Computational Intelligence & Machine Learning

Particle Swarm Optimization (PSO)

Adopted from Mohammed Al-Alaw & Qiangfu Zhao

Introduction

- Inspired by the flocking and schooling patterns of birds and fish.
- Imagine a flock of birds circling over an area where they can smell a hidden source of food.
- The one who is closest to the food chirps the loudest and the other birds swing around in his direction.
- If any of the other circling birds comes closer to the target than the first, it chirps louder and the others veer over toward him.
- This tightening pattern continues until one of the birds happens upon the food.

Introduction

Introduction

- **Particle Swarm Optimization (PSO)** was invented by Russell Eberhart and James Kennedy in 1995.

- Originally, these two started out developing computer software simulations of birds flocking around food sources

- They realized how well their algorithms worked on optimization problems.

- Over a number of iterations, a group of variables have their values adjusted closer to the member whose value is closest to the target at any given moment.

- It's an algorithm that's simple and easy to implement.

Introduction

- In computer science, Particle Swarm Optimization (PSO) is a computational method that optimizes a problem by iteratively trying to improve a **candidate solution** with regard to a given measure of quality (This is the **stopping Condition**).

- PSO optimizes a problem by having a population of candidate solutions, (known as **particles**), and moving these particles around in the search-space

- It moves according to simple mathematical formulae over the particle's **position** (Current DATA ex: x,y,z, etc…) and **velocity** (indicating how much the Data can be changed).

Introduction

- The algorithm was simplified and it was observed to be performing optimization (first it was not intended to be used in this manner).

- PSO is a **metaheuristic** as it makes few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions.

- However, metaheuristics such as PSO do not guarantee an optimal solution is ever found.

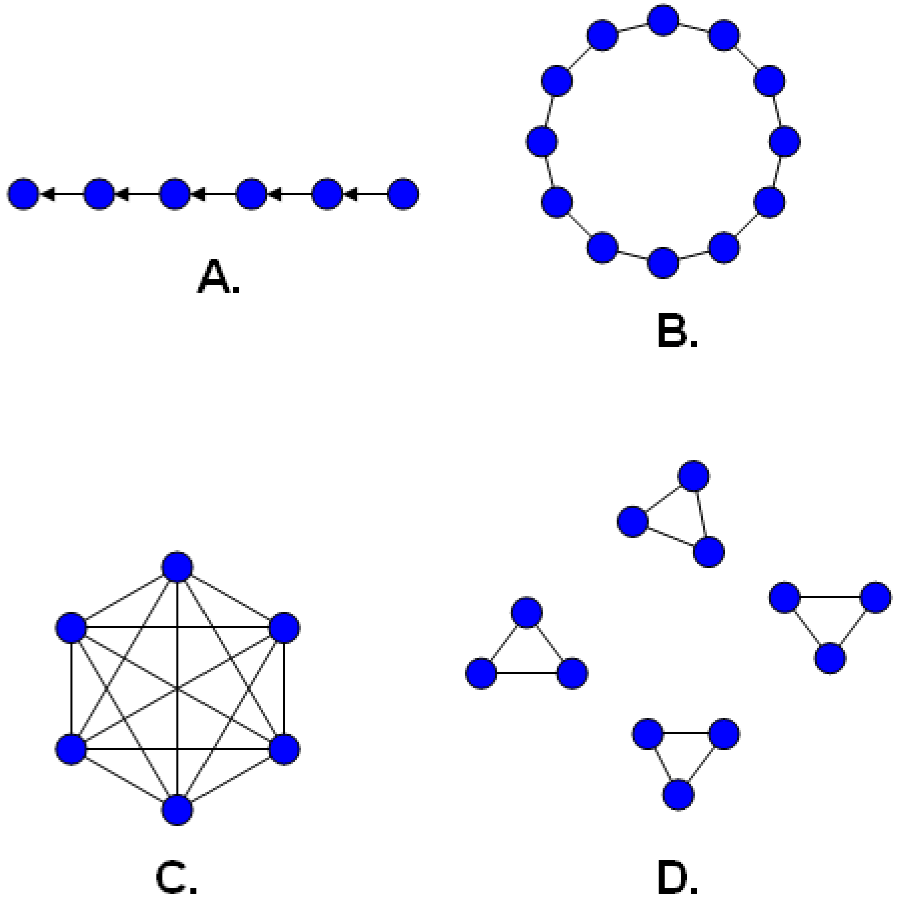
Introduction

- Each particle's movement is influenced by its **local best** known position but, is also guided toward the **best known positions in the search-space**

- The best positions are updated as better positions when they are found by other particles

- This is expected to move the swarm toward the best solutions.

Introduction



A few common population topologies (neighborhoods).
(A) Single-sighted. (B) Ring topology. (C) Fully connected topology. (D) Isolated,

Introduction

- PSO does not use the gradient of the problem being optimized, which means PSO does not require that the optimization problem be differentiable as is required by classic optimization methods

- To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or of the approximate gradient) of the function at the current point and quasi-newton methods.

- PSO can therefore also be used on optimization problems that are partially irregular, noisy, change over time, etc., i.e. ,they are used for real time & data analysis & applications.

The Algorithm

- The algorithm keeps track of three global variables:
  - Target value or condition
  - Global best (**gBest**) value indicating which particle's data is currently closest to the Target

- Stopping value indicating when the algorithm should stop if the Target isn't found

- Each particle consists of:
  - Data representing a possible solution
  - A Velocity value indicating how much the data can be changed
  - A personal best (**pBest**) value indicating the closest the particle's Data has ever come to the Target

The Algorithm

- The particles' data could be anything. In the flocking birds example above, the data would be the X, Y, Z coordinates of each bird.

- The individual coordinates of each bird would try to move closer to the coordinates of the bird which is closer to the food's coordinates (gBest).

- If the data is a pattern or sequence, then individual pieces of the data would be manipulated until the pattern matches the target pattern.

The Algorithm

- The **velocity** value is calculated according to how far an individual's data is from the target. The further it is, the larger the velocity value.

- In the birds example, the individuals furthest from the food would make an effort to keep up with the others by flying faster toward the gBest bird.

- If the data is a pattern or sequence, the velocity would describe how different the pattern is from the target, and thus, how much it needs to be changed to match the target (making it similar to Neural Networks).

The Algorithm

- Each particle's pBest value only indicates the closest the data has ever come to the target since the algorithm started.

- The gBest value only changes when any particle's pBest value comes closer to the target than gBest.

- Through each iteration of the algorithm, gBest gradually moves closer and closer to the target until one of the particles reaches the target.

- It's also common to see PSO algorithms using population topologies, or "**neighborhoods**", which can be smaller, localized subsets of the global best value.

The Algorithm

- Neighborhoods can involve two or more particles which are predetermined to act together, or subsets of the search space that particles happen into during testing.

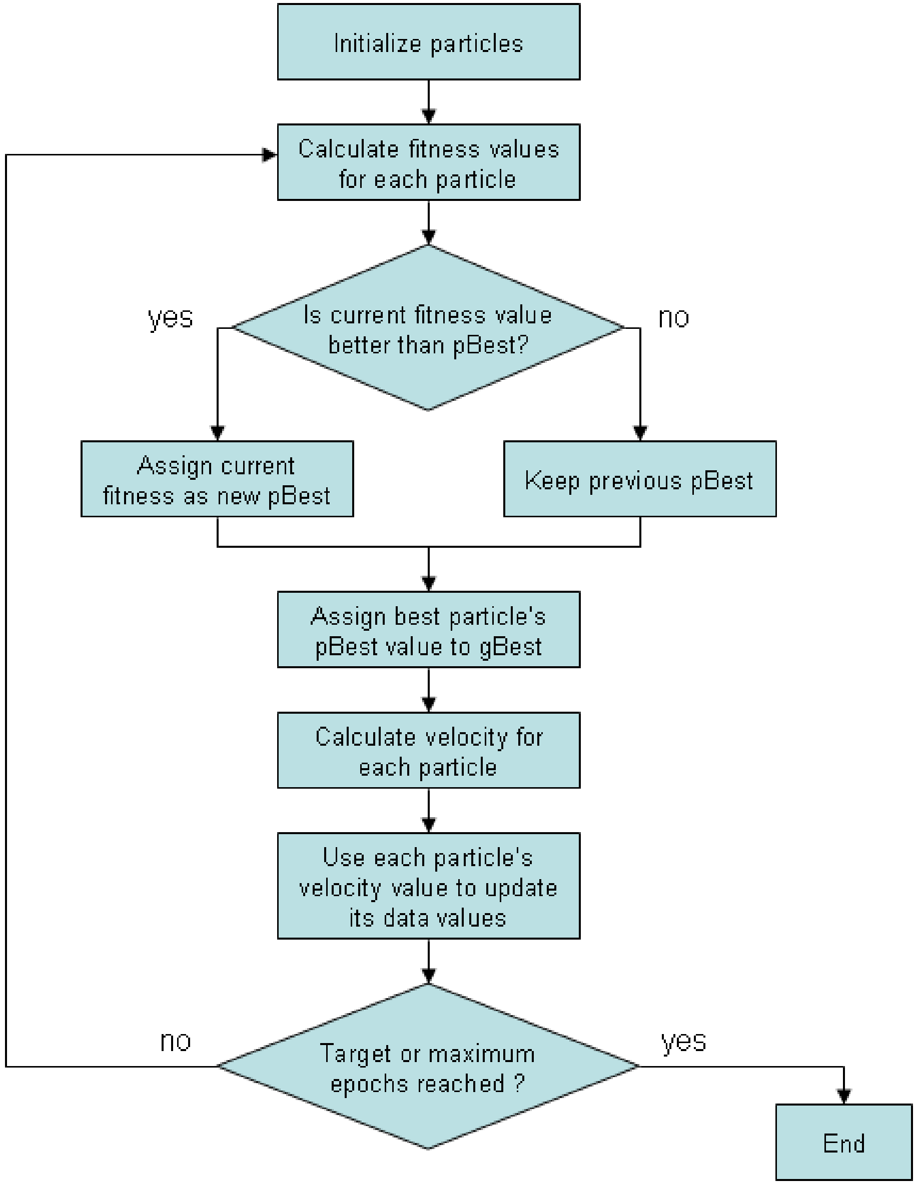
- The use of neighborhoods often help the algorithm to avoid getting stuck in local minima.

- Neighborhood definitions and how they're used have different effects on the behavior of the algorithm.

The Algorithm

- Stopping Conditions:
  - Terminate when a maximum number of iterations, or FEs, has been exceeded
  - Terminate when an acceptable solution has been found
  - Terminate when no improvement is observed over a number of iterations
  - Terminate when the normalized swarm radius is close to zero

The Algorithm



The Algorithm

- Step 1: Randomly initialize the swarm.
- Step 2: Evaluate all particles.
- Step 3: For each particle
  - Update its velocity;
  - Update its position;
  - Evaluate the particle.
- Step 4: Update if necessary the leader of the swarm and the best position obtained by each particle.
- Step 5: Stop if terminating condition satisfied; return to Step 3 otherwise.

The Algorithm

- The velocity of a particle is updated as follows:

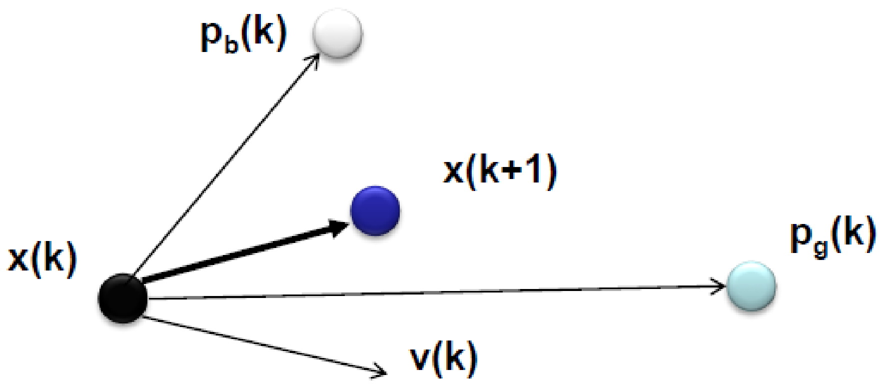
$$\mathbf{v}^{new} = a\mathbf{v}^{old} + bw_1 \times (\mathbf{x}_{my\_best} - \mathbf{x}^{old}) + cw_2 \times (\mathbf{x}_{best} - \mathbf{x}^{old})$$

where a is the inertia weight, b and c are the learning factors called personal factor and social factor, respectively, and w_1 and w_2 are random numbers taken from $[0,1]$.

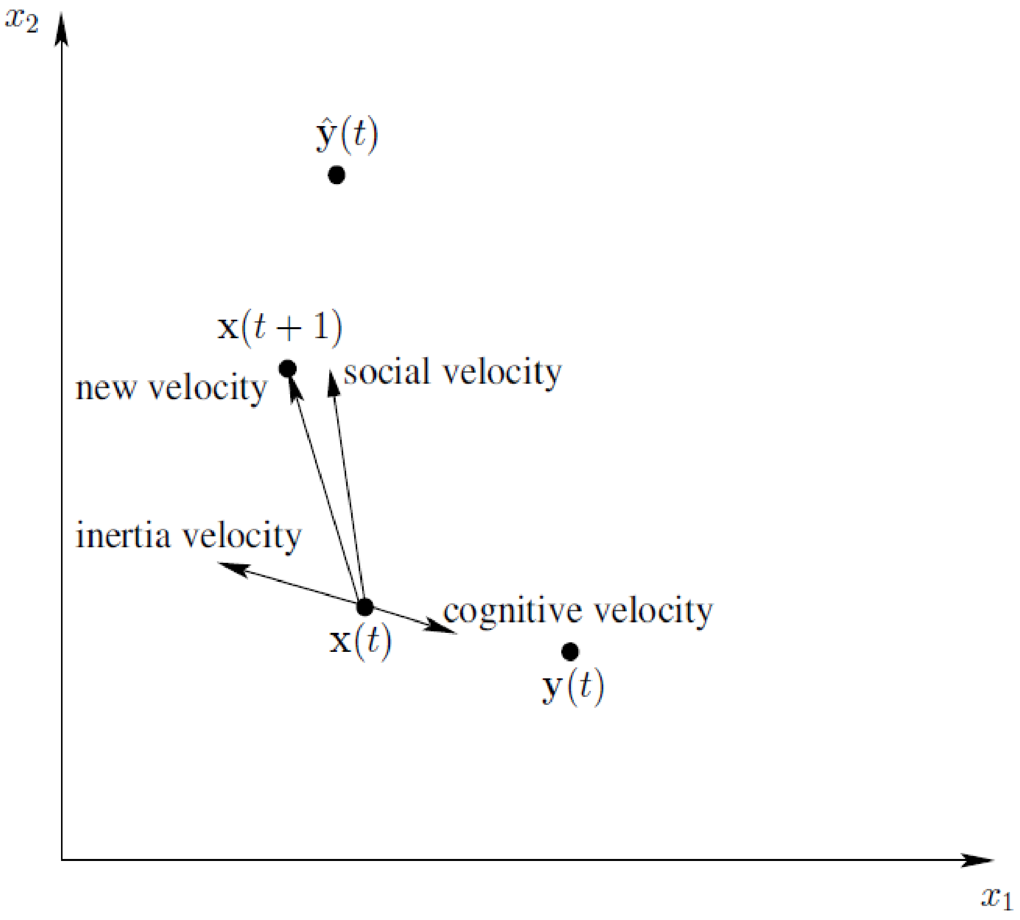
The Algorithm

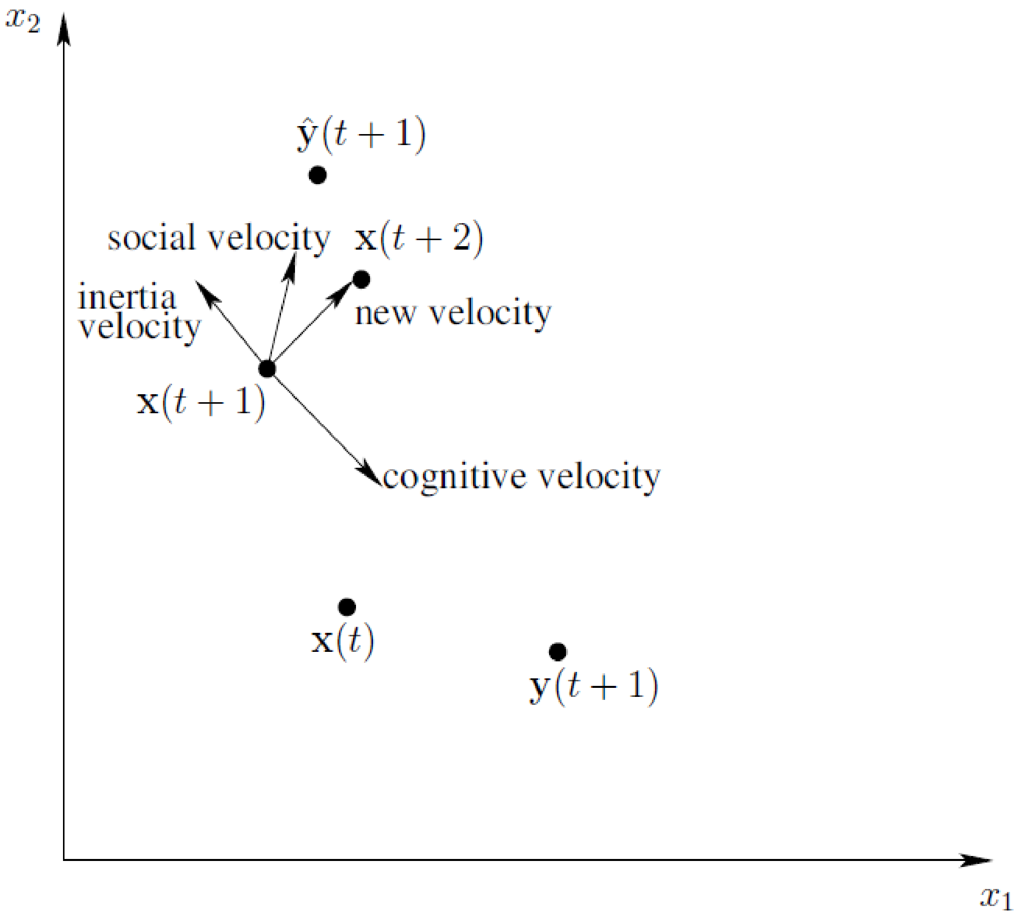
Based on the new velocity, the new position is obtained as follows:

