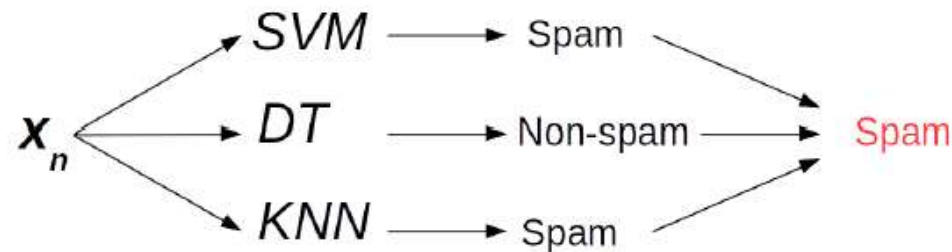
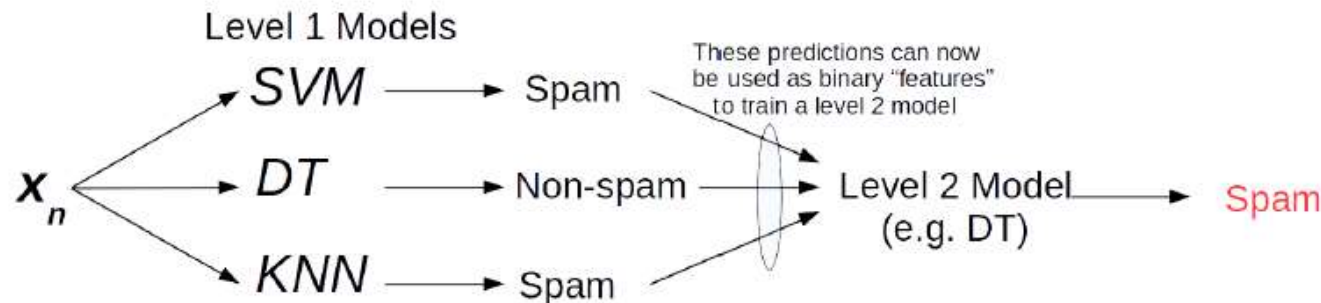

Ensemble Models

Simple Models

- Voting or Averaging of predictions of multiple pre-trained models

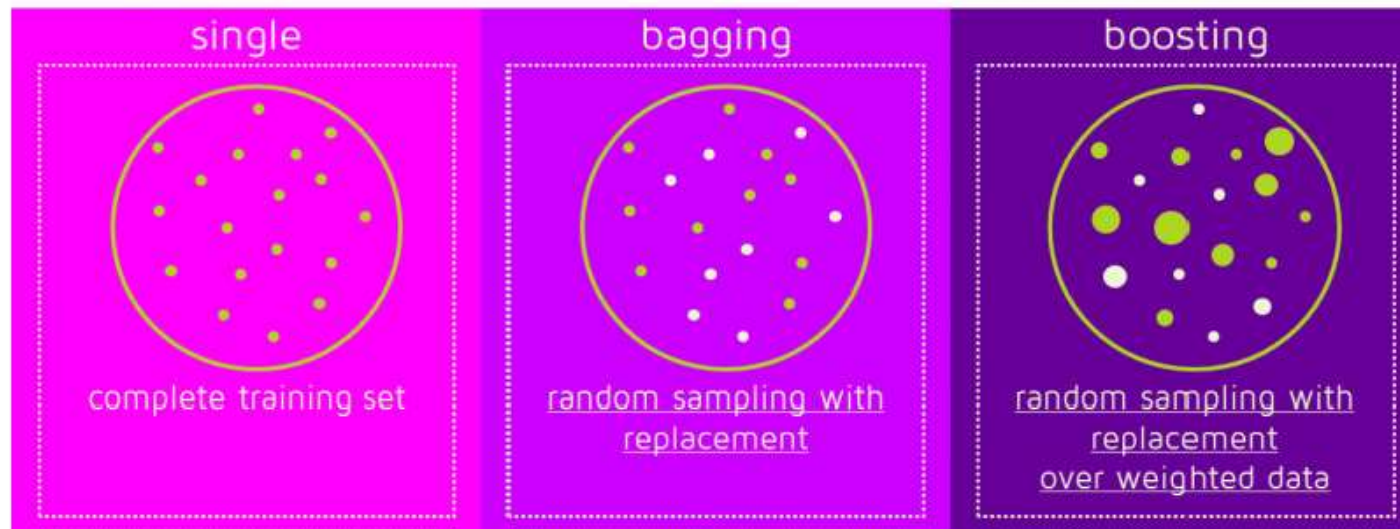


- “Stacking”: Use predictions of multiple models as “features” to train a new model and use the new model to make predictions on test data



New Approach

- Instead of training different models on same data, train **same model** multiple times on **different data sets**, and “combine” these “different” models
- We can use some simple/weak model as the base model
- How do we get multiple training data sets (in practice, we only have one data set at training time)?

