;06]

Example 2

Step 6: Evaluate the objective function

Chromosome[1] = [02;05;17;01]

F_obj[1] = Abs((02 + 2*05 + 3*17 + 4*01) - 30)
= Abs((2 + 10 + 51 + 4) - 30) = Abs(67 - 30) = 37

Chromosome[2] = [10;04;13;14]

F_obj[2] = Abs((10 + 2*04 + 3*13 + 4*14) - 30)
= Abs((10 + 8 + 33 + 56) - 30) = Abs(107 - 30) = 77

Chromosome[3] = [12;05;23;02]

F_obj[3] = Abs((12 + 2*05 + 3*23 + 4*02) - 30)
= Abs((12 + 10 + 69 + 8) - 30) = Abs(87 - 30) = 47

Example 2

Step 6: Evaluate the objective function

Chromosome[4] = [20;04;13;14]

F_obj[4] = Abs((20 + 2*04 + 3*13 + 4*14) - 30)
= Abs((20 + 8 + 39 + 56) - 30) = Abs(123 - 30) = 93

Chromosome[5] = [10;05;18;03]

F_obj[5] = Abs((10 + 2*05 + 3*18 + 4*03) - 30)
= Abs((10 + 10 + 54 + 12) - 30) = Abs(86 - 30) = 56

Chromosome[6] = [20;01;10;06]

F_obj[6] = Abs((20 + 2*01 + 3*10 + 4*06) - 30)
= Abs((20 + 2 + 30 + 24) - 30) = Abs(76 - 30) = 46

Example 2

New Chromosomes for next iteration are:

Chromosome[1] = [02;05;17;01]

Chromosome[2] = [10;04;13;14]

Chromosome[3] = [12;05;23;02]

Chromosome[4] = [20;04;13;14]

Chromosome[5] = [10;05;18;03]

Chromosome[6] = [20;01;10;06]

Example 2

For this example, after running 50 generations, best chromosome is obtained:

Chromosome = [07; 05; 03; 01]

This means that: **a** = 7, **b** = 5, **c** = 3, **d** = 1

If we use the number in the problem equation:

a + 2**b** + 3**c** + 4**d** = 30

7 + (2 * 5) + (3 * 3) + (4 * 1) = 30

THANK YOU!