$$\mathbf{x}^{new} = \mathbf{x}^{old} + \mathbf{v}^{new}$$

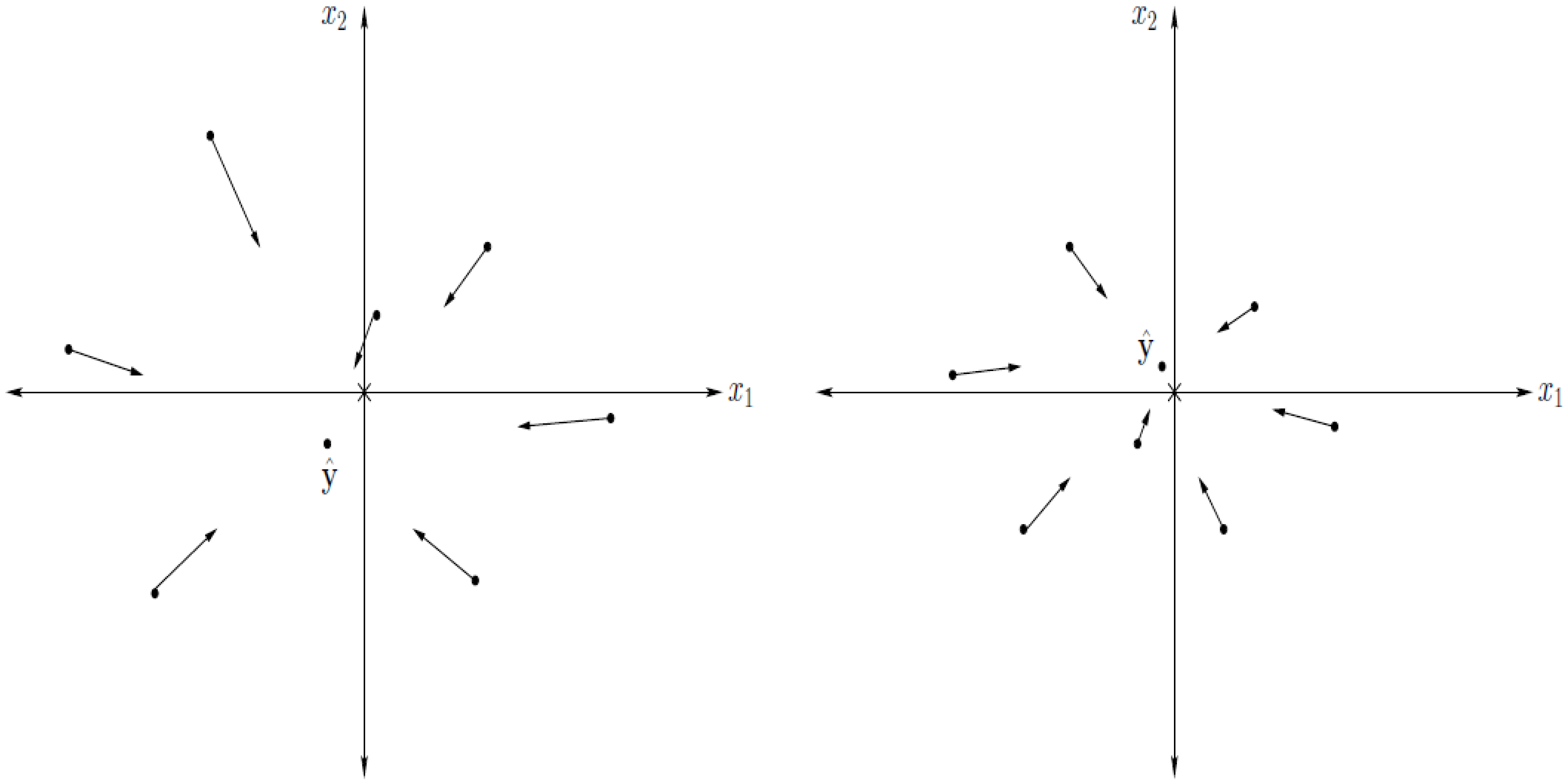


The Algorithm

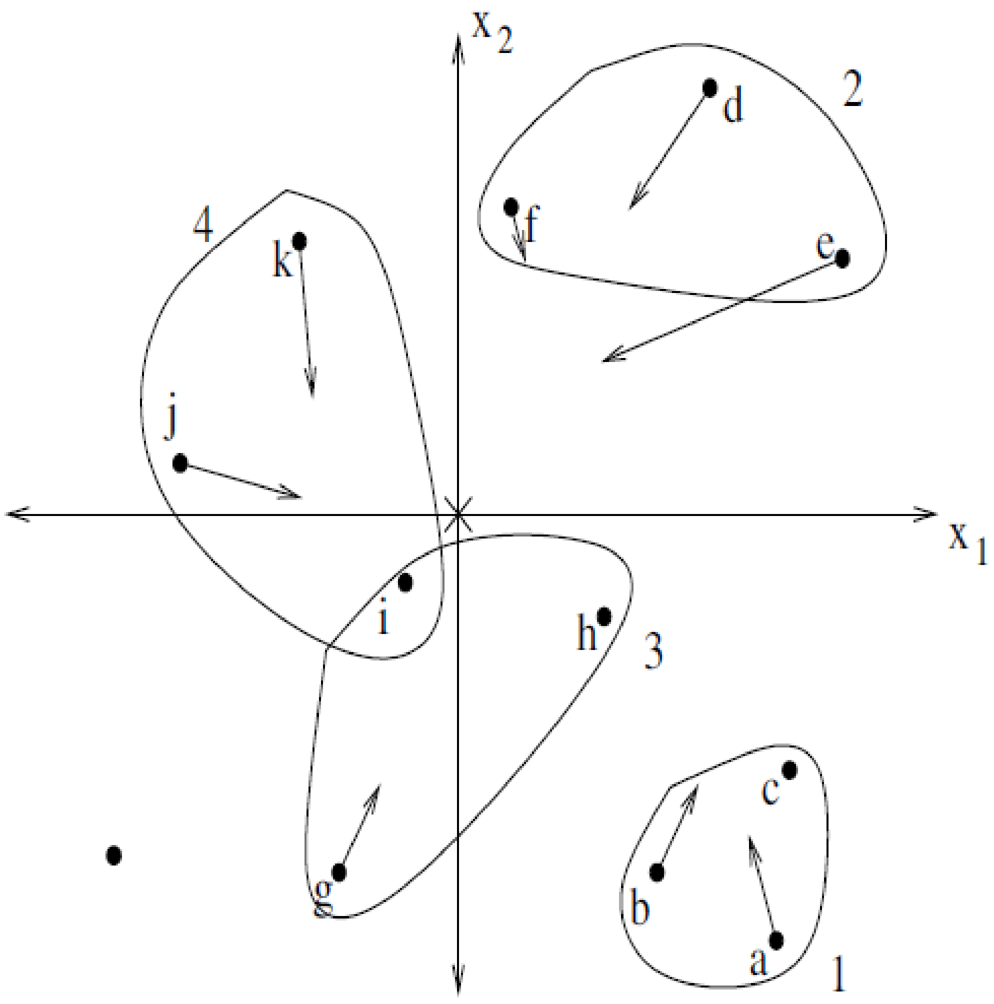




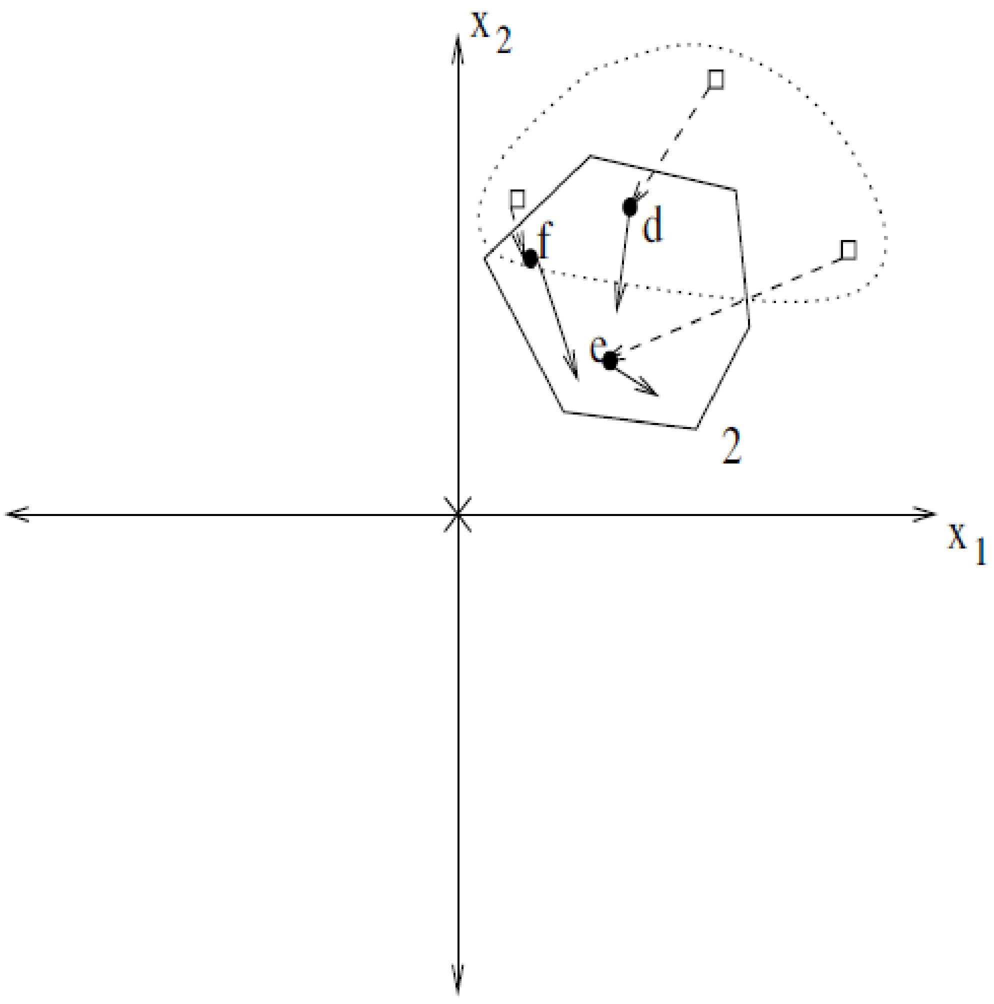
The Algorithm



The Algorithm



(a) Local Best Illustrated – Initial Swarm



(b) Local Best – Second Swarm

The Algorithm

- Approaches to update the inertia weight
  - **Random adjustments**, where a different inertia weight is randomly selected at each iteration, e.g., ~ N(0.72, σ) where *σ* is small enough to ensure that *w* ***(inertia weight)*** is not predominantly greater than one
  - **Linear decreasing** where an initially large inertia weight (usually 0.9) is linearly decreased to a small value (usually 0.4)

$$w(t) = (w(0) - w(n_t))\frac{(n_t - t)}{n_t} + w(n_t)$$

  - **Nonlinear decreasing**, where an initially large value decreases nonlinearly to a small value

$$w(t+1) = \frac{(w(t) - 0.4)(n_t - t)}{n_t + 0.4}$$

  - **Fuzzy adaptive inertia**, where the inertia weight is dynamically adjusted on the basis of fuzzy sets and rules

Visualization and Examples

https://pypi.org/project/swarmlib/

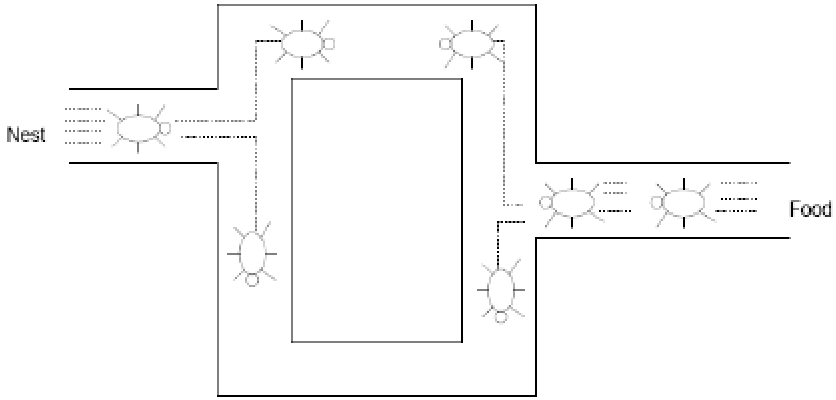
https://nathanrooy.github.io/posts/2016-08-17/simple-particle-swarm-optimization-with-python/

Ant Colony Optimization (ACO)

Adopted from Michael Herrmann

Introduction

Biological inspiration: ants find the shortest path between their nest and a food source using **pheromone trails**.



Ant Colony Optimisation is a population-based search technique for the solution of combinatorial optimisation problems which is inspired by this behaviour.

Introduction

- Real ants find shortest routes between food and nest
- They hardly use vision (almost blind)
- They lay pheromone trails, chemicals left on the ground, which act as a signal to other ants – **STIGMERGY**
- If an ant decides, with some probability, to follow the pheromone trail, it itself lays more pheromone, thus reinforcing the trail.
- The more ants follow the trail, the stronger the pheromone, the more likely ants are to follow it.
- Pheromone strength decays over time (half-life: a few minutes)
- Pheromone builds up on shorter path faster (it doesn't have so much time to decay), so ants start to follow it.

Introduction

stigma (mark, sign) +
ergon (work, action)

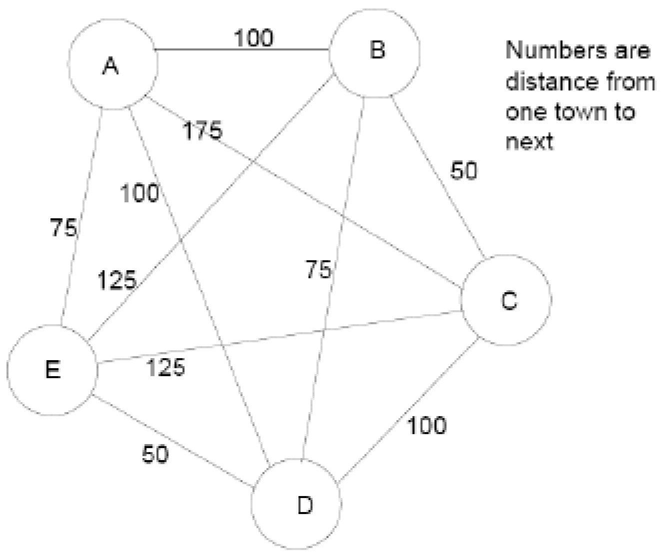
Pierre-Paul Grassé
(1959)

Artificial Ant Systems

- Do have some memory (data structures)
- Are able to sense "environment" if necessary (not just pheromone)
- Use discrete time
- Are optimisation algorithms

So can we apply them to an optimisation problem: Travelling Salesperson Problem

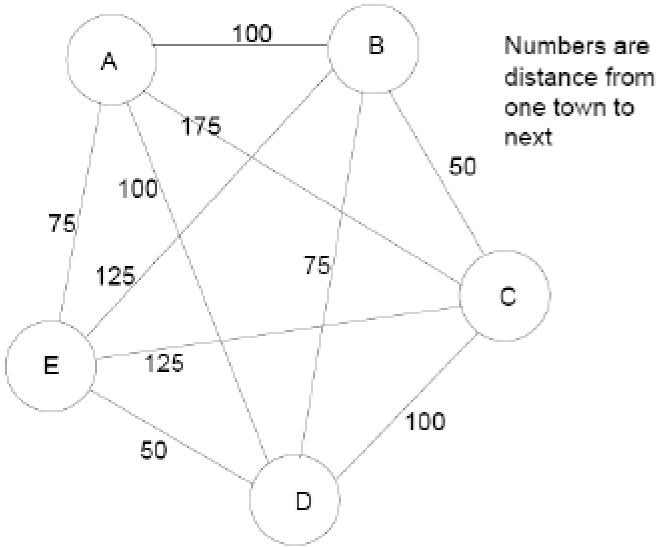
Example in TSP



Find the tour that minimises the distance travelled in visiting all towns.