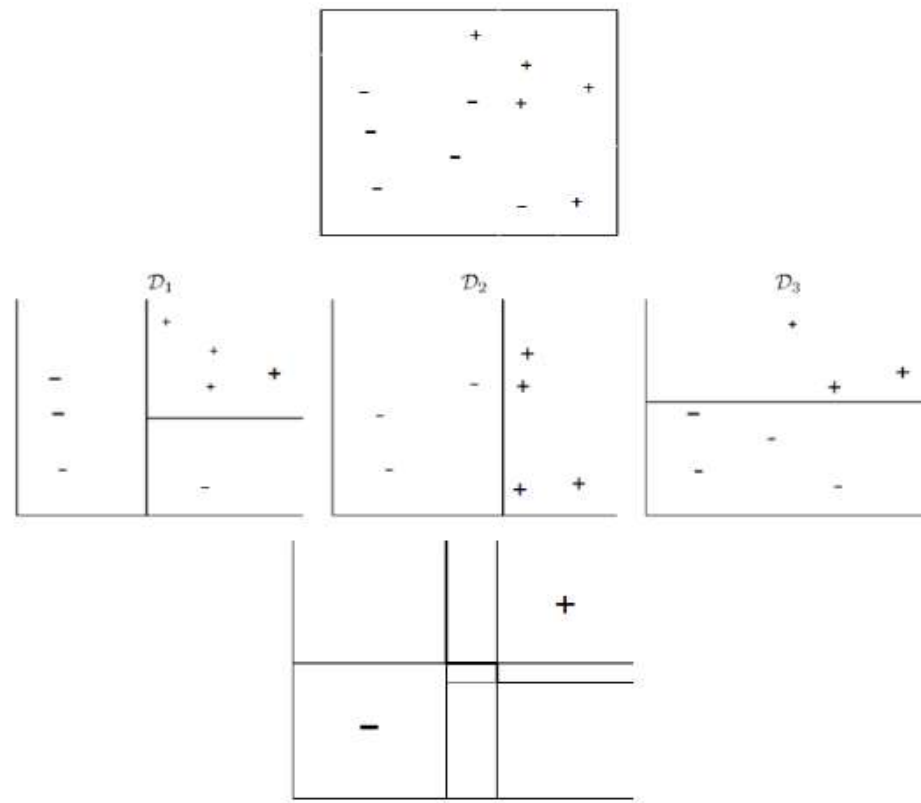


Bagging

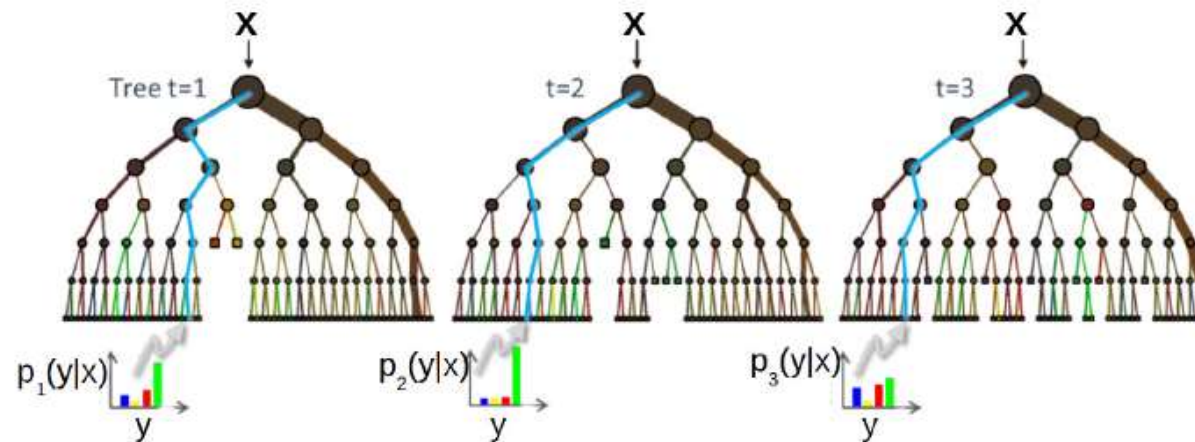
- Bagging stands for Bootstrap Aggregation
- Takes original data set D with N training examples
- Creates M copies $\{\tilde{D}_m\}_{m=1}^M$
 - Each \tilde{D}_m is generated from D by **sampling with replacement**
 - Each data set \tilde{D}_m has the same number of examples as in data set D
 - These data sets are reasonably different from each other (since only about 63% of the original examples appear in any of these data sets)
- Train models h_1, \dots, h_M using $\tilde{D}_1, \dots, \tilde{D}_M$, respectively
- Use an averaged model $h = \frac{1}{M} \sum_{m=1}^M h_m$ as the final model
- Useful for models with high variance and noisy data

Bagging

Top: Original data, Middle: 3 models (from some model class) learned using three data sets chosen via bootstrapping, Bottom: averaged model



Random Forests



- An ensemble of decision tree (DT) classifiers
- Uses bagging on features (each DT will use a random set of features)
 - Given a total of D features, each DT uses \sqrt{D} randomly chosen features
 - Randomly chosen features make the different trees uncorrelated
- All DTs usually have the same depth
- Each DT will split the training data differently at the leaves
- Prediction for a test example votes on/averages predictions from all the DTs

Boosting

- The basic idea
 - Take a weak learning algorithm
 - Only requirement: Should be slightly better than random
 - Turn it into an awesome one by making it focus on difficult cases
- Most boosting algorithms follow these steps:
 - 1 Train a weak model on some training data
 - 2 Compute the error of the model on each training example
 - 3 Give higher importance to examples on which the model made mistakes
 - 4 Re-train the model using “importance weighted” training examples
 - 5 Go back to step 2

AdaBoost

- Given: Training data $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)$ with $y_n \in \{-1, +1\}, \forall n$
- Initialize **weight** of each example (\mathbf{x}_n, y_n) : $D_1(n) = 1/N, \forall n$
- For round $t = 1 : T$
 - Learn a weak $h_t(\mathbf{x}) \rightarrow \{-1, +1\}$ using training data **weighted as per** D_t
 - Compute the **weighted** fraction of errors of h_t on this training data

$$\epsilon_t = \sum_{n=1}^N D_t(n) \mathbb{1}[h_t(\mathbf{x}_n) \neq y_n]$$

- Set “importance” of h_t : $\alpha_t = \frac{1}{2} \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$ (gets larger as ϵ_t gets smaller)
- **Update the weight** of each example

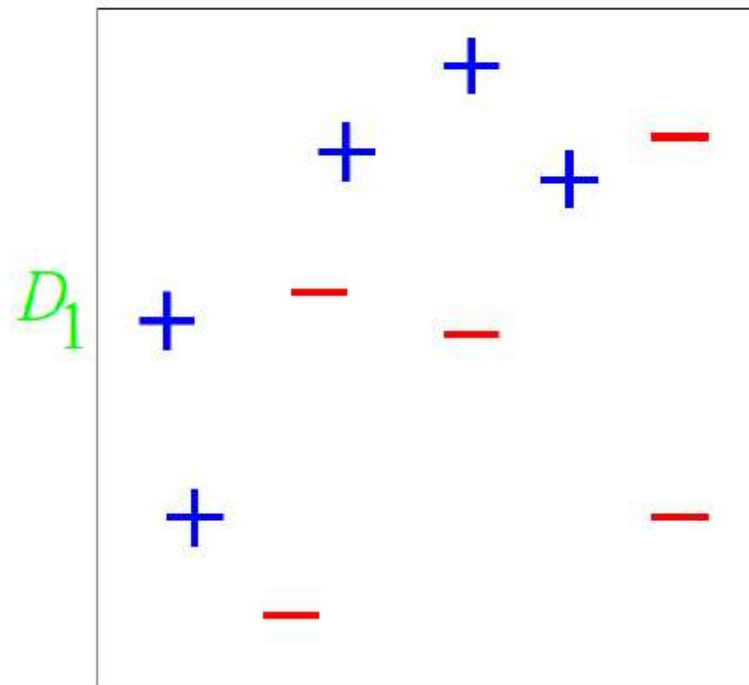
$$\begin{aligned} D_{t+1}(n) &\propto \begin{cases} D_t(n) \times \exp(-\alpha_t) & \text{if } h_t(\mathbf{x}_n) = y_n \quad (\text{correct prediction: decrease weight}) \\ D_t(n) \times \exp(\alpha_t) & \text{if } h_t(\mathbf{x}_n) \neq y_n \quad (\text{incorrect prediction: increase weight}) \end{cases} \\ &= D_t(n) \exp(-\alpha_t y_n h_t(\mathbf{x}_n)) \end{aligned}$$

- Normalize D_{t+1} so that it sums to 1: $D_{t+1}(n) = \frac{D_{t+1}(n)}{\sum_{m=1}^N D_{t+1}(m)}$
- Output the “boosted” final hypothesis $H(\mathbf{x}) = \text{sign}(\sum_{t=1}^T \alpha_t h_t(\mathbf{x}))$

AdaBoost Example

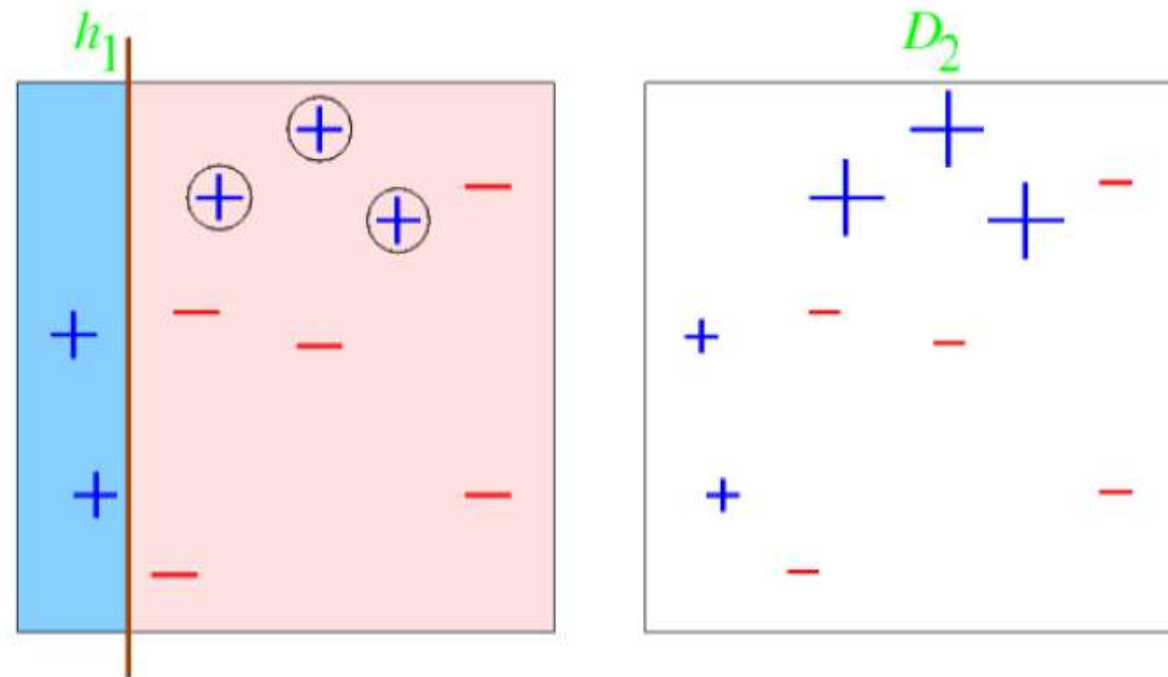
Consider binary classification with 10 training examples

Initial weight distribution D_1 is **uniform** (each point has equal weight = $1/10$)



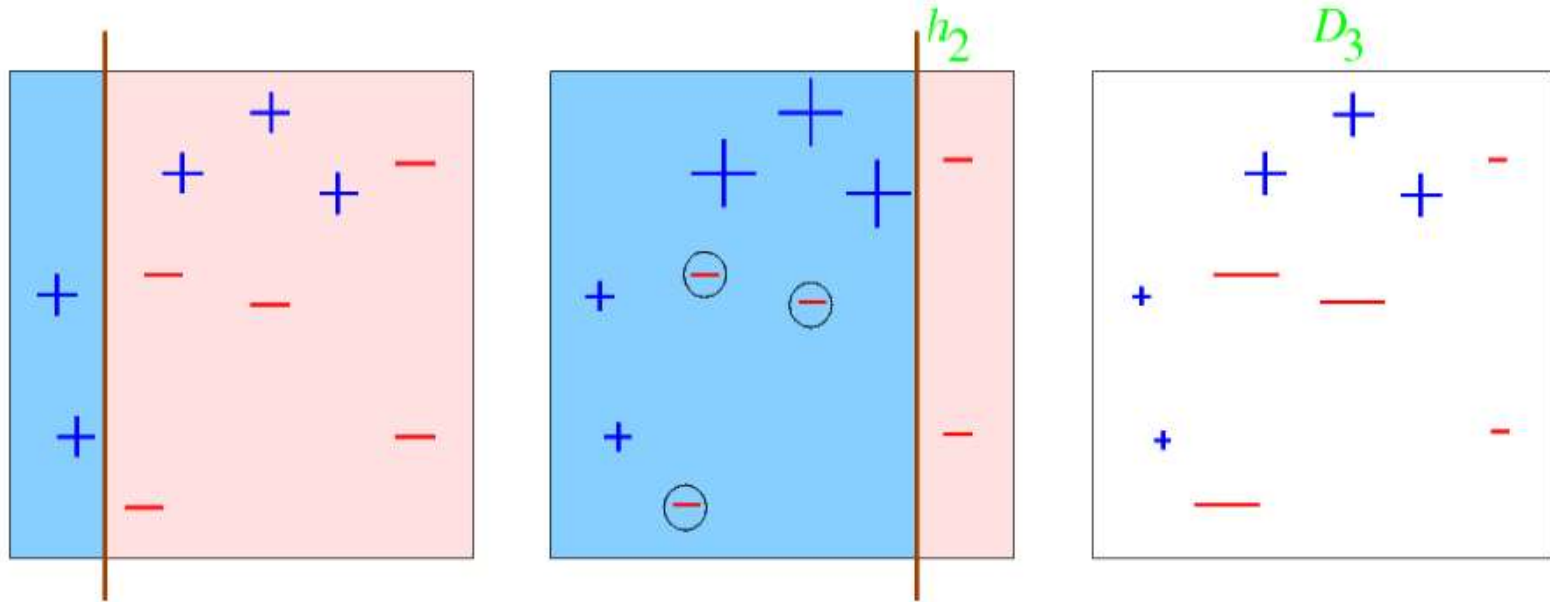
Each of our weak classifiers will be an **axis-parallel linear classifier**

AdaBoost Example



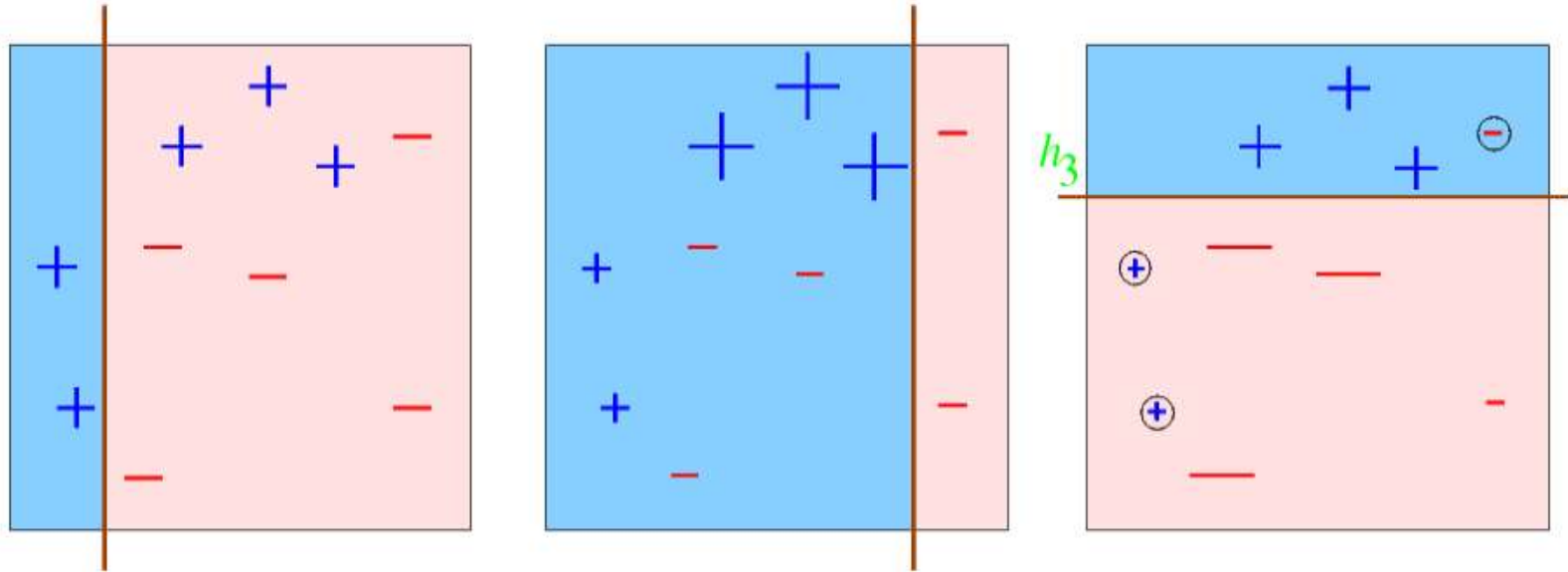
- Error rate of h_1 : $\epsilon_1 = 0.3$; weight of h_1 : $\alpha_1 = \frac{1}{2} \ln((1 - \epsilon_1)/\epsilon_1) = 0.42$
- Each **misclassified** point **upweighted** (weight multiplied by $\exp(\alpha_2)$)
- Each **correctly classified** point **downweighted** (weight multiplied by $\exp(-\alpha_2)$)