Example in TSP

- Each ant builds its own tour from a starting city
- Each ant chooses a town to go to with a probability: this is a function of the town's distance and the amount of pheromone on the connecting edge
- Legal tours: transitions to already visited towns disallowed till tour complete (keep a tabu list)
- When tour completed, lay pheromone on each edge visited
- Next city j after city i chosen according to Probability Rule



Example in TSP

- While building tour, apply an improvement heuristic at each step to each ant's partial tour.
- For example: use 3-opt: cut the tour in three places (remove three links) and attempt to connect up the cities in alternative ways that shorten the path.
- Reduces time, almost always finds optimal path.

Probability Rule

$$p(i,j) = \frac{[\tau(i,j)].[\eta(i,j)]^{\beta}}{\sum_{g \in \text{allowed}} [\tau(i,g)].[\eta(i,g)]^{\beta}}$$

- Strength of pheromone $\tau(i,j)$ is favourability of j following i
  Emphasises "**global** goodness": the **pheromone matrix**
- Visibility $\eta(i,j) = 1/d(i,j)$ is a simple heuristic guiding construction of the tour. In this case it's greedy – the nearest town is the most desirable (seen from a **local** point of view)
- β is a constant, e.g. 2
- $\sum_{g \in \text{allowed}}$: normalise over all the towns g that are still permitted to be added to the tour, i.e. not on the tour already
- So τ and η trade off global and local factors in construction of tour

Pheromone

- Pheromone trail evaporates a small amount after every iteration

$$\tau(i,j) = \rho.\tau(i,j) + \Delta\tau_{ij}$$

where $0 < \rho < 1$ is an evaporation constant

- The density of pheromone laid on edge (i,j) by the m ants at that timestep is

$$\Delta\tau_{ij} = \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$$

- $\Delta\tau_{ij}^{k} = Q/L_k$ if kth ant uses edge (i,j) in its tour, else 0. Q is a constant and L_k is the length of k's tour. Pheromone density for k's tour.

Pheromone

- Initialise: set pheromone strength to a small value
- Transitions chosen to trade off visibility (choose close towns with high probability – greedy) and trail intensity (if there's been a lot of traffic the trail must be desirable).
- In one iteration all the ants build up their own individual tours (so an iteration consists of lots of moves/town choices/timesteps – until the tour is complete) and pheromone is laid down once all the tours are complete
- Remember: we're aiming for the shortest tour – and expect pheromone to build up on the shortest tour faster than on the other tours

Algorithm

- Position ants on different towns, initialise pheromone intensities on edges.
- Set first element of each ant's tabu list to be its starting town.
- Each ant moves from town to town according to the probability $p(i, j)$
- After n moves all ants have a complete tour, their tabu lists are full; so compute L_k and $\Delta\tau_{ij}^k$. Save shortest path found and empty tabu lists. Update pheromone strengths.
- Iterate until tour counter reaches maximum or until *stagnation* – all ants make same tour.

Can also have different pheromone-laying procedures, e.g. lay a certain quantity of pheromone Q at each timestep, or lay a certain density of pheromone Q/d_{ij} at each timestep.

The ACO Algorithm

Algorithm 1 The framework of a basic ACO algorithm

```
input: An instance P of a CO problem model 𝒫 = (𝒮, f, Ω).
InitializePheromoneValues(𝒯)
𝔰_bs ← NULL                                              init best-so-far solution
while termination conditions not met do
    𝔖_iter ← ∅                                           set of valid solutions
    for j = 1, ..., n_a do                               loop over ants
        𝔰 ← ConstructSolution(𝒯)
        if 𝔰 is a valid solution then
            𝔰 ← LocalSearch(𝔰)          {optional}
            if (f(𝔰) < f(𝔰_bs)) or (𝔰_bs = NULL) then 𝔰_bs ← 𝔰    update best-so-far
            𝔖_iter ← 𝔖_iter ∪ {𝔰}                      store valid solutions
        end if
    end for
    ApplyPheromoneUpdate(𝒯,𝔖_iter,𝔰_bs)
end while
output: The best-so-far solution 𝔰_bs
```

Applications

- Bus routes, garbage collection, delivery routes
- Machine scheduling: Minimization of transport time for distant production locations
- Feeding of lacquering machines
- Protein folding
- Telecommunication networks: Online optimization
- Personnel placement in airline companies
- Composition of products

Performance

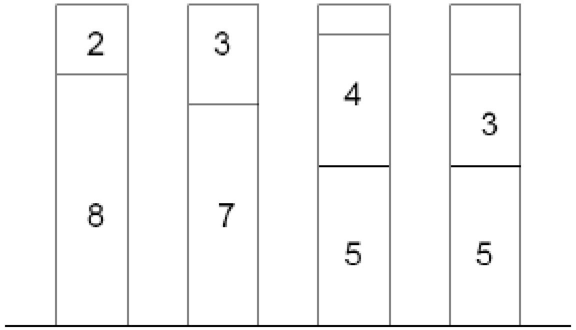
Problem	ACS (avge)	SA (avge)	EN (avge)	SOM (avge)
50-city set 1	5.88	5.88	5.98	6.06
50-city set 2	6.05	6.01	6.03	6.25
50-city set 3	5.58	5.65	5.70	5.83
50-city set 4	5.74	5.81	5.86	5.87
50-city set 5	6.18	6.33	6.49	6.70

ACS – ant colony system, SA–simulated annealing, EN–elastic net, SOM–self-organising map

From Dorigo and Gambardella: Ant Colony System: A cooperative learning approach to the TSP. IEEE Trans. Evol. Comp 1 (1) 53–66 1997.

Can do larger problems, e.g. finds optimal in 100-city problem KroA100, close to optimal on 1577-city problem fl1577.

Bin Packing Problems

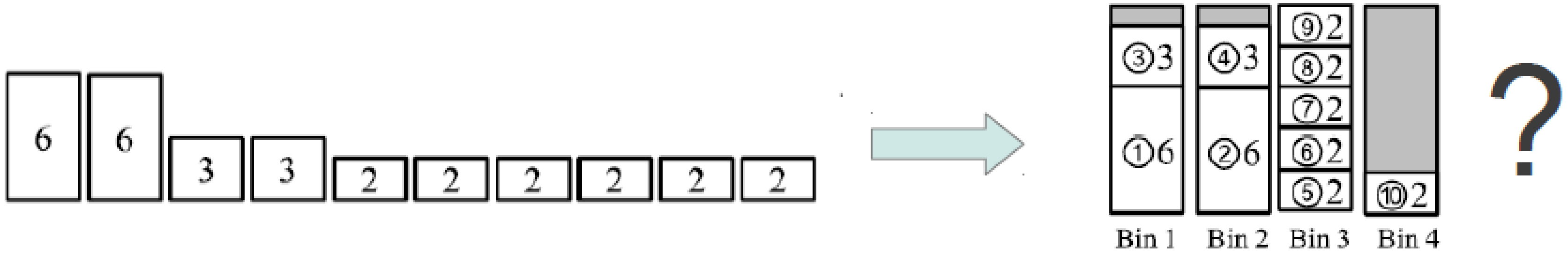


- Packing a number of items in bins of a fixed capacity
- Bins have capacity C, set of items S with size/weight w_i
- Pack items into as few bins as possible
- Lower bound on no. bins: $L_1 = \lceil \sum w_i / C \rceil$ ($\lceil x \rceil$ is smallest integer $\geq x$)
- Slack $= L_1 C - \sum w_i$

Solving the BPP

- Greedy algorithm: first fit decreasing (FFD):
  - Order items in order of non-increasing weight/size
  - Pick up one by one and place into first bin that is still empty enough to hold them
  - If no bin is left that the item can fit in, start a new bin

- Or apply Ant Colony Optimisation: what is the trail/pheromone? what is the "visibility"?



Applying ACO to the BPP

1. How can good packings be reinforced via a pheromone matrix?
2. How can the solutions be constructed stochastically, with influence from the pheromone matrix and a simple heuristic?
3. How should the pheromone matrix be updated after each iteration?
4. What fitness function should be used to recognised good solutions?
5. What local search technique should be used to improve the solutions generated by the ants?

Pheromone Matrix

- BPP as an ordering problem? TSP is an ordering problem – put cities into some order. But in BPP many orderings are possible:

$$\begin{aligned} &|82|73|54|53| \\ = &|53|73|82|54| \\ = &|35|73|28|54| \end{aligned}$$

- BPP as a grouping problem? $\tau(i, j)$ expresses the favourability of having items of size i and j in the same bin - possibly
- Pheromone matrix works on item sizes, not items themselves
- There can be several items of **size** i or j, but there are fewer item sizes than there are items, so small pheromone matrix
- Pheromone matrix encodes good packing patterns – combinations of sizes

Building Solutions

- Every ant k starts with an empty bin b
- New items j are added to k's partial solution s stochastically:

$$p_k(s,b,j) = \frac{[\tau_b(j)]^\alpha.[\eta(j)]^\beta}{\sum_{g \in \text{allowed}}[\tau_b(g)]^\alpha.[\eta(g)]^\beta}$$

- The allowed items are those that are still small enough to fit in bin b.
- $\eta(j)$ is the weight/size of the item, so $\eta(j) = j$ – prefer largest
- $\tau_b(j)$ is the sum of pheromone between item of size j and the items already in bin b divided by the number of items in bin b
- α and β are empirical parameters, e.g. 1 and 2, giving the relative weighting of local and global terms

Pheromone Updating

- Pheromone trail evaporates a small amount after every iteration (i.e. when all ants have solutions)

$$\tau(i, j) = \rho . \tau(i, j) + m . f(s_{\text{best}})$$

- Minimum pheromone level set by parameter τ_{min}, evaporation parameter ρ
- The pheromone is increased for every time items of size i and j are combined in a bin in the best solution (combined m times)
- Only the iteration best ant increases the pheromone trail (quite aggressive, but allows exploration)
- Occasionally (every γ iterations) update with the global best ant instead (strong exploitation)

Evaluation Function

- Total number of bins in solution? Would give an extremely unfriendly evaluation landscape – no guidance from $N+1$ bins to N bins – there may be many possible solutions with just one bin more than the optimal
- Need large reward for full or nearly full bins

$$f(s_k) = \frac{\sum_{b=1}^{N}(F_b/C)^2}{N}$$

N the number of bins in s_k, F_b the sum of items in bin b, C the bin capacity

- Includes how full the bins are and number of bins
- Promotes full bins with the spare capacity in one "big lump" not spread among lots of bins

Local Search

- In every ant's solution, the n_{bins} least full bins are opened and their contents are made free
- Items in the remaining bins are replaced by larger free items
- This gives fuller bins with larger items and smaller free items to reinsert
- The free items are reinserted via FFD (first-fit-decreasing)
- The procedure is repeated until no further improvement is possible
- Deterministic and fast local search procedure
- ACO gives coarse-grained search, local search gives finer-grained search

Setting the Parameters

- Ducatelle used 10 existing problems for which solutions known to investigate parameter setting
- $\beta = 2$
- $n_{\text{ants}} = 10$
- $n_{\text{bins}} = 3$ to be opened in local search
- $\tau_{\min} = 0.001$
- $\rho = 0.75$
- Alternate global and iteration best ant laying pheromone 1/1
- $n_{\text{iter}} = 50000$
- Local search: replace 2 current items by 2 free items; then 2 current by 1 free; then 1 current by 1 free

Applying ACO to Optimization

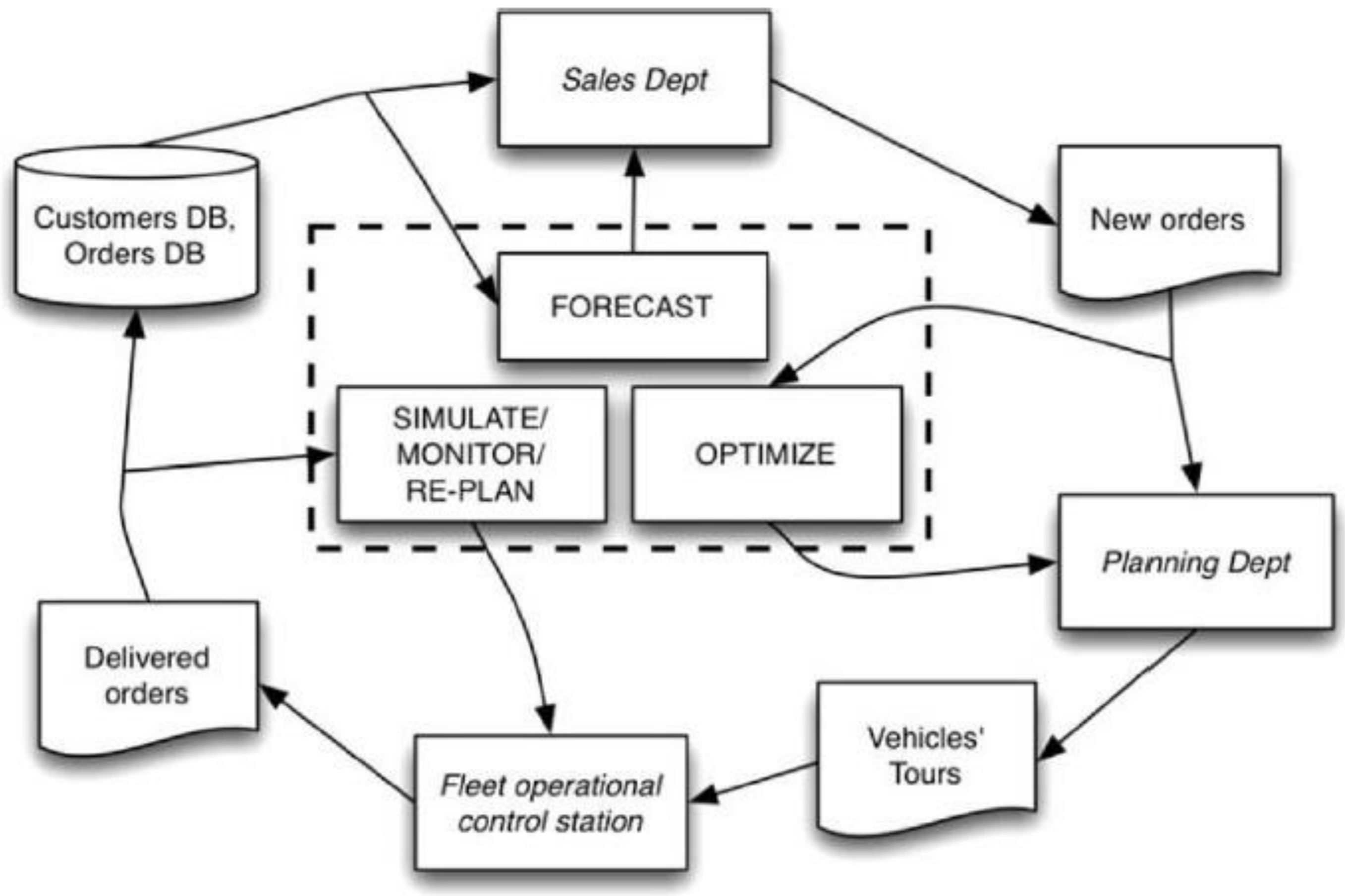
What we need to set up an ACO

- *Problem representation* that allows the solution to be built up incrementally
- *Desirability heuristic* η to help in building up the solution
- *Constraints* that permit only feasible/valid solutions to be constructed
- *Pheromone update rule* incorporating quality of the solution
- *Probability rule* that is a function of desirability and pheromone strength

Considerations

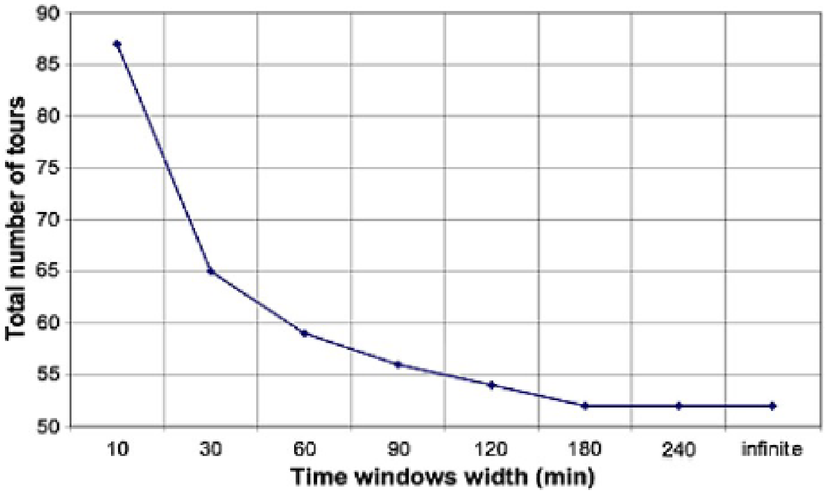
- Best ant laying pheromone (global-best ant or, in some versions of ACO, iteration-best ant) encourage ants to follow the best tour or to search in the neighbourhood of this tour (make sure that $\tau_{min}>0$).
- Local updating (the ants lay pheromone as they go along without waiting till end of tour). Can set up the evaporation rate so that local updating "eats away" pheromone, and thus visited edges are seen as less desirable, encourages exploration. (Because the pheromone added is quite small compared with the amount that evaporates.)
- Heuristic improvements like 3-opt – not really "ant"-style
- "Guided parallel stochastic search in region of best tour" [Dorigo and Gambardella], i.e. assuming a non-deceptive problem.