AdaBoost Example



- Error rate of h_2 : $\epsilon_2 = 0.21$; weight of h_2 : $\alpha_2 = \frac{1}{2} \ln((1 - \epsilon_2)/\epsilon_2) = 0.65$
- Each **misclassified** point **upweighted** (weight multiplied by $\exp(\alpha_2)$)
- Each **correctly classified** point **downweighted** (weight multiplied by $\exp(-\alpha_2)$)

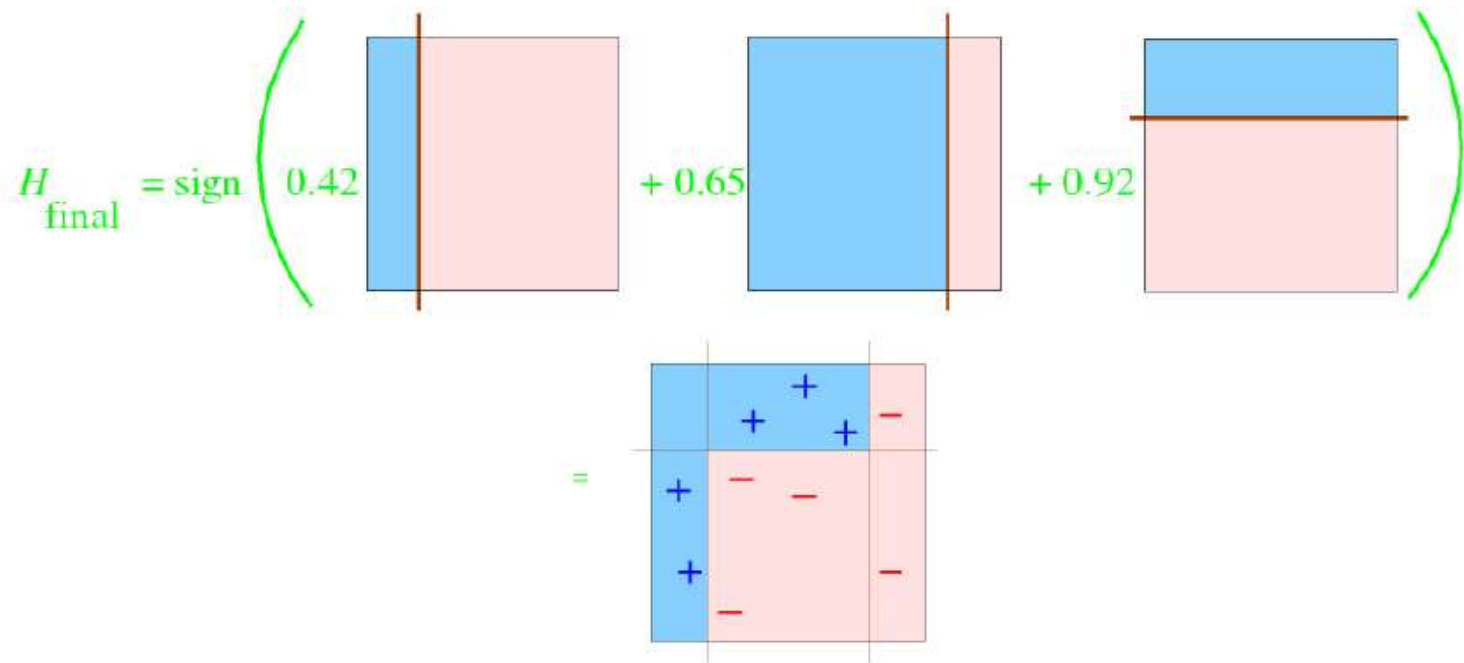
AdaBoost Example



- Error rate of h_3 : $\epsilon_3 = 0.14$; weight of h_3 : $\alpha_3 = \frac{1}{2} \ln((1 - \epsilon_3)/\epsilon_3) = 0.92$
- Suppose we decide to stop after round 3
- Our **ensemble** now consists of 3 classifiers: h_1, h_2, h_3

AdaBoost Example

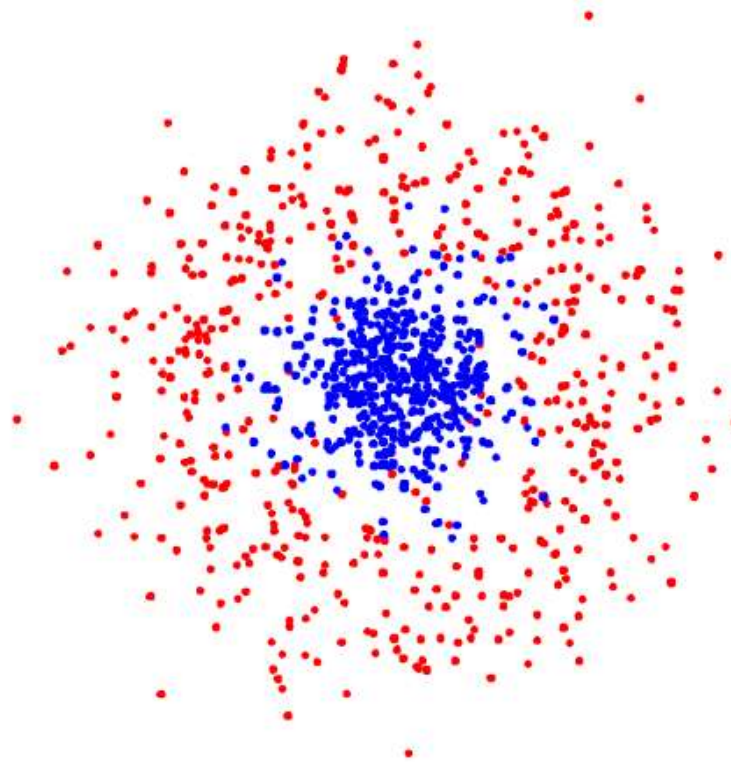
- Final classifier is a **weighted linear combination** of all the classifiers
- Classifier h_i gets a weight α_i



- Multiple **weak, linear classifiers combined** to give a **strong, nonlinear classifier**

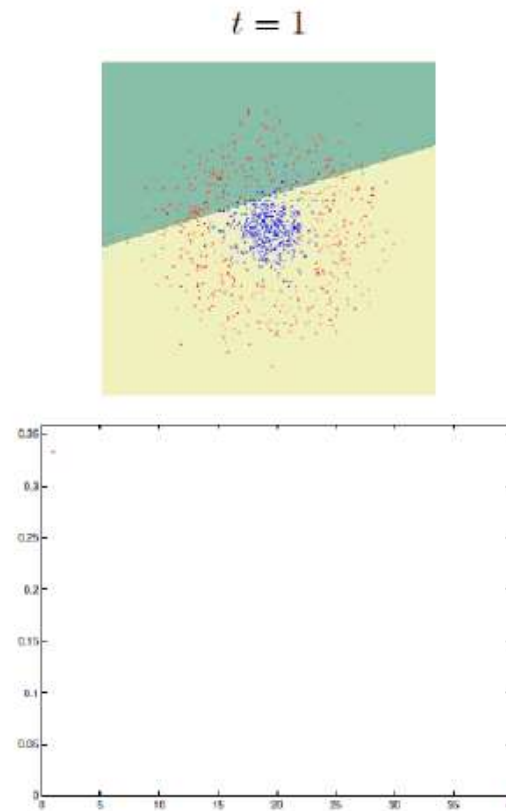
Second Example

- Given: A nonlinearly separable dataset
- We want to use Perceptron (linear classifier) on this data