Vehicle Routing



Vehicle Routing

- E.g. distribute 52000 pallets to 6800 customers over a period of 20 days
- Dynamic problem: continuously incoming orders
- Strategic planning: Finding feasible tours is hard
- Computing time: 5 min (3h for human operators)
- More tours required for narrower arrival time window
- Implicit knowledge on traffic learned from human operators



	Human planner	AR-RegTW	AR-Free
Total number of tours	2056	1807	1614
Total km	147271	143983	126258
Average truck loading	76.91%	87.35%	97.81%

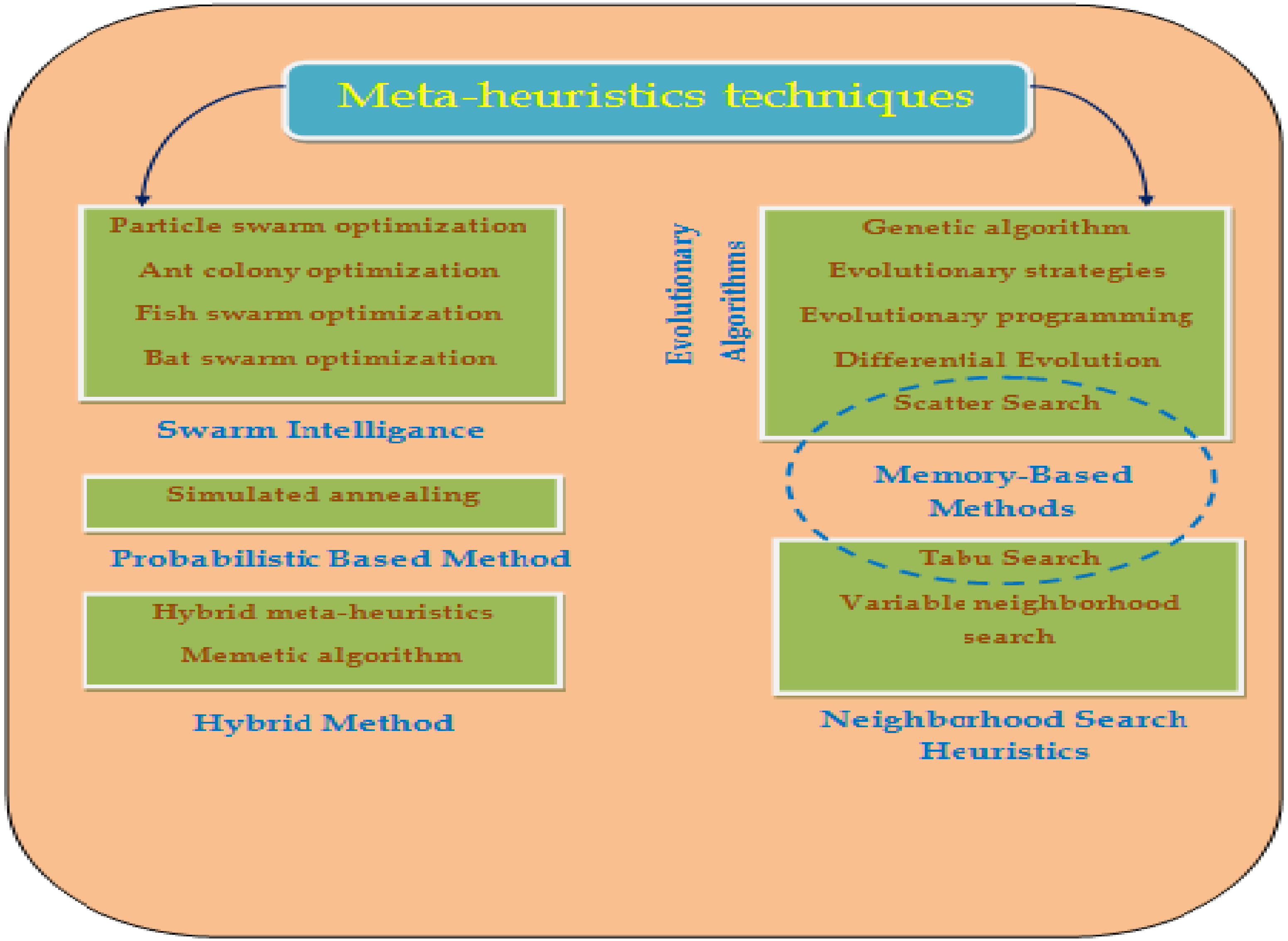
The ACO Algorithm

http://thiagodnf.github.io/aco-simulator/#

Artificial Bee Colony (ABC)

Adopted from Ahmed Fouad Ali

Metaheuristics



Introduction

- Artificial Bee Colony (ABC) is one of the most recently defined algorithms by Dervis Karaboga in 2005, motivated by the intelligent behavior of honey bees.
- Since 2005, D. Karaboga and his research group have studied on ABC algorithm and its applications to real world-problems.

Main Idea

- The ABC algorithm is a swarm based meta-heuristics algorithm.
- It based on the foraging behavior of honey bee colonies.
- The artificial bee colony contains three groups:
    - Scouts
    - Onlookers
    - Employed bees

Algorithm

- The ABC generates a randomly distributed initial population of SN solutions (food source positions), where SN denotes the size of population.
- Each solution xi (i = 1, 2, ..., SN) is a D-dimensional vector.
- After initialization, the population of the positions (solutions) is subjected to repeated cycles, C = 1, 2, ...,MCN, of the search processes of the employed bees, the onlooker bees and scout bees.

Algorithm

- An **employed bee produces a modification on the position** (solution) in her memory depending on the **nectar amount (fitness value)** of the new source (new solution).

- Provided that the nectar amount of the new one is higher than that of the previous one, **the bee memorizes the new position and forgets the old one**.

- After all employed bees complete the search process, **they share the nectar information of the food sources and their position information with the onlooker bees on the dance area**.

Algorithm

- An onlooker bee **evaluates the nectar information taken from all employed bees and chooses a food source with a probability related to its nectar amount**.

- As in the case of the employed bee, it produces a modification on the position in its memory and checks the nectar amount of the candidate source.

- Providing that its nectar is higher than that of the previous one, the bee memorizes the new position and forgets the old one.

Algorithm

- An artificial onlooker bee chooses a food source depending on the probability value associated with that food source, p_i,

$$p_i = \frac{fit_i}{\sum_{n=1}^{SN} fit_n}$$

- fit_i is the fitness value of the solution i
- SN is the number of food sources which is equal to the number of employed bees (BN).

Algorithm

- In order to produce a candidate food position from the old one in memory, the ABC uses the following expression

$$v_{ij} = x_{ij} + \phi_{ij}(x_{ij} - x_{kj})$$

- where k ∈ {1, 2,..., SN} and j ∈ {1, 2,...,D} are randomly chosen indexes.
- k is determined randomly, it has to be different from i.
- $\phi_{i,j}$ is a random number between [-1, 1].

Algorithm

- The food source of which the nectar is abandoned by the bees is replaced with a new food source by the **scouts**.

- In ABC, providing that a position can not be improved further through a predetermined number of cycles, which is called “limit” then that food source is assumed to be abandoned.

$$x_i^j = x_{\min}^j + \mathrm{rand}(0,1)(x_{\max}^j - x_{\min}^j)$$

Algorithm

Algorithm 1 Artificial Bee Colony algorithm

Generate the initial population x_i randomly, $i = 1, \ldots, NS$. ▷ **Initialization**
Evaluate the fitness function fit_i of all solutions in the population.
Keep the best solution x_{best} in the population. ▷ **Memorize the best solution**
Set cycle=1.
repeat
    Generate new solutions v_i from old solutions x_i where $v_{ij} = x_{ij} + \phi_{ij}(x_{ij} - x_{kj})$, $\phi_{ij} \in [-1, 1]$, $k \in \{1, 2, \ldots, NS\}$, $j \in \{1, 2, \ldots, n\}$, and $i \neq k$. ▷ **Employed bees**
    Evaluate the fitness function fit_i of all new solutions in the population.
    Keep the best solution between current and candidate solutions. ▷ **Greedy selection**
    Calculate the probability P_i, for the solutions x_i where $P_i = fit_i / \sum_{j=1}^{NS} fit_j$.
    Generate the new solutions v_i from the solutions selecting depending on its P_i. ▷ **Onlookers bees**
    Evaluate the fitness function fit_i of all new solutions in the population.
    Keep the best solution between current and candidate solutions. ▷ **Greedy selection**
    Determine the abandoned solution if exist, replace it with a new randomly solution x_i. ▷ **Scout bee**
    Keep the best solutionx_{best} found so far in the population.
    $cycle = cycle + 1$
until $cycle \leq MCN$. ▷ **MCN is maximum cycle number**

Control Parameters

- Swarm size
- Employed bees (50% of swarm)
- Onlookers (50% of swarm)
- Scouts (1)
- Limit
- Dimension

Pros and Cons

- Advantages
  - Few control parameters
  - Fast convergence
  - Both exploration & exploitation

- Disadvantages
  - Search space limited by initial solution (normal distribution sample should use in initialize step)

Example

Consider the optimization problem as follows:

Minimize $f(x) = x^2_1 + x^2_2$ $\quad -5 \leq x1, x2 \leq 5$

Control Parameters of ABC Algorithm are set as:

Colony size, CS = 6

Limit for scout, L = (CS*D)/2 = 6

Dimension of the problem, D = 2

Example

First, we initialize the positions of 3 food sources (CS/2) of employed bees, randomly using uniform distribution in the range (-5, 5).

$$x = \begin{matrix} 1.4112 & -2.5644 \\ 0.4756 & 1.4338 \\ -0.1824 & -1.0323 \end{matrix}$$

f(x) values are:

8.5678
2.2820
1.0990

Example

Fitness function: $fit_i = \begin{cases} \dfrac{1}{1+f_i} & if\ f_i \geq 0 \\ 1 + abs\,(f_i) & if\ f_i < 0 \end{cases}$

Initial fitness vector is:
0.1045
0.3047
0.4764

Example

Maximum fitness value is 0.4764, the quality of the best food source.

Cycle=1

Employed bees phase

- 1st employed bee

Solutions before first cycle

$$x = \begin{matrix} 1.4112 & -2.5644 \\ 0.4756 & 1.4338 \\ -0.1824 & -1.0323 \end{matrix}$$

$$\upsilon_{i,j} = x_{i,j} + \Phi_{ij}(x_{i,j} - x_{k,j})$$

with this formula, produce a new solution.

k=1 k is a random selected index.

j=0 j is a random selected index.

Example

Φ = 0.8050   Φ is randomly produced number in the range [-1, 1].

υ0= 2.1644   -2.5644

Calculate f(υ0) and the fitness of υ0.
f(υ0) = 11.2610 and the fitness value is 0.0816.

Apply greedy selection between x0 and υ0

0.0816 < 0.1045, the solution 0 couldn’t be improved, increase its trial counter.

Example

2nd employed bee

$$\upsilon_{i,j} = x_{i,j} + \Phi_{ij}(x_{i,j} - x_{k,j})$$

with this formula produce a new solution.

Solutions before first cycle

x =	1.4112	-2.5644
	0.4756	1.4338
	-0.1824	-1.0323

k=2 k is a random selected solution in the neighborhood of i.
j=1 j is a random selected dimension of the problem.

Φ = 0.0762 Φ is randomly produced number in the range [-1, 1].

υ1= 0.4756 1.6217
Calculate f(υ1) and the fitness of υ1.
f(υ1) = 2.8560 and the fitness value is 0.2593.

Apply greedy selection between x1 and υ1
0.2593 < 0.3047, the solution 1 couldn't be improved, increase its trial counter.

Example

3rd employed bee

$$\upsilon_{i,j} = x_{i,j} + \Phi_{ij}(x_{i,j} - x_{k,j})$$

Solutions before first cycle

$$x = \begin{matrix} 1.4112 & -2.5644 \\ 0.4756 & 1.4338 \\ -0.1824 & -1.0323 \end{matrix}$$

with this formula produce a new solution.
k=0 //k is a random selected solution in the neighborhood of i.
j=0 //j is a random selected dimension of the problem.

Φ = -0.0671 // Φ is randomly produced number in the range [-1, 1].

υ2= **-0.0754 -1.0323**

Calculate f(υ2) and the fitness of υ2.
f(υ2) = 1.0714 and the fitness value is 0.4828.

Apply greedy selection between x2 and υ2.
0.4828 > 0.4764, the solution 2 was improved, set its trial counter as 0 and replace the solution x2 with υ2.

Example

Solutions after the first cycle

x =

1.4112	-2.5644
0.4756	1.4338
-0.0754	*-1.0323* ← *updated*

f(x) values are:

8.5678

2.2820

1.0714

fitness vector is:

0.1045

0.3047

0.4828

Example

Calculate the probability values p for the solutions x by means of their fitness values by using the formula;

$$p_i = \frac{fit_i}{\sum_{i=1}^{CS/2} fit_i}.$$

p = 0.1172

0.3416

0.5412

Example

Onlooker bees phase
Produce new solutions υ_i for the onlookers from the solutions x_i selected depending on p_i and evaluate them.

1st onlooker bee
$i=3$
$\upsilon_3 = -0.0754 \quad -2.2520$

Calculate $f(\upsilon_3)$ and the fitness of υ_3.
$f(\upsilon_3) = 5.0772$ and the fitness value is 0.1645.
Apply greedy selection between x_3 and υ_3

$0.1645 < 0.4828$, the solution 3 couldn't be improved, increase its trial counter.

Example

2nd onlooker bee
i=2

$\upsilon 2$= 0.1722   1.4338

Calculate $f(\upsilon 2)$ and the fitness of $\upsilon 2$.
$f(\upsilon 2) = 2.0855$ and the fitness value is 0.3241.

Apply greedy selection between $x2$ and $\upsilon 2$
$0.3241 > 0.3047$, the solution 2 was improved, set its trial counter as 0 and replace the solution $x2$ with $\upsilon 2$.

Example

x =

1.4112	-2.5644
0.1722	1.4338
-0.0754	-1.0323

f(x) values are

8.5678
2.0855
1.0714

fitness vector is:

0.1045
0.3241
0.4828

Example

3rd onlooker bee
 $i=3$
$\upsilon3 = 0.0348 \quad -1.0323$

Calculate $f(\upsilon3)$ and the fitness of $\upsilon3$.
$f(\upsilon3) = 1.0669$ and the fitness value is 0.4838.
Apply greedy selection between $x3$ and $\upsilon3$

$0.4838 > 0.4828$, the solution 3 was improved, set its trial counter as 0 and replace the solution $x3$ with $\upsilon3$.

Example

x =

1.4112	-2.5644
0.1722	1.4338
0.0348	-1.0323

f(x) values are

8.5678
2.0855
1.0669

fitness vector is:

0.1045
0.3241
0.4838

Example

Memorize best
Best = 0.0348   -1.0323
Scout bee phase
Trial Counter =
1
0
0

There is no abandoned solution since L = 6
If there is an abandoned solution (the solution of which the trial counter value is higher than L = 6);

Generate a new solution randomly to replace with the abandoned one.
Cycle = Cycle+1
The procedure is continued until the termination criterion is attained.

Resources

https://abc.erciyes.edu.tr/

Cuckoo Search Algorithm

Adopted from Ahmed Fouad Ali

Introduction

- A method of global optimization based on the behavior of cuckoos was proposed by Yang & Deb (2009).

- The original "cuckoo search (CS) algorithm" is based on the idea of the following:

  - How cuckoos lay their eggs in the host nests.

  - How, if not detected and destroyed, the eggs are hatched to chicks by the hosts.

  - How a search algorithm based on such a scheme can be used to find the global optimum of a function.

Behaviour

- The CS was inspired by the obligate brood parasitism of some cuckoo species by laying their eggs in the nests of host birds.

- Some cuckoos have evolved in such a way that female parasitic cuckoos can imitate the colors and patterns of the eggs of a few chosen host species.

- This reduces the probability of the eggs being abandoned and, therefore, increases their reproductivity .

Behaviour

- If host birds discover the eggs are not their own, they will either throw them away or simply abandon their nests and build new ones.

- Parasitic cuckoos often choose a nest where the host bird just laid its own eggs.

- In general, the cuckoo eggs hatch slightly earlier than their host eggs.

Behaviour

- Once the first cuckoo chick is hatched, his first instinct action is to evict the host eggs by blindly propelling the eggs out of the nest.

- This action results in increasing the cuckoo chick's share of food provided by its host bird.

- Moreover, studies show that a cuckoo chick can imitate the call of host chicks to gain access to more feeding opportunity.

Characteristics

- Each egg in a nest represents a solution, and a cuckoo egg represents a new solution.
- The aim is to employ the new and potentially better solutions (cuckoos) to replace not-so-good solutions in the nests.
- In the simplest form, each nest has one egg.
- The algorithm can be extended to more complicated cases in which each nest has multiple eggs representing a set of solutions

Characteristics

- The CS is based on three idealized rules:
  - Each cuckoo lays one egg at a time, and dumps it in a randomly chosen nest
  - The best nests with high quality of eggs (solutions) will carry over to the next generations
  - The number of available host nests is fixed, and a host can discover an alien egg with probability p ϵ [0,1] .
- In this case, the host bird can either throw the egg away or abandon the nest to build a completely new nest in a new location.

Lèvy Flights

- In nature, animals search for food in a random or quasi-random manner.
- Generally, the foraging path of an animal is effectively a random walk because the next move is based on both the current location/state and the transition probability to the next location.
- The chosen direction implicitly depends on a probability, which can be modelled mathematically.

Lèvy Flights

- A Lévy flight is a random walk in which the step-lengths are distributed according to a heavy-tailed probability distribution.
- After a large number of steps, the distance from the origin of the random walk tends to a stable distribution.

Algorithm

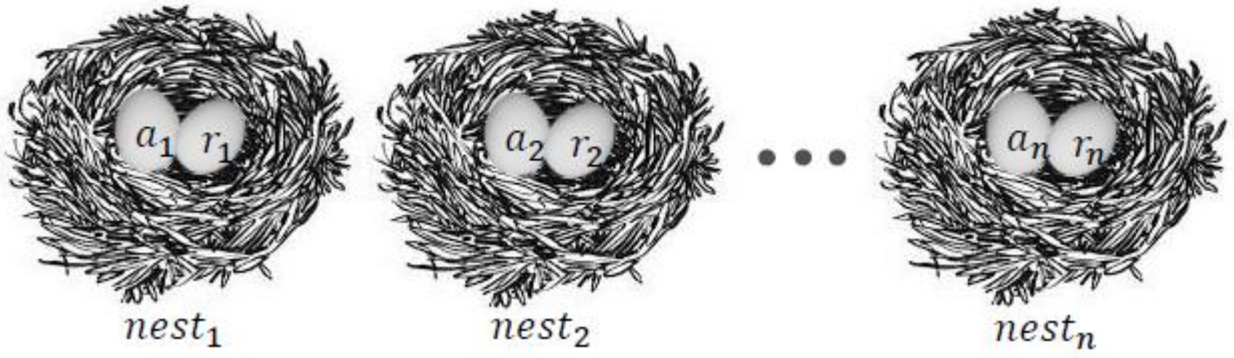
Algorithm 1 Cuckoo search algorithm

Set the initial value of the host nest size n, probability $p_a \in [0, 1]$ and maximum number of iterations Max_{itr}.
Set $t := 0$. {Counter initialization}.
for $(i = 1 : i \leq n)$ **do**
    Generate initial population of n host $x_i^{(t)}$. {n is the population size}.
    Evaluate the fitness function $f(x_i^{(t)})$.
end for
repeat
    Generate a new solution (Cuckoo) $x_i^{(t+1)}$ randomly by Lévy flight.
    Evaluate the fitness function of a solution $x_i^{(t+1)}$ $f(x_i^{(t+1)})$
    Choose a nest x_j among n solutions randomly.
    if $(f(x_i^{(t+1)}) > f(x_j^{(t)}))$ **then**
        Replace the solution x_j with the solution $x_i^{(t+1)}$
    end if
    Abandon a fraction p_a of worse nests.
    Build new nests at new locations using Lévy flight a fraction p_a of worse nests
    Keep the best solutions (nests with quality solutions)
    Rank the solutions and find the current best solution
    Set $t = t + 1$. {Iteration counter increasing}.
until $(t < Max_{itr})$. {Termination criteria satisfied}.
Produce the best solution.

Steps

The following steps describe the main concepts of Cuckoo search algorithm

Step1. Generate initial population of n host nests.



(ai, ri) : a candidate for optimal parameters

Steps

Step2. Lay the egg (ak',bk') in the k nest.

K nest is randomly selected.
Cuckoo’s egg is very similar to host egg.

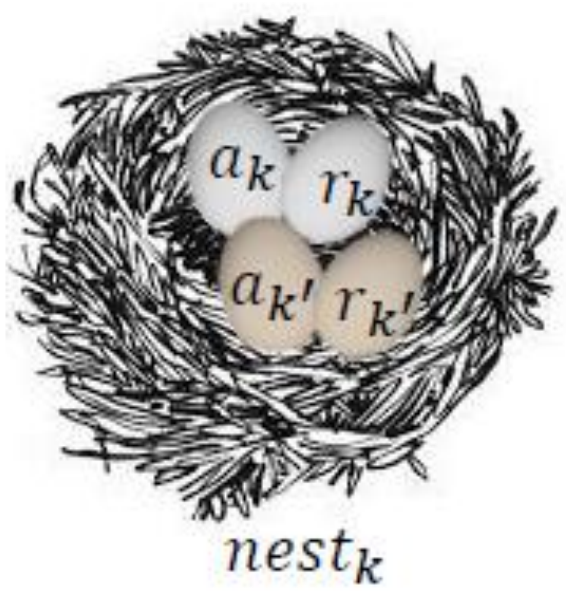
Where

ak'=ak+$Randomwalk$ (Lèvy flight) ak
rk'=rk+$Randomwalk$ (Lèvy flight) rk

Steps

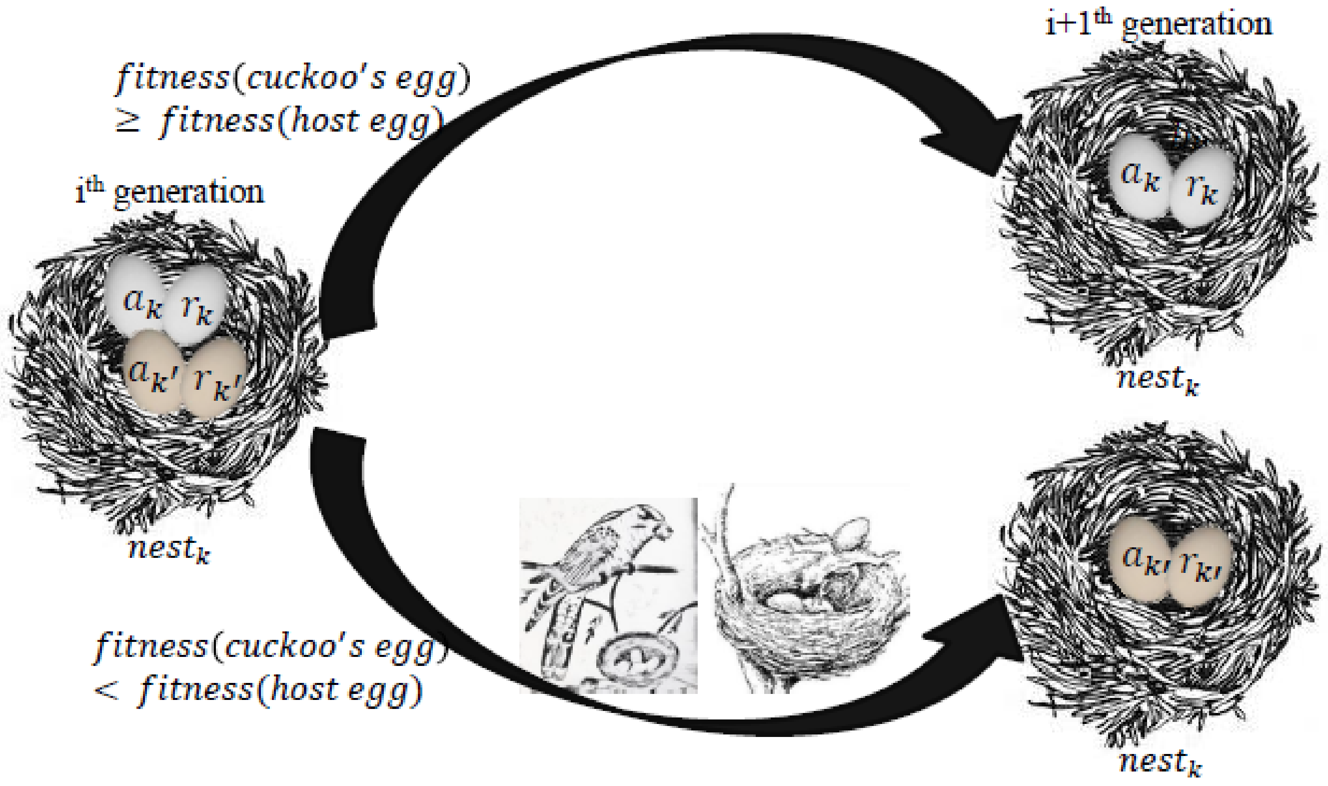
Step3. Compare the fitness of cuckoo's egg with the fitness of the host egg.

- Root Mean Square Error (RMSE)



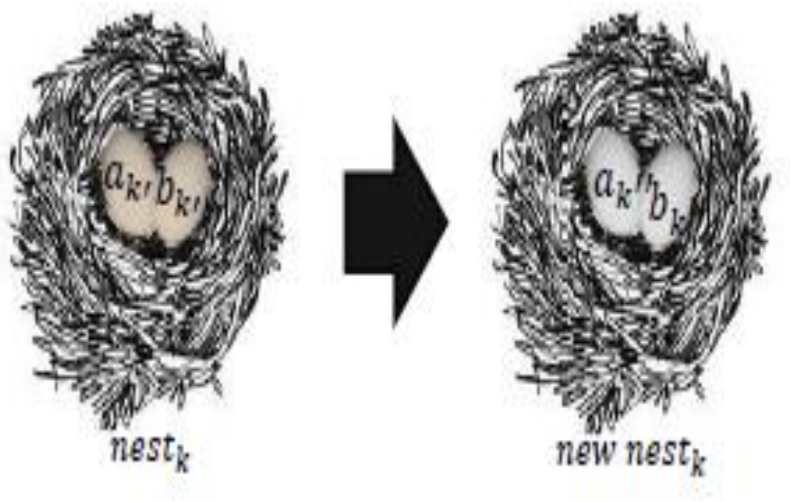
Steps

Step4. If the fitness of cuckoo's egg is better than host egg, replace the egg in nest k by cuckoo's egg.



Steps

Step5. If host bird notice it, the nest is abandoned and new one is built (p <0.25) (to avoid local optimization)



Iterate steps 2 to 5 until termination criterion satisfied

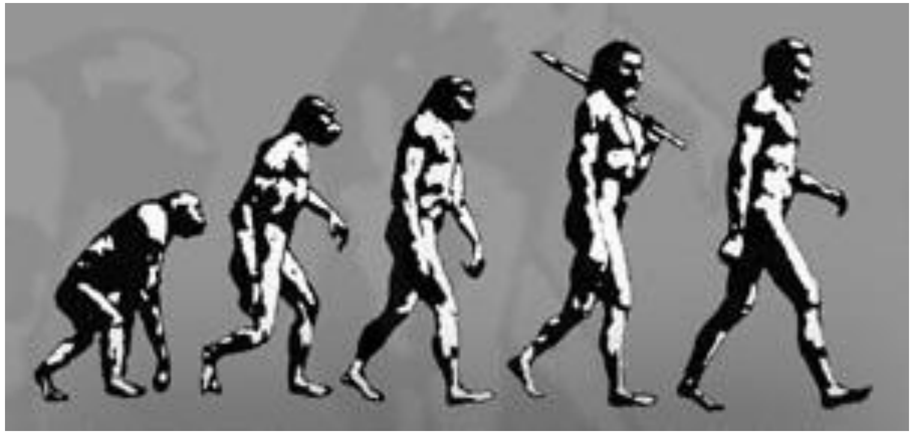
Applications

- Engineering optimization problems
- NP hard combinatorial optimization problems
- Data fusion in wireless sensor networks
- Nanoelectronic technology based operation-amplifier (OP-AMP)
- Train neural network
- Manufacturing scheduling
- Nurse scheduling problem

Evolutionary Computation

Introduction

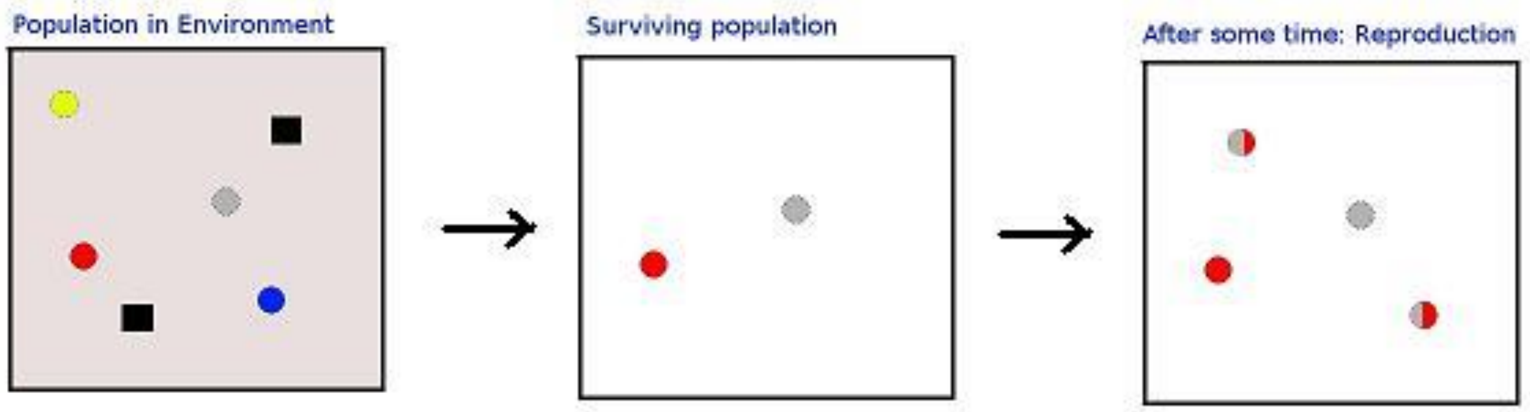
- Evolution is the change in the inherited traits of a population from one generation to the next.



- Natural selection leading to better and better species

Introduction

- Survival of the fittest.
- Change in species is due to change in genes over reproduction or/and due to mutation.



- An Example showing the concept of survival of the fittest and reproduction over generations.