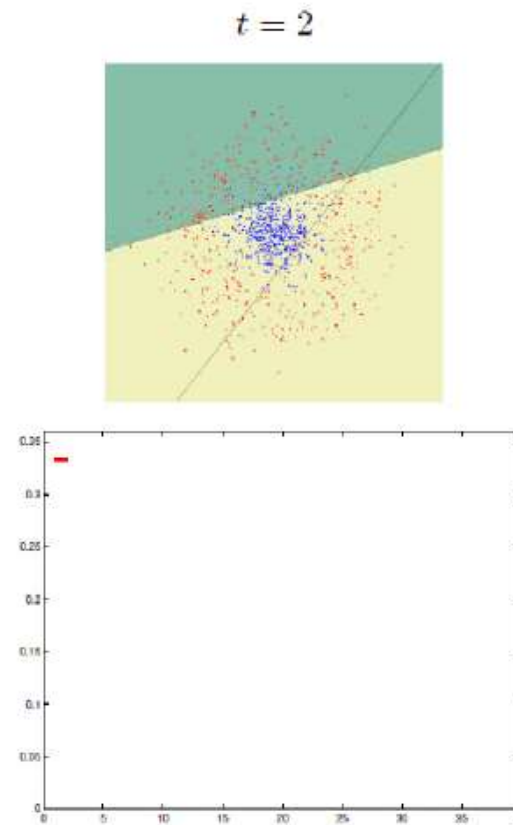
Second Example

- After round 1, our ensemble has 1 linear classifier (Perceptron)
- Bottom figure: X axis is number of rounds, Y axis is training error



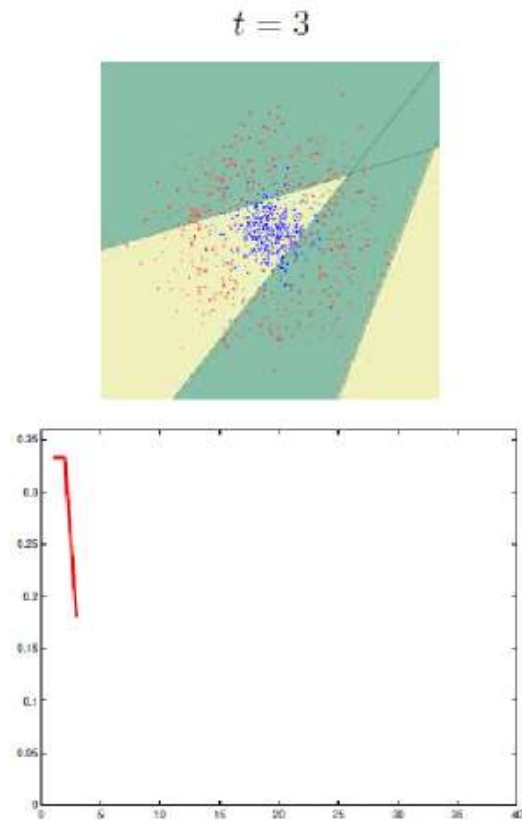
Second Example

- After round 2, our ensemble has 2 linear classifiers (Perceptrons)
- Bottom figure: X axis is number of rounds, Y axis is training error



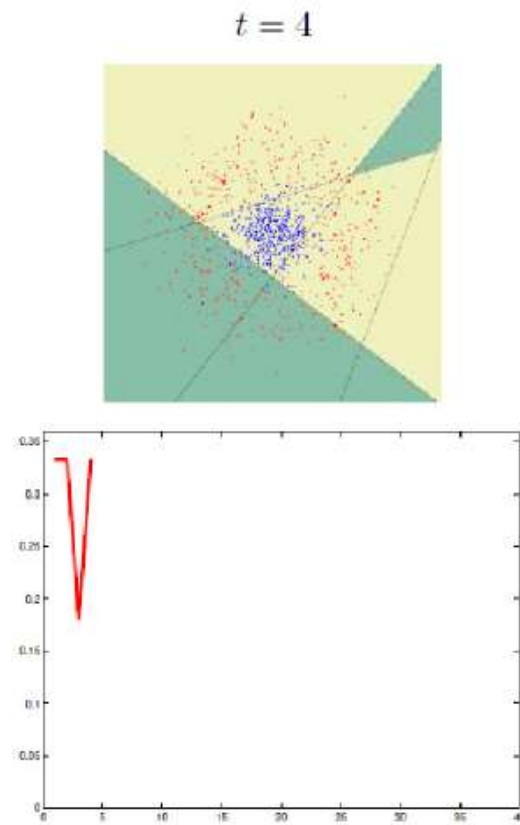
Second Example

- After round 3, our ensemble has 3 linear classifiers (Perceptrons)
- Bottom figure: X axis is number of rounds, Y axis is training error



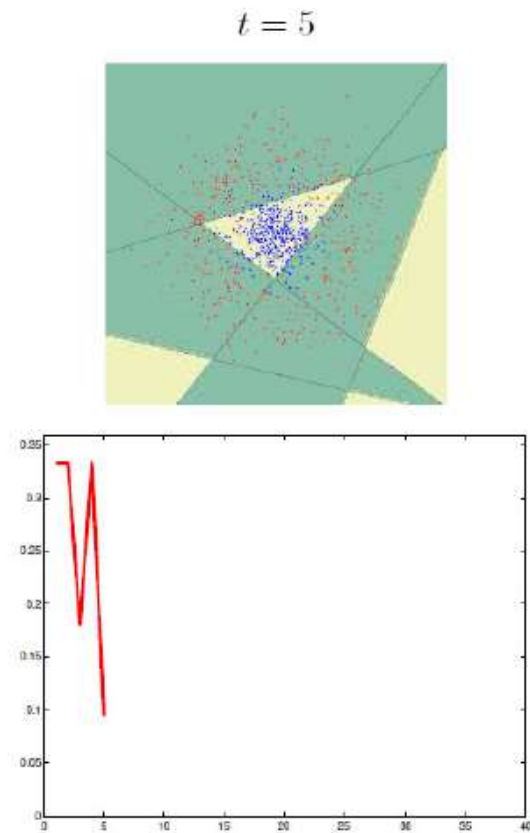
Second Example

- After round 4, our ensemble has 4 linear classifiers (Perceptrons)
- Bottom figure: X axis is number of rounds, Y axis is training error



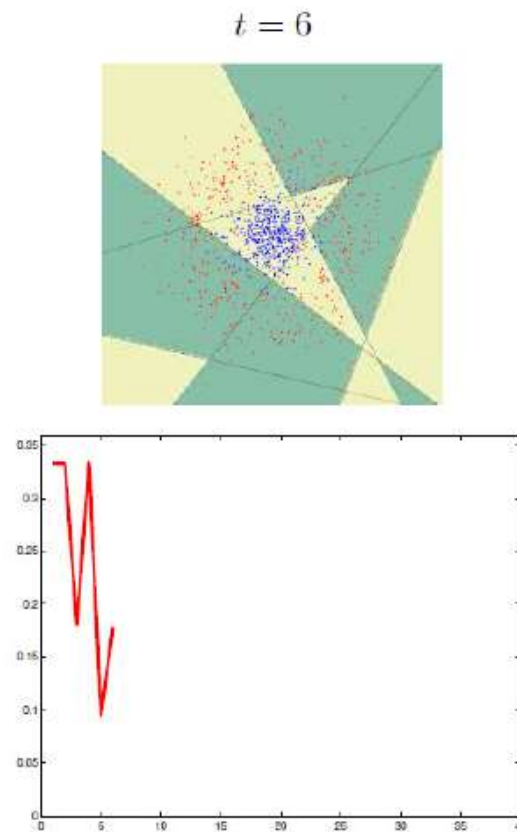
Second Example

- After round 5, our ensemble has 5 linear classifiers (Perceptrons)
- Bottom figure: X axis is number of rounds, Y axis is training error