Introduction

- Mimicking natural evolution to evolve better « solutions »
- Generation of successive populations with survival and reproduction of the fittests
- Using mutation and cross-over as reproduction operators
- Genotype vs. Phenotype
- A kind of generalized optimization method
- A population of “solutions” : size
- Reproduction operators
- Selection of the fittests

History

- "Evolutionary computing"
  - I. Rechenberg in the 60s.
  - *Optimization on real valued domains*
- Genetic algorithms
  - John Holland, "*Adaptation in Natural and Artificial Systems*", 1975.
  - *Bit representation / Schema theorem / Problem-Solving method*
- Genetic Programming
  - John Koza, First book on Genetic Programming, 1992.
  - *Programs represented as trees*

Evolutionary Computation

- **Evolutionary Computation (EC)** refers to computer-based problem solving systems that use computational models of evolutionary process.
- Terminology:
  - **Chromosome** – It is an individual representing a candidate solution of the optimization problem.
  - **Population** – A set of chromosomes.
  - **Gene** – It is the fundamental building block of the chromosome, each gene in a chromosome represents each variable to be optimized. It is the smallest unit of information.
  - **Objective**: To find a best possible chromosome to a given optimization problem.

Evolutionary Algorithm

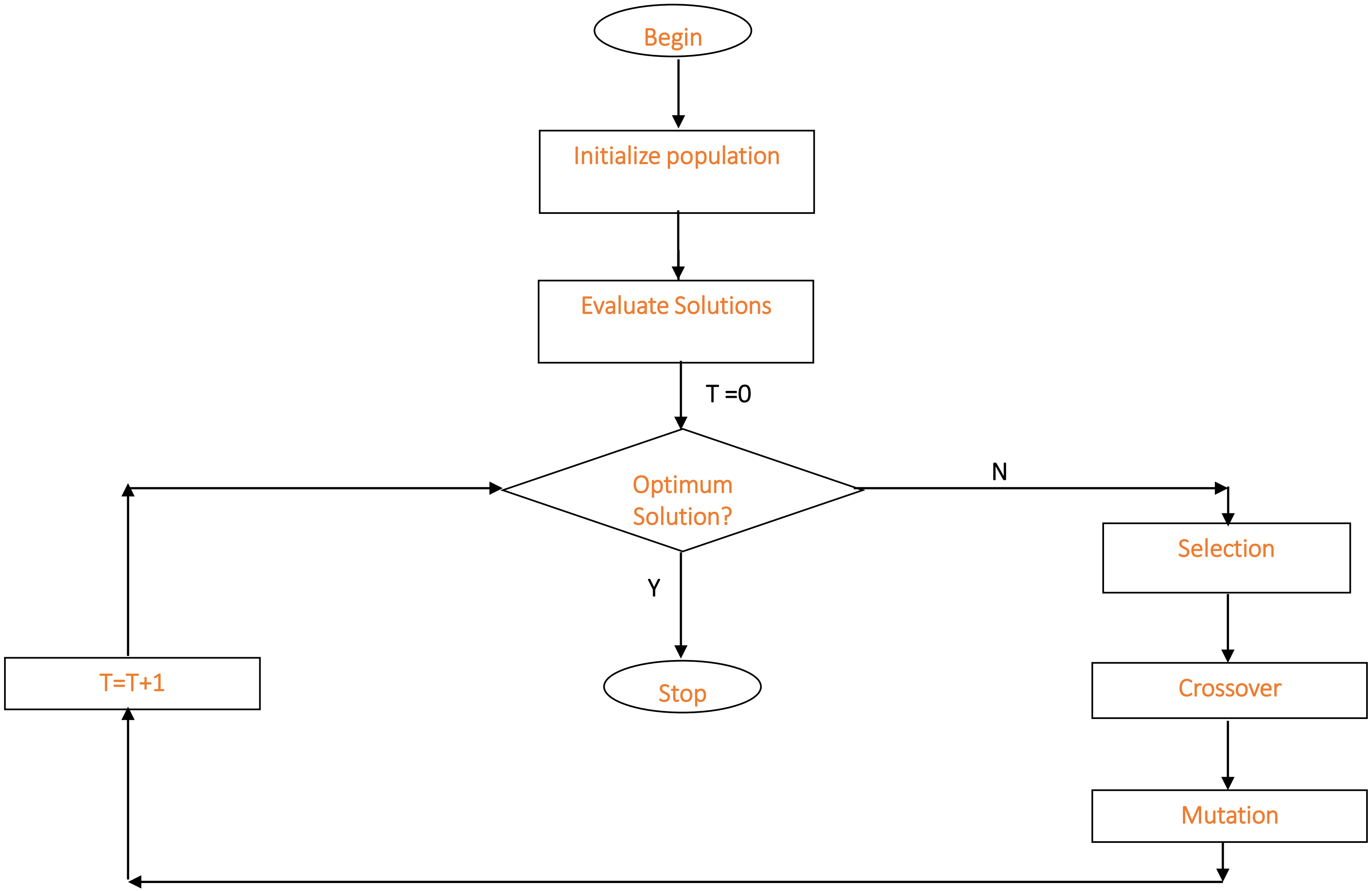
Let t = 0 be the generation counter;
create and initialize a population P(0);

repeat

    Evaluate the fitness, f(xi), for all xi belonging to P(t);
    Perform cross-over to produce offspring;
    Perform mutation on offspring;
    Select population P(t+1) of new generation;
    Advance to the new generation, i.e. t = t+1;

until stopping condition is true;

Evolutionary Algorithm



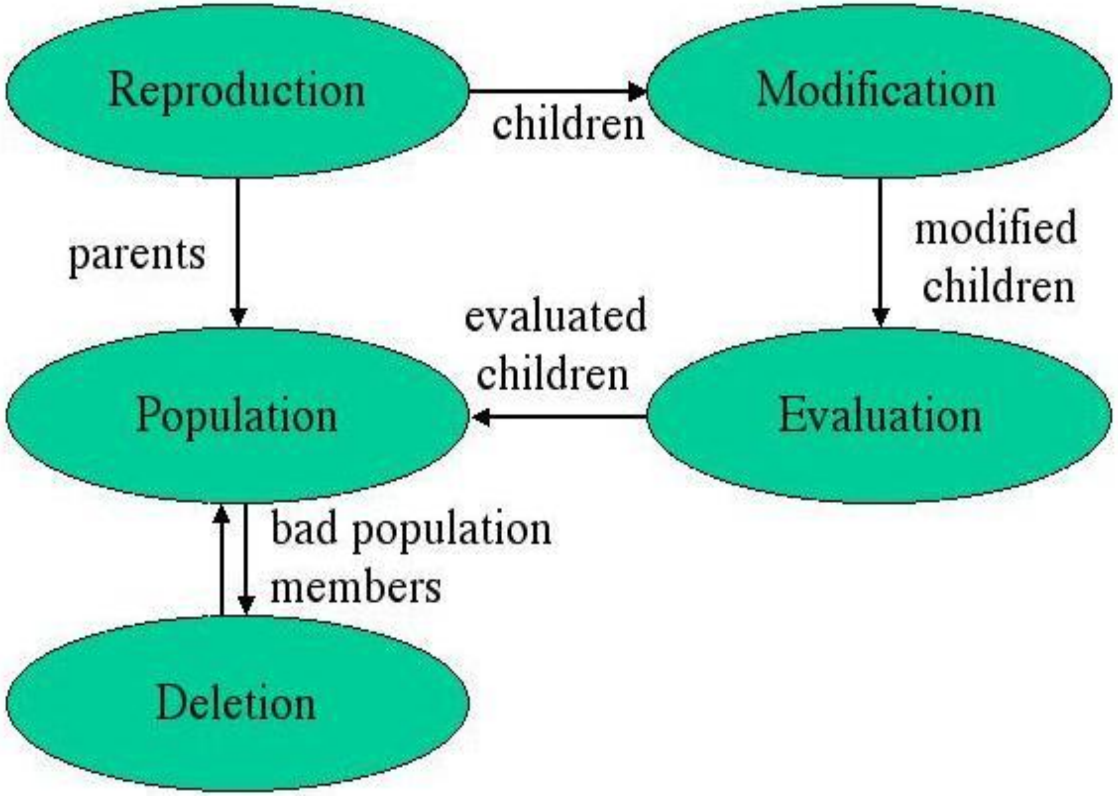
Genetic Algorithms

- GA emulate genetic evolution.
- A GA has distinct features:
  - A string representation of chromosomes.
  - A selection procedure for initial population and for off-spring creation.
  - A cross-over method and a mutation method.
  - A fitness function be to minimized.
  - A replacement procedure.
  - Parameters that affect GA are initial population, size of the population, selection process and fitness function.

Genetic Algorithms

Natural Evolution	*Evolutionary Computation*
Population	Pool of solutions
Individual	Solution to a problem
Fitness of an individual	Quality of a solution
Chromosome	Encoding of a solution
Gene	Part of the encoding
Reproduction	Mutation and/or crossover

Anatomy


Reproduction
children
Modification
parents
modified
children
evaluated
children
Population
Evaluation
bad population
members
Deletion

Representation

Various encoding schemes

- Bit strings
- Strings of values
- Real value
- tree

Chromosome 1	1 1 0 1 0 1 1 0 0 0 1
Chromosome 2	1 0 0 1 0 1 1 1 0 0 0
…	…

Chromosome 1	1 5 3 6 0 1 2 7 3 0 8
Chromosome 2	9 2 4 1 8 3 2 6 2 1 0
…	…

Initialization

- *N* individuals generally randomly generated
- *N* is domain-dependent
  - Often in [~50 - ~1000]

Fitness Function

- Evaluates the quality of the solution
  - E.g. *z-value* in function optimization
  - *Length of the circuit* in the travelling salesman problem
  - *Time before falling down* in the inverse pole

- Beware of its cost
  - Keep values in memory

Selection

- Selection is a procedure of picking parent chromosome to produce off-spring.
- Types of selection:
  - **Random Selection** – Parents are selected randomly from the population.
  - **Proportional Selection** – probabilities for picking each chromosome is calculated as:

$$P(\mathbf{x}_i) = f(\mathbf{x}_i)/\Sigma f(\mathbf{x}_j) \qquad \text{for all } j$$

  - **Rank Based Selection** – This method uses ranks instead of absolute fitness values.

$$P(\mathbf{x}_i) = (1/\beta)(1 - e^{r(\mathbf{x}_i)})$$

Wheel Selection

- Let $i = 1$, where i denotes chromosome index;
- Calculate $P(\mathbf{x}_i)$ using proportional selection;
- $sum = P(\mathbf{x}_i)$;
- choose $r \sim U(0,1)$;

  while $sum < r$ **do**

  $i = i + 1$; i.e. next chromosome

  $sum = sum + P(\mathbf{x}_i)$;

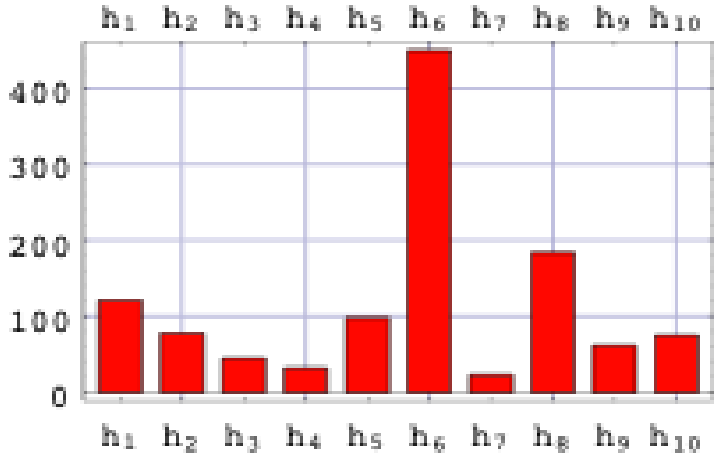
  end

  return $\mathbf{x}_i$ as one of the selected parent;

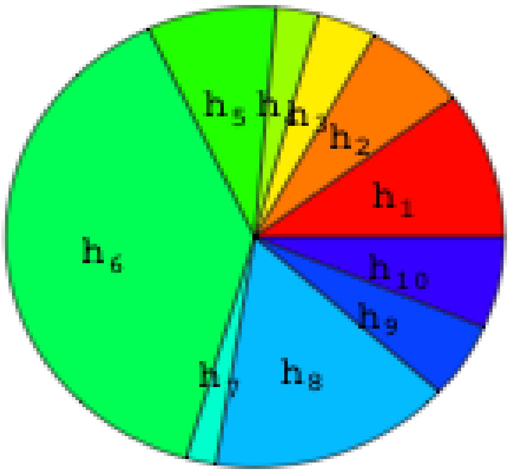
  repeat until all parents are selected

Wheel Selection

The **probability** of selecting an individual is **proportional to its fitness**



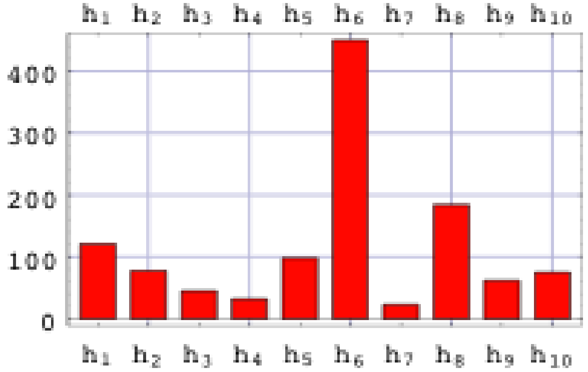
Fitness



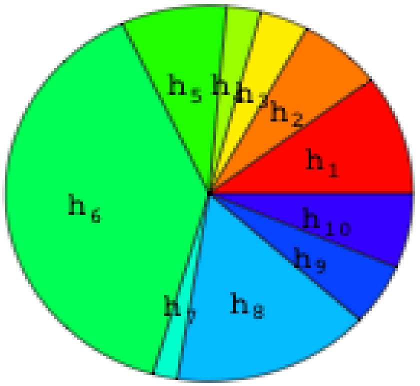
Probability of selection

Wheel Selection

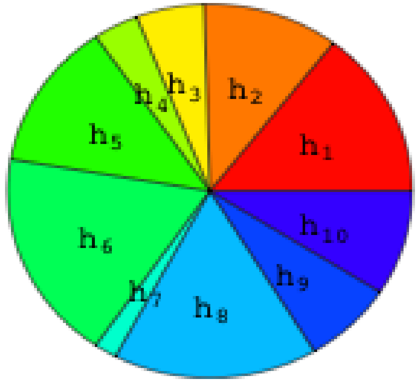
The **probability** of selecting an individual is **proportional to its** rank



Fitness



Probability of selection according to **fitness**



Probability of selection according to **rank**

Tournament

- **Selection by fitness or rank** implies the evaluation of the fitness of all individuals
- **Selection by tournament** avoids this
  - If n individuals must be selected (within a population of size N)
  - Organize n tournaments, each between $m < N$ randomly chosen individuals (m controls the selective pressure)
  - Select the best individual / or select the best and second best / or …

Reproduction

- Reproduction is a processes of creating new chromosomes out of chromosomes in the population.
- Parents are put back into population after reproduction.
- **Cross-over and Mutation** are two parts in reproduction of an off-spring.

- Cross-over : It is a process of creating one or more new individuals through the combination of genetic material randomly selected from two or more parents.

Crossover

- Uniform cross-over : where corresponding bit positions are randomly exchanged between two parents.
- One point : random bit is selected and entire sub-string after the bit is swapped.
- Two point : two bits are selected and the sub-string between the bits is swapped.

	Uniform Cross-over	One point Cross-over	Two point Cross-over
Parent1 Parent2	00110110 11011011	00110110 11011011	00110110 11011011
Off-spring1 Off-spring2	01110111 10011010	00111011 11010110	01011010 10110111

Mutation

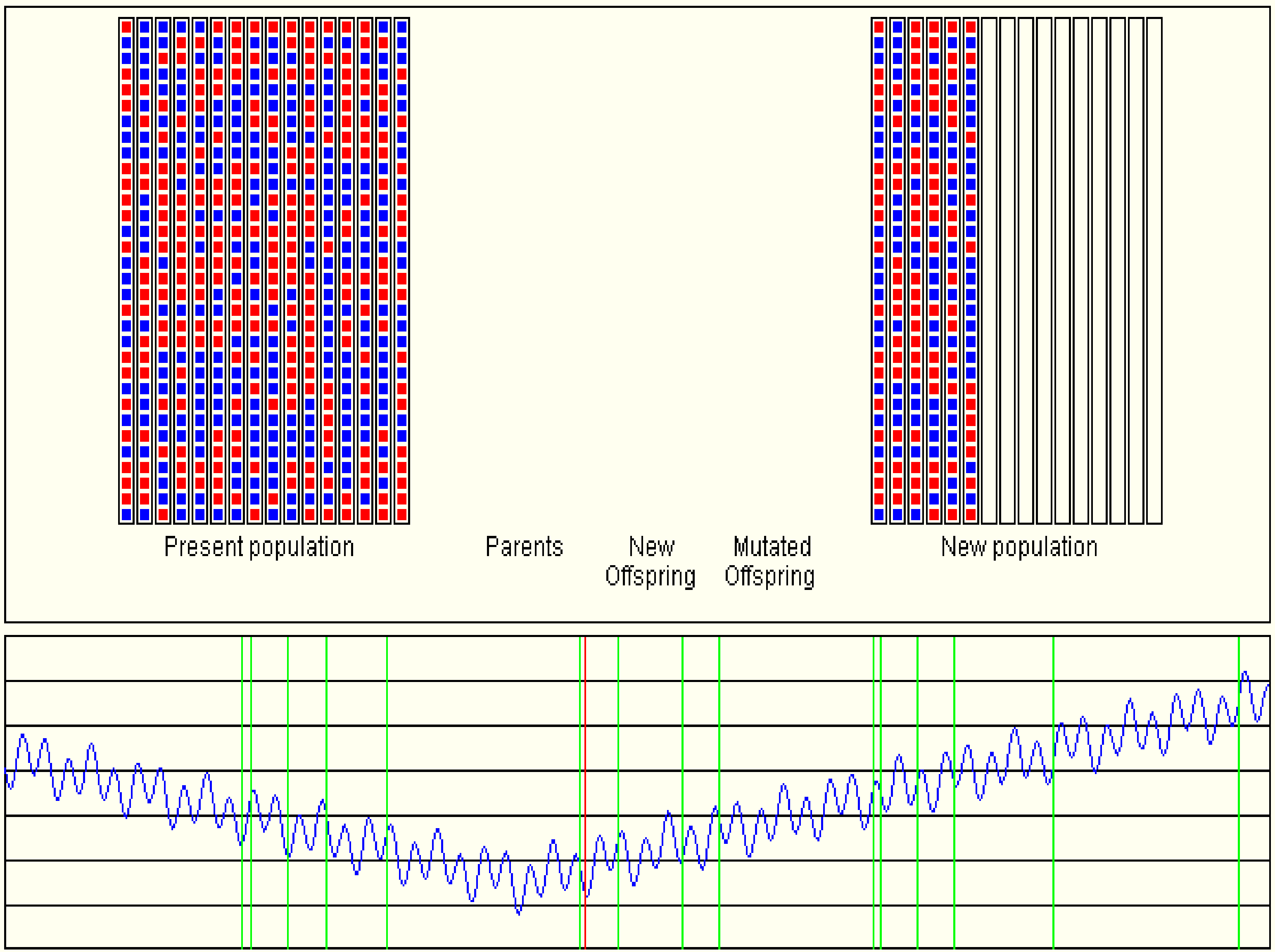
- Mutation procedures depend upon the representation schema of the chromosomes.
- This is to prevent falling all solutions in population into a local optimum.
- For a bit-vector representation:
  - random mutation : randomly negates bits
  - in-order mutation : performs random mutation between two randomly selected bit position.

	Random Mutation	In-order Mutation
Before mutation	1110010011	1110010011
After mutation	1100010111	1110011010

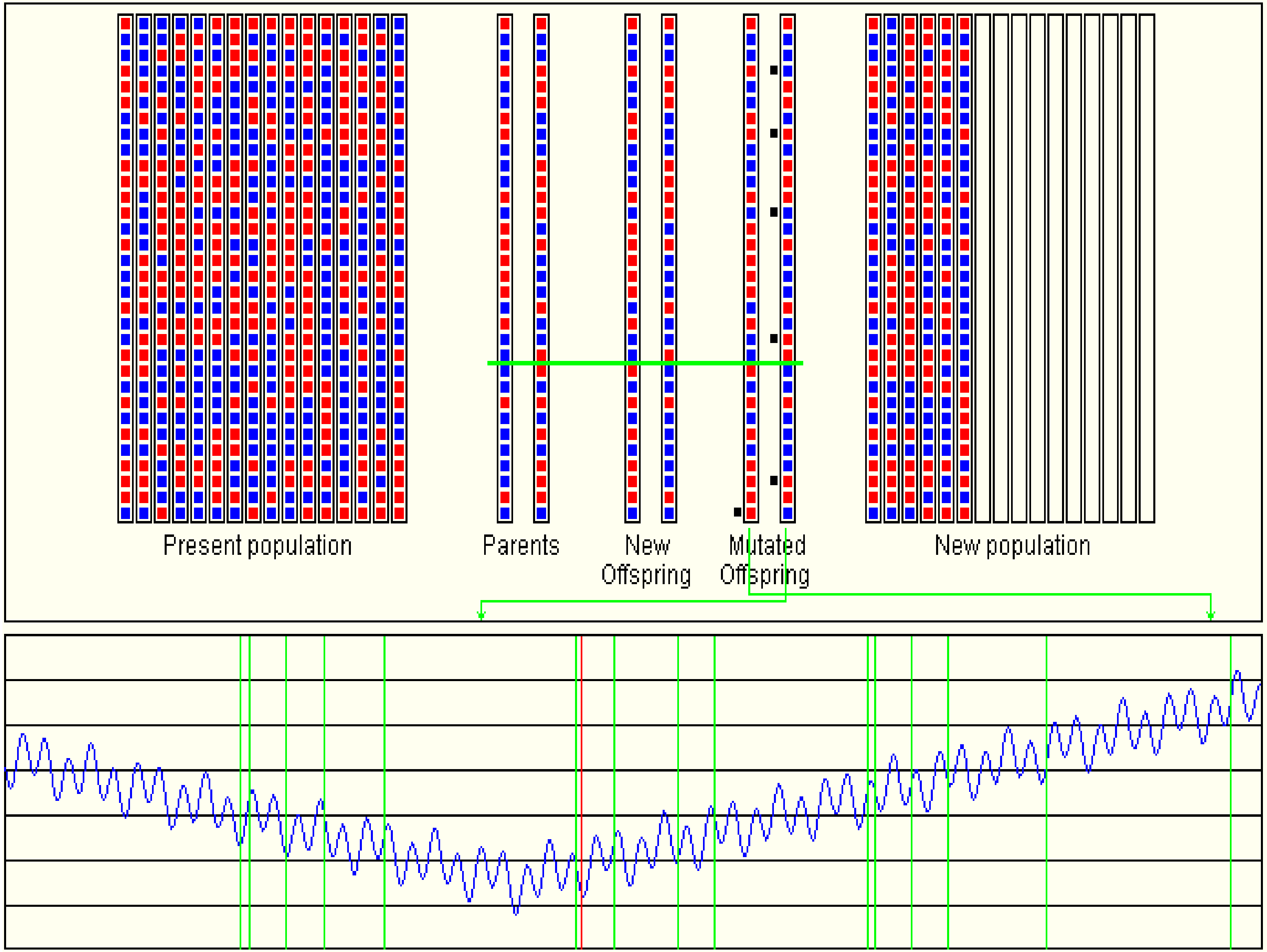
Operators

- Assure trade-off between
  - *Exploitation*
    - Preserve best individuals and explore nearby locations
    - **Mutation** is exploitation oriented
    - Small steps but brings new alleles
  - *Exploration*
    - Search unexplored regions for possible good candidates
    - **Crossover** is exploration oriented
    - Large steps but does not bring new alleles

Introduction

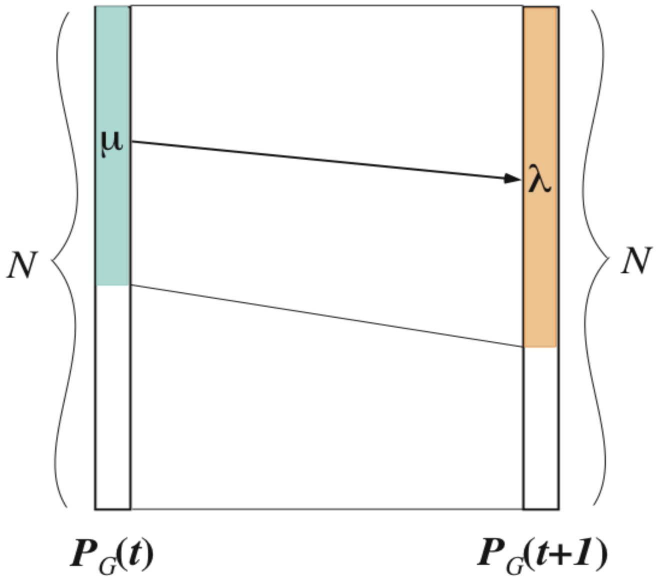


Introduction



Replacements

- Selection of *m* parents
  - By fitness / rank / tournament / …
- Generation of *l* children
  - Mutation / crossover / copy
  - And selection of the best
- Completion to *N*
  - Elimination of the worst individuals and copy of others

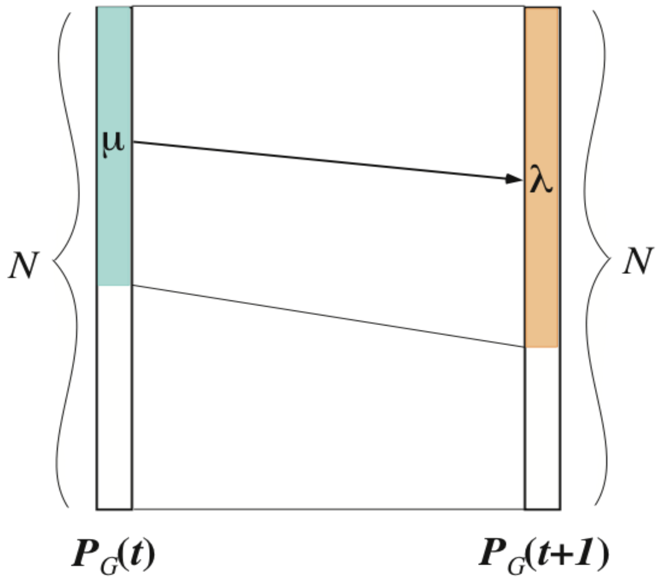


Strategies

1. Completely replace the previous population (called **(m,l)** ***replacement***)
   - Risk: loosing the good individuals of previous population

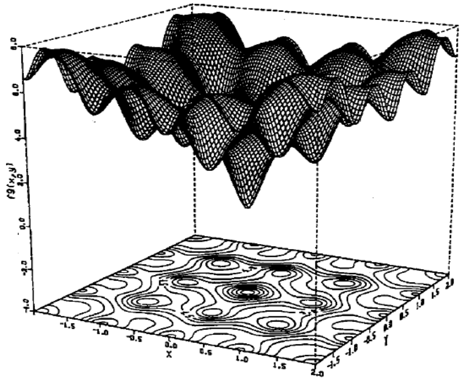
2. Draw the *N* new individuals from the selected m parents and l children (called (m + l) *replacement*)

3. *Steady state*
   - Select a sub-population and make replacement for this sub-population only (possibility of parallel and asynchronous process



Example

- Problem: finding Argmax of x^2 over {0,…,31}
- GA approach
  - **Representation**: binary code (e.g. 0 1 1 0 1 <-> 13)
  - Population **size** = 4
  - **Operators**
    - Single-point crossover
    - Mutation
  - Roulette wheel **selection** according to fitness
  - **Random initialization** of the population



A more complex optimization problem

Example

Selection

String no.	Initial population	x Value	Fitness $f(x) = x^2$	$Prob_i$	Expected count	Actual count
1	0 1 1 0 1	13	169	0.14	0.58	1
2	1 1 0 0 0	24	576	0.49	1.97	2
3	0 1 0 0 0	8	64	0.06	0.22	0
4	1 0 0 1 1	19	361	0.31	1.23	1
Sum			1170	1.00	4.00	4
Average			293	0.25	1.00	1
Max			576	0.49	1.97	2

Example

Crossover

String no.	Mating pool	Crossover point	Offspring after xover	x Value	Fitness $f(x) = x^2$
1	0 1 1 0 | 1	4	0 1 1 0 0	12	144
2	1 1 0 0 | 0	4	1 1 0 0 1	25	625
2	1 1 | 0 0 0	2	1 1 0 1 1	27	729
4	1 0 | 0 1 1	2	1 0 0 0 0	16	256
Sum					1754
Average					439
Max					729

Example

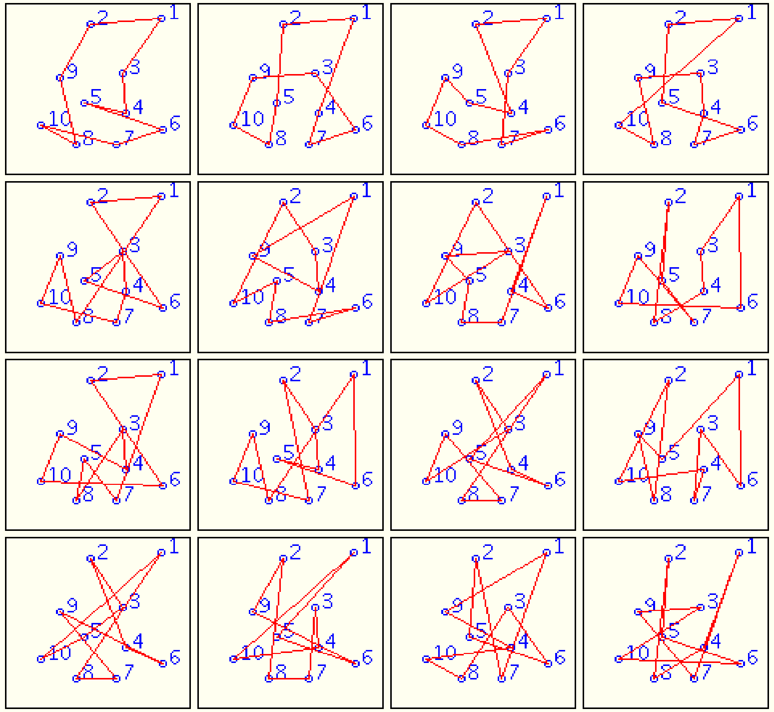
Mutation

String no.	Offspring after xover	Offspring after mutation	x Value	Fitness $f(x) = x^2$
1	0 1 1 0 0	$\boxed{1}$ 1 1 0 0	26	676
2	1 1 0 0 1	1 1 0 0 1	25	625
2	1 1 0 1 1	1 1 0 1 1	27	729
4	1 0 0 0 0	1 0 $\boxed{1}$ 0 0	18	324
Sum				2354
Average				588.5
Max				729

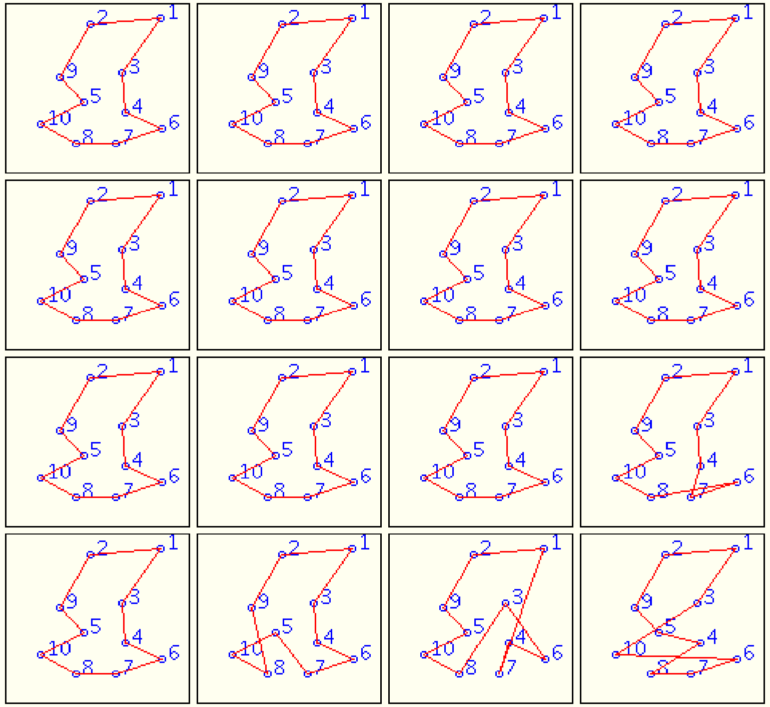
TSP

- The traveling salesman problem is difficult to solve by traditional genetic algorithms because of the requirement that each node **must be visited *exactly*** once.
- One way to solve this problem is by introducing more operators. Example in simulated annealing.
- The idea is to change the encoding pattern of chromosomes such that GA meta-heuristic can still be applicable.
- Transfer the TSP from a permutation problem into a priority assignment problem.