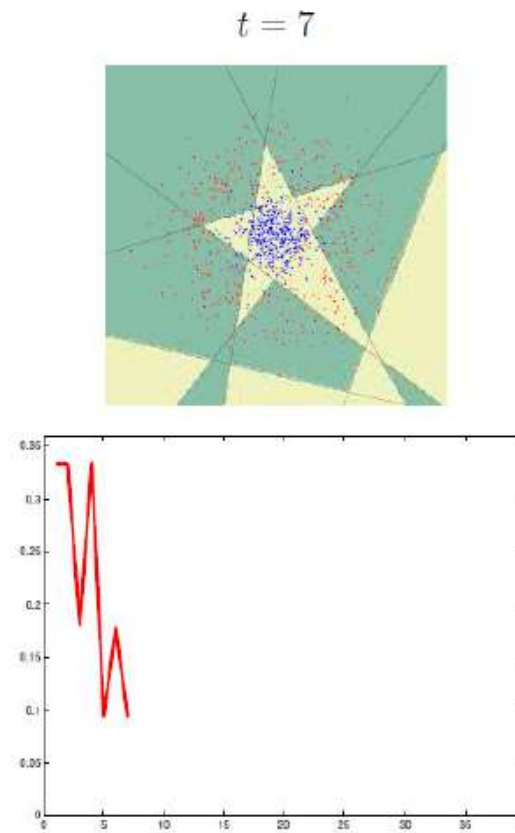
Second Example

- After round 6, our ensemble has 6 linear classifiers (Perceptrons)
- Bottom figure: X axis is number of rounds, Y axis is training error



Second Example

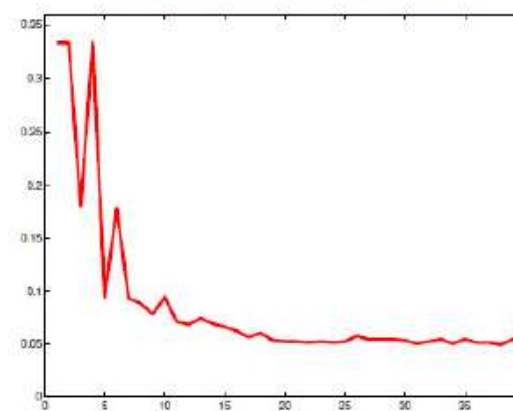
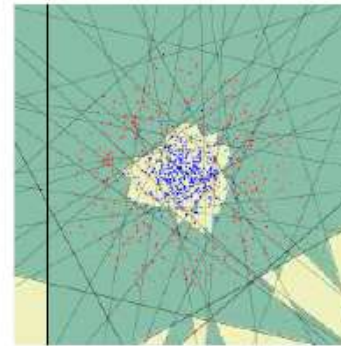
- After round 7, our ensemble has 7 linear classifiers (Perceptrons)
- Bottom figure: X axis is number of rounds, Y axis is training error



Second Example

- After round 40, our ensemble has 40 linear classifiers (Perceptrons)
- Bottom figure: X axis is number of rounds, Y axis is training error

$t = 40$



Comments

- For AdaBoost, given each model's error $\epsilon_t = 1/2 - \gamma_t$, the training error consistently gets better with rounds

$$\text{train-error}(H_{\text{final}}) \leq \exp(-2 \sum_{t=1}^T \gamma_t^2)$$

- Boosting algorithms can be shown to be minimizing a loss function
 - E.g., AdaBoost has been shown to be minimizing an exponential loss

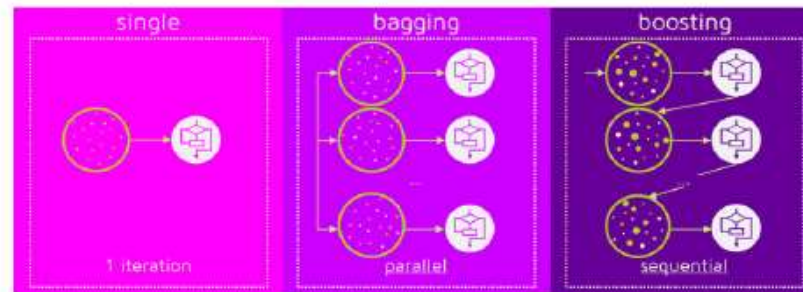
$$\mathcal{L} = \sum_{n=1}^N \exp\{-y_n H(\mathbf{x}_n)\}$$

where $H(\mathbf{x}) = \text{sign}(\sum_{t=1}^T \alpha_t h_t(\mathbf{x}))$, given weak base classifiers h_1, \dots, h_T

- Boosting in general can perform badly if some examples are outliers

Comparison

- No clear winner; usually depends on the data
- Bagging is computationally more efficient than boosting (note that bagging can train the M models in parallel, boosting can't)



- Both reduce variance (and overfitting) by combining different models
 - The resulting model has higher stability as compared to the individual ones
- Bagging usually can't reduce the bias, boosting can (note that in boosting, the training error steadily decreases)
- Bagging usually performs better than boosting if we don't have a high bias and only want to reduce variance (i.e., if we are overfitting)

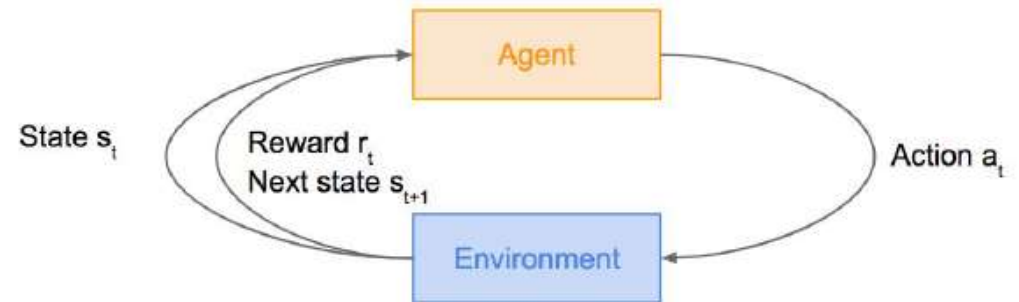
Reinforcement Learning

Adopted from Fei-Fei Li, Justin Johnson, Serena Yeung

Introduction

Problems involving an **agent** interacting with an **environment**, which provides numeric **reward** signals

Goal: Learn how to take actions in order to maximize reward



Introduction

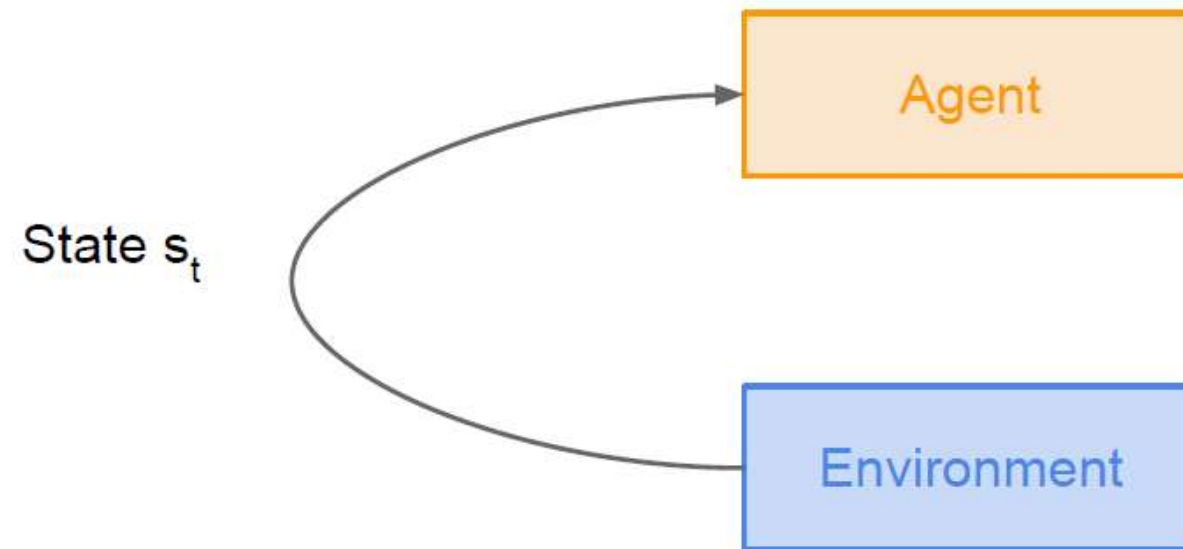


Agent

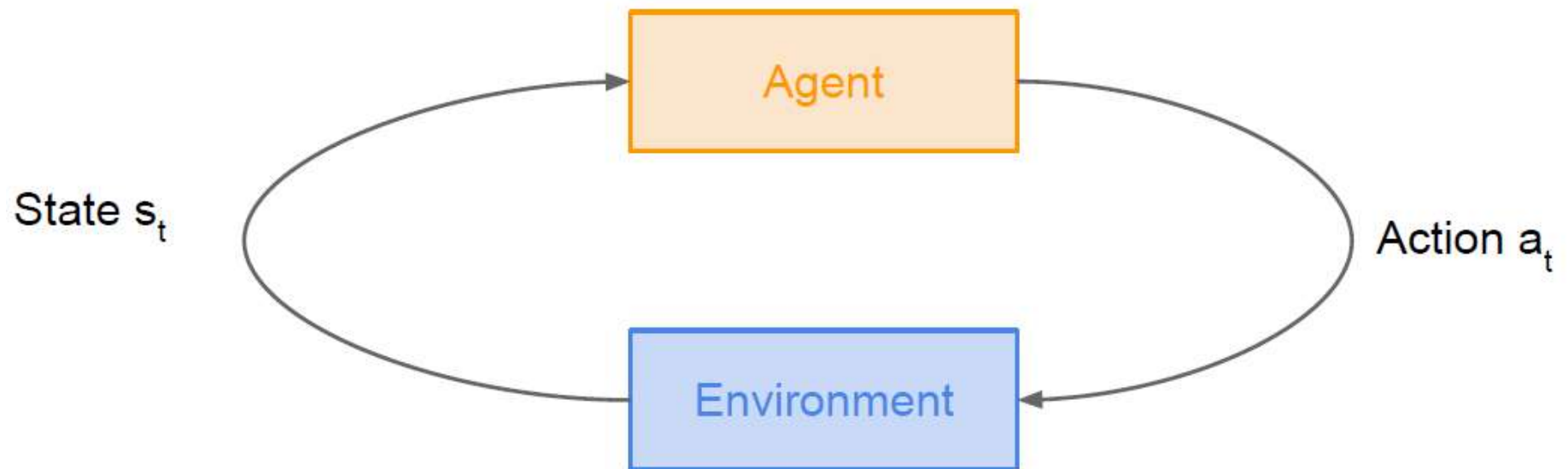
The diagram consists of two vertically stacked rectangular boxes. The top box is light orange with an orange border and contains the word 'Agent' in orange text. The bottom box is light blue with a blue border and contains the word 'Environment' in blue text.