TSP



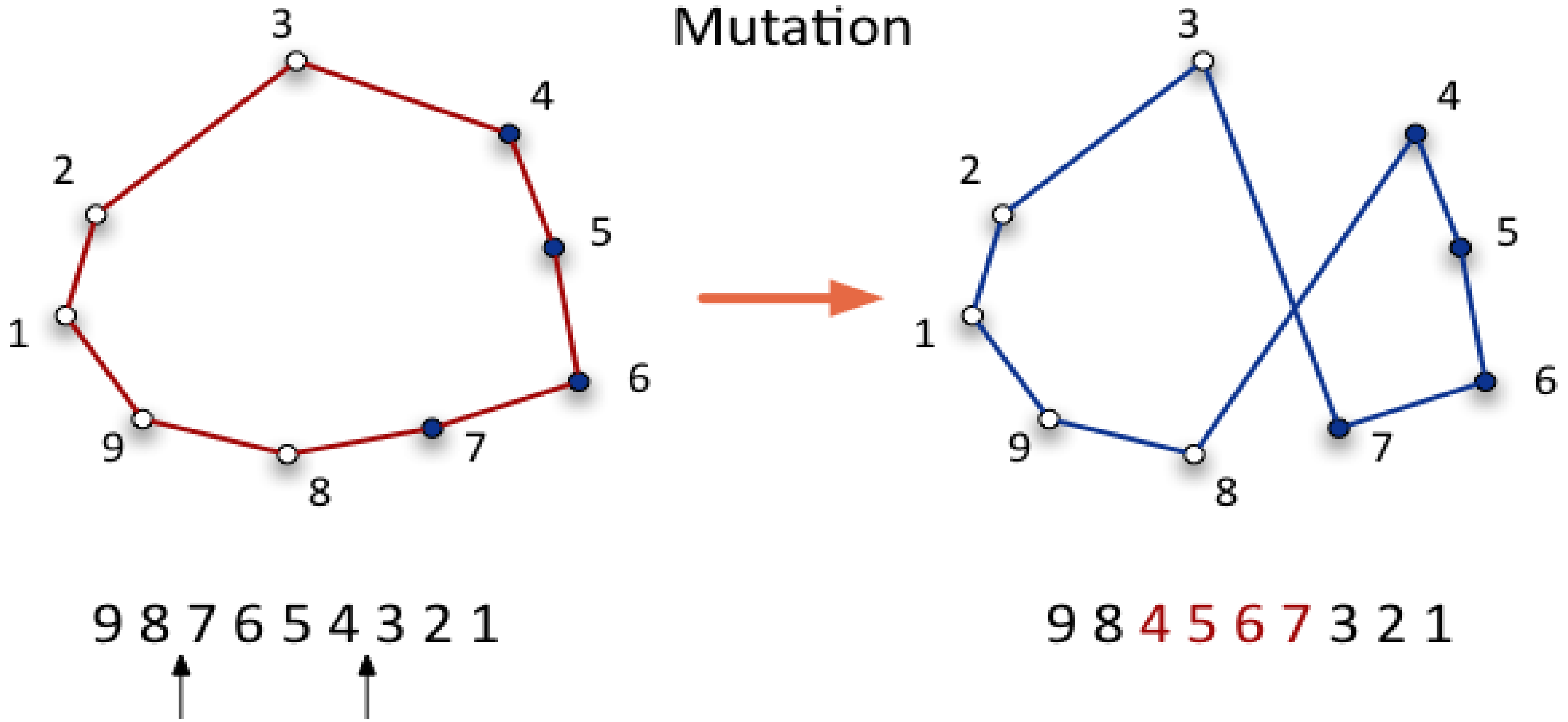
1000 steps



Population = 16

TSP

A solution: the **"2-opt mutation"**



Optimizing Sorting

- Normal sorting algorithms do not take into account the characteristics of the architecture and the nature of the input data

- Different sorting techniques are best suited for different types of input

Optimizing Sorting

- For example radix sort is the best algorithm to use when the standard deviation of the input is high as there will be less cache misses (Merge Sort better in other cases etc)

- The objective is to create a composite sorting algorithm

- The composite sorting algorithm evolves from the use of a Genetic Algorithm (GA)

Optimizing Sorting

- Sorting Primitives – these are the building blocks of our composite sorting algorithm

- Partitioning
  - Divide by Value (DV) (Quicksort)
  - Divide by Position (DP) (Merge Sort)
  - Divide by Radix (DR) (Radix Sort)

Optimizing Sorting

- Branch by Size (BS) : this primitive is used to select different sorting paths based on the size of the partition

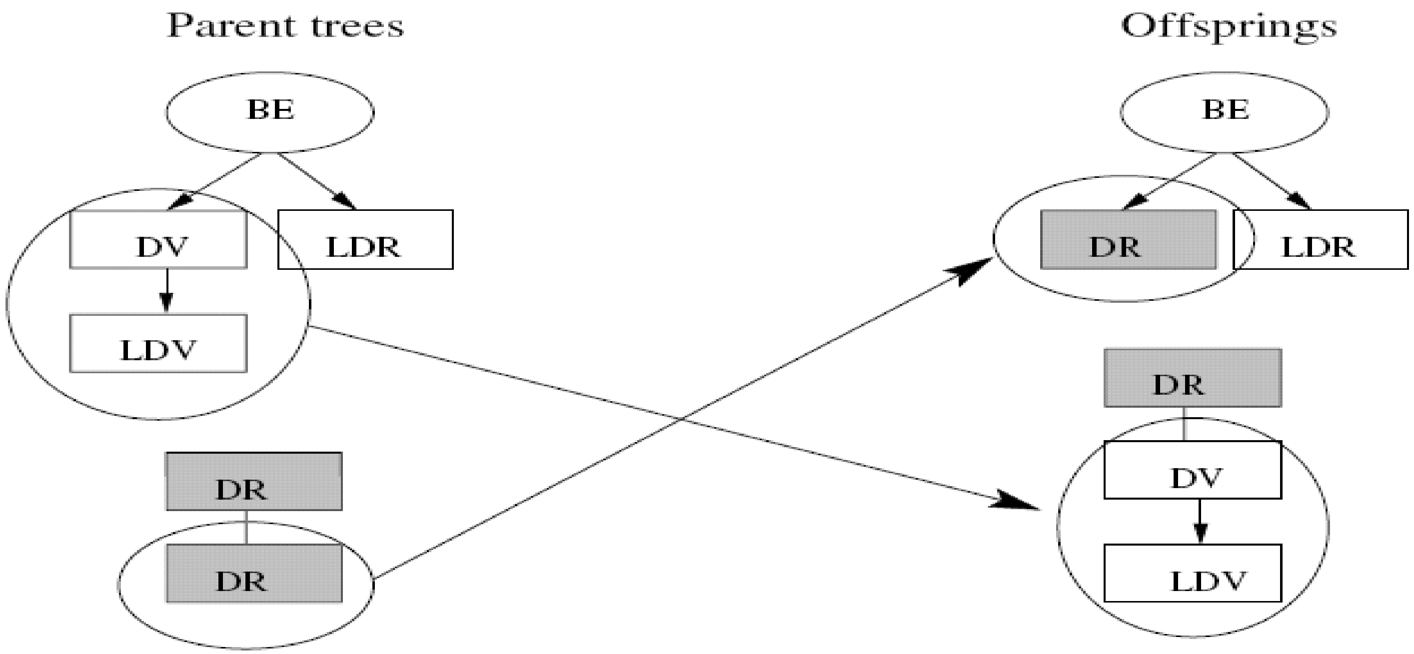
- Branch by Entropy (BE): this primitive is used to select different paths based on the entropy of the input

Optimizing Sorting

- The efficiency of radix sort increases with standard deviation of the input
- A measure of this is calculated as follows.
- We scan the input set and compute the number of keys that have a particular value for each digit position.
- For each digit the entropy is calculated as $\Sigma_i -P_i * \log P_i$ where $P_i = c_i/N$ where c_i = number of keys with value 'i' in that digit and N is the total number of keys

Optimizing Sorting

New offspring are generated using random single point crossovers



Optimizing Sorting

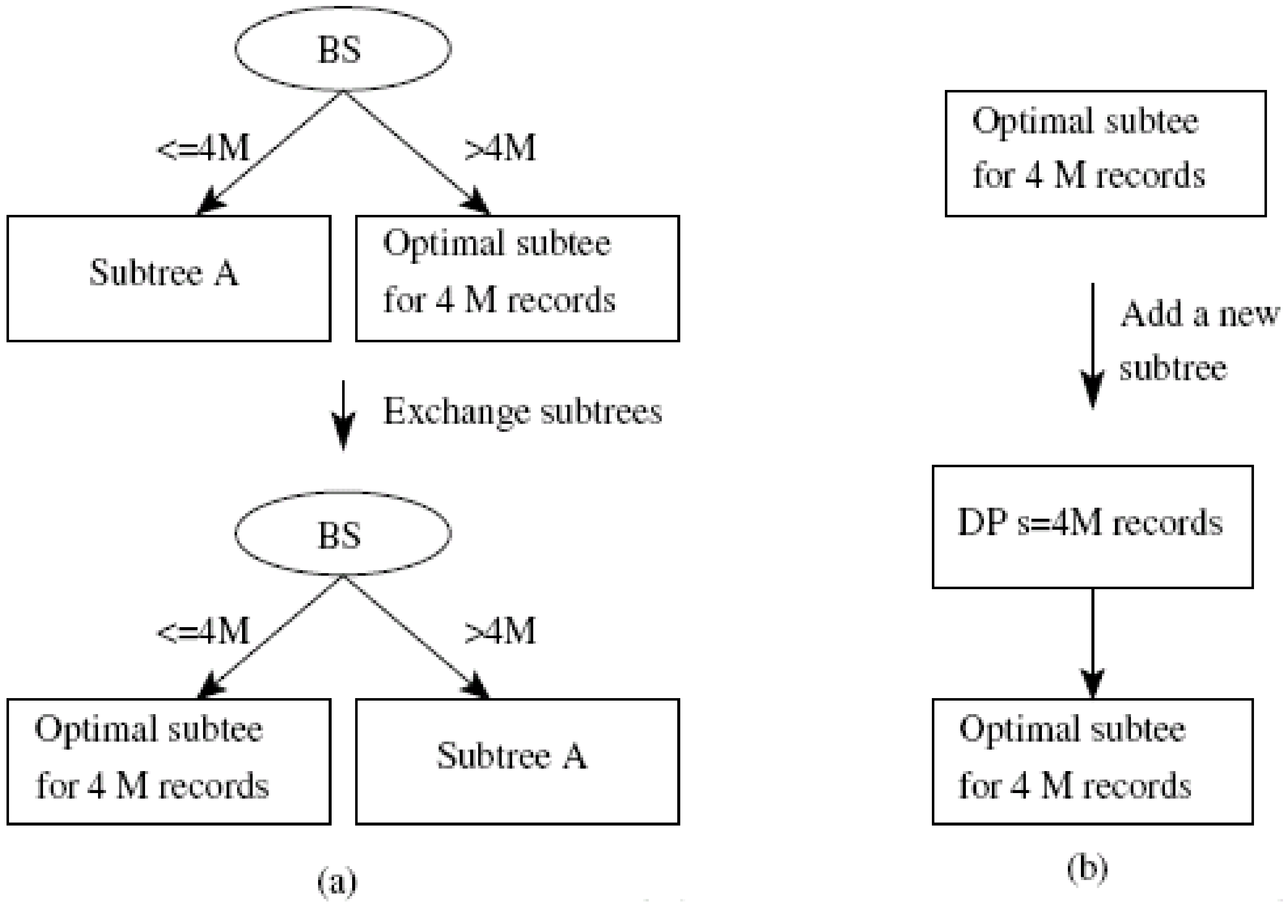
1. Change the values of the parameters of the sorting and selection primitives

2. Exchange two subtrees

3. Add a new subtree. This kind of mutation is useful where more partitioning is needed along a path of the tree

4. Remove a subtree

Optimizing Sorting

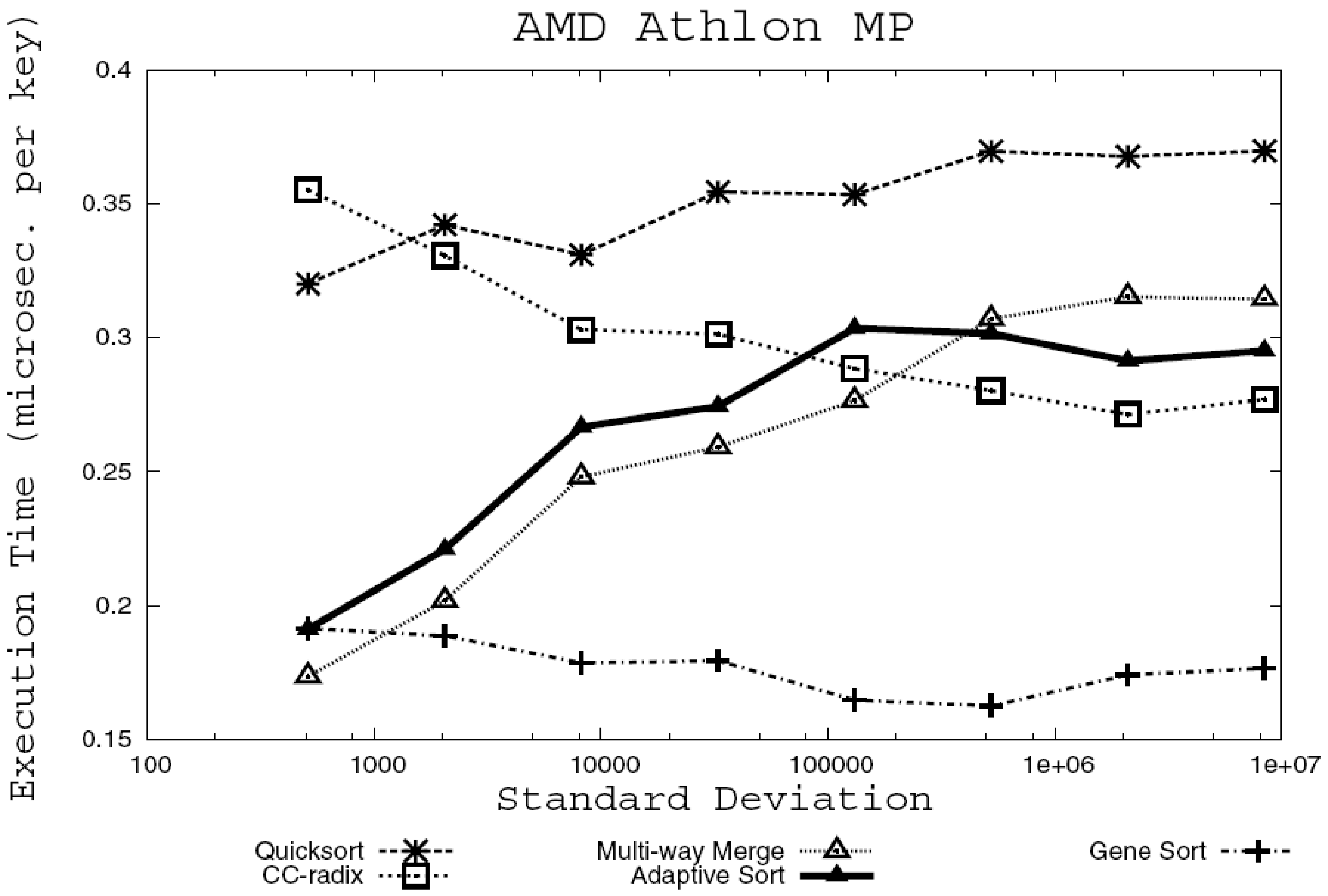


Fitness Function

- We are searching for a sorting algorithm that performs well over all possible inputs hence the average performance of the tree is its base fitness

- Premature convergence is prevented by using ranking of population rather than absolute performance difference between trees enabling exploring areas outside the neighbourhood of the highly fit trees

Results



AMD Athlon MP
Execution Time (microsec. per key)
0.4
0.35
0.3
0.25
0.2
0.15
100
1000
10000
100000
1e+06
1e+07
Standard Deviation
Quicksort
CC-radix
Multi-way Merge
Adaptive Sort
Gene Sort

GA - Advantages

1. Because only primitive procedures like "cut" and "exchange" of strings are used for generating new genes from old, it is easy to handle large problems simply by using long strings.

2. Because only values of the objective function for optimization are used to select genes, this algorithm can be robustly applied to problems with any kinds of objective functions, such as nonlinear, indifferentiable, or step functions;

3. Because the genetic operations are performed at random and also include mutation, it is possible to avoid being trapped by local-optima.

Conclusions

- Evolutionary Algorithms are heavily used in the search of solution spaces in many NP-Complete problems

- NP-Complete problems like Network Routing, TSP and even problems like Sorting are optimized by the use of Genetic Algorithms as they can rapidly locate good solutions, even for difficult search spaces.