Environment

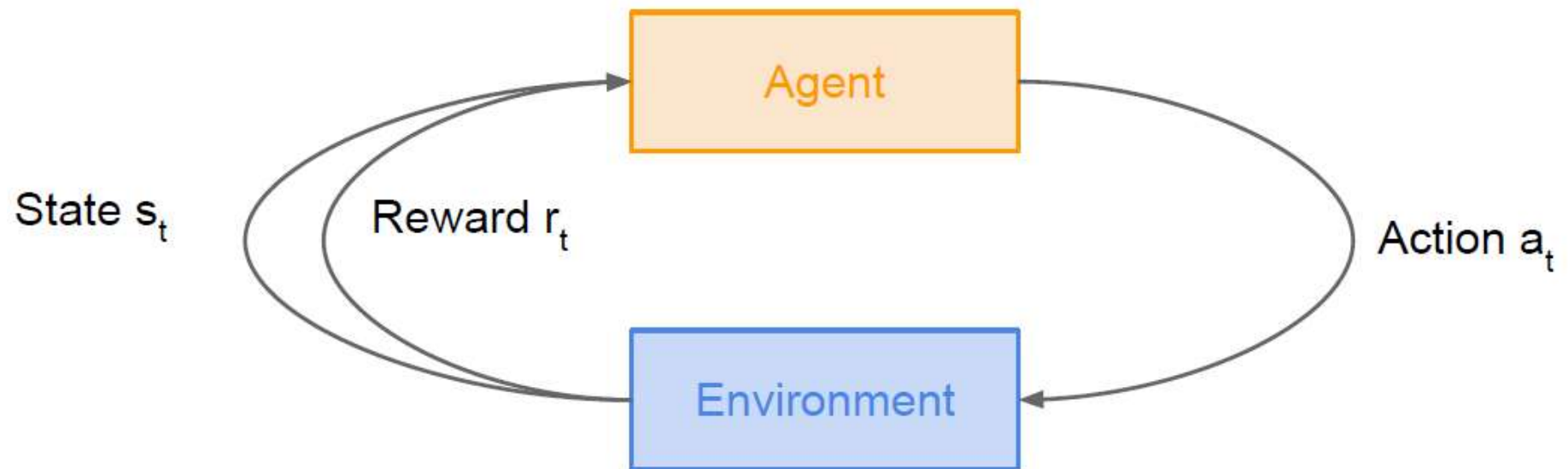
Introduction



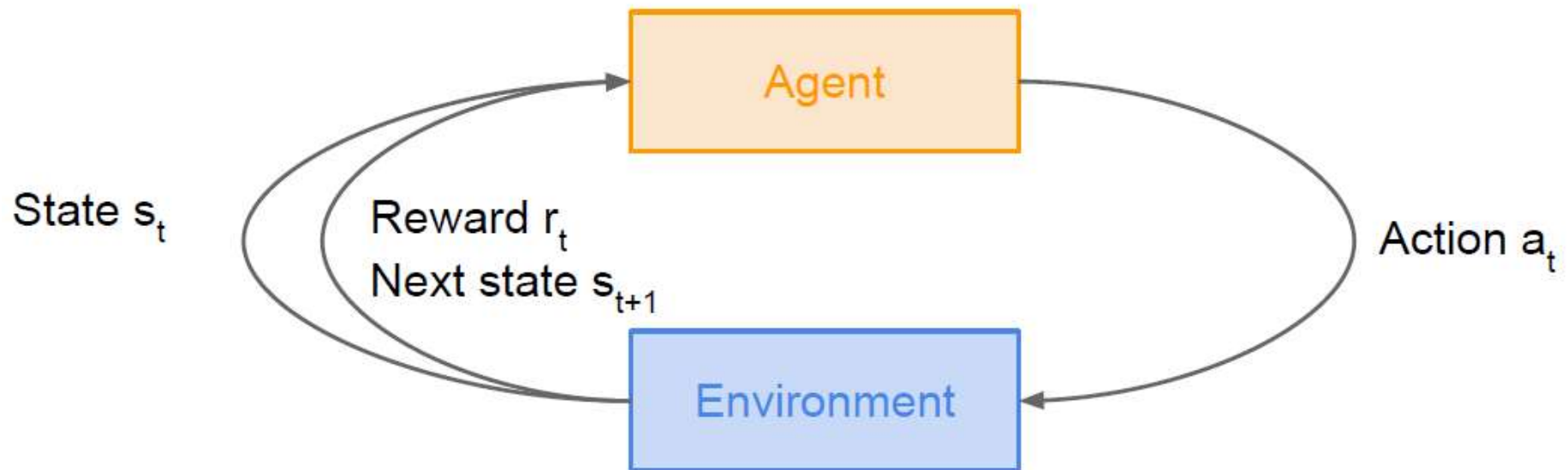
Introduction



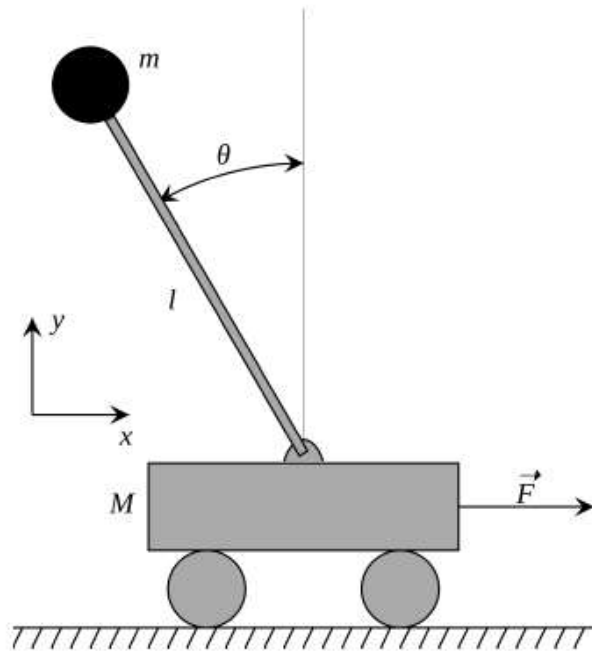
Introduction



Introduction



Introduction



Objective: Balance a pole on top of a movable cart

State: angle, angular speed, position, horizontal velocity

Action: horizontal force applied on the cart

Reward: 1 at each time step if the pole is upright

Passive vs Active Learning

- Passive learning
 - The agent simply watches the world going by and tries to learn the utilities of being in various states
- Active learning
 - The agent not simply watches, but also acts

Passive Learning

```
function PASSIVE-RL-AGENT(e) returns an action
  static: U, a table of utility estimates
           N, a table of frequencies for states
           M, a table of transition probabilities from state to state
           percepts, a percept sequence (initially empty)

  add e to percepts
  increment N[STATE[e]]
  U ← UPDATE(U, e, percepts, M, N)
  if TERMINAL?[e] then percepts ← the empty sequence
  return the action Observe
```

Markov Decision Process

- Mathematical formulation of the RL problem
- **Markov property**: Current state completely characterises the state of the world

Defined by: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$

\mathcal{S} : set of possible states

\mathcal{A} : set of possible actions

\mathcal{R} : distribution of reward given (state, action) pair

\mathbb{P} : transition probability i.e. distribution over next state given (state, action) pair

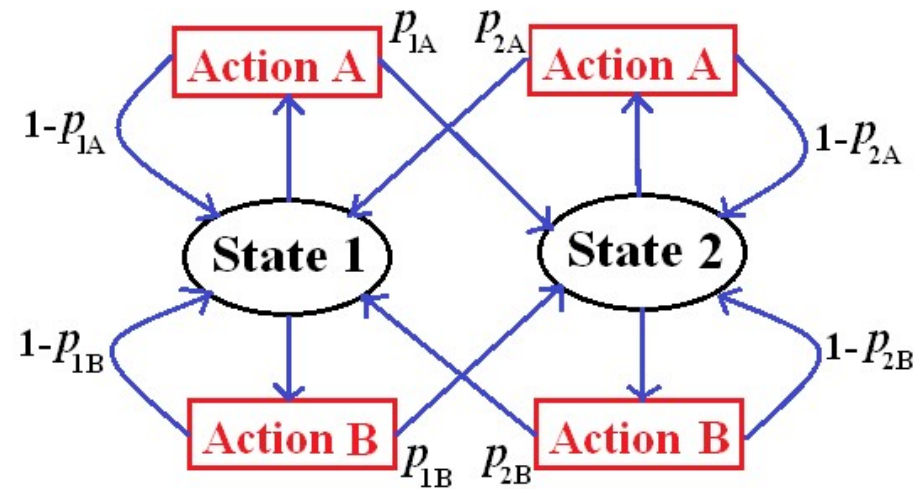
γ : discount factor

Markov Decision Process

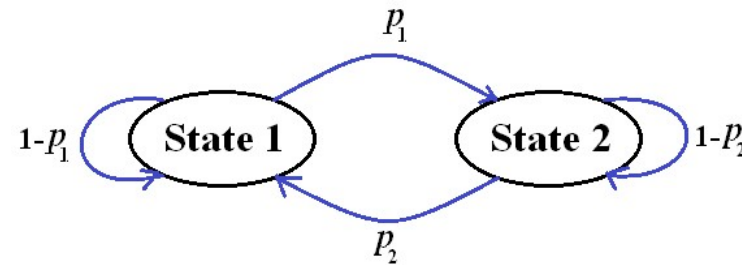
- At time step $t=0$, environment samples initial state $s_0 \sim p(s_0)$
- Then, for $t=0$ until done:
 - Agent selects action a_t
 - Environment samples reward $r_t \sim R(\cdot | s_t, a_t)$
 - Environment samples next state $s_{t+1} \sim P(\cdot | s_t, a_t)$
 - Agent receives reward r_t and next state s_{t+1}
- A policy π is a function from S to A that specifies what action to take in each state
- **Objective:** find policy π^* that maximizes cumulative discounted reward: $\sum_{t \geq 0} \gamma^t r_t$

Markov Decision Process

Markov
Decision
Process



Markov
Chain



Example

actions = {

1. right →

2. left ←

3. up ↑

4. down ↓

}

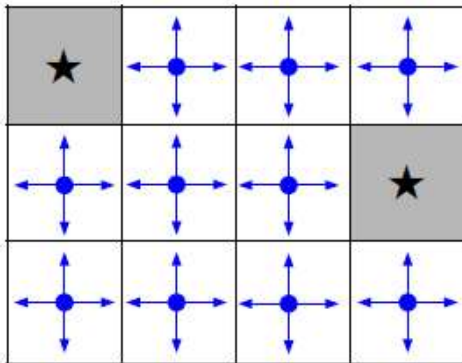
states

★			
			★

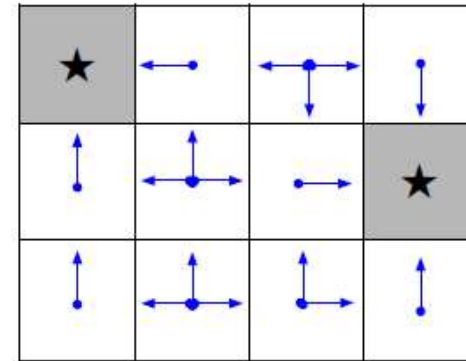
Set a negative “reward”
for each transition
(e.g. $r = -1$)

Objective: reach one of terminal states (greyed out) in
least number of actions

Example



Random Policy



Optimal Policy

Optimal Policy

We want to find optimal policy π^* that maximizes the sum of rewards.

How do we handle the randomness (initial state, transition probability...)?
Maximize the **expected sum of rewards!**

$$\text{Formally: } \pi^* = \arg \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | \pi \right] \text{ with } s_0 \sim p(s_0), a_t \sim \pi(\cdot | s_t), s_{t+1} \sim p(\cdot | s_t, a_t)$$

Value Function

Following a policy produces sample trajectories (or paths) $s_0, a_0, r_0, s_1, a_1, r_1, \dots$

How good is a state?

The **value function** at state s , is the expected cumulative reward from following the policy from state s :

$$V^\pi(s) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, \pi \right]$$

How good is a state-action pair?

The **Q-value function** at state s and action a , is the expected cumulative reward from taking action a in state s and then following the policy:

$$Q^\pi(s, a) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi \right]$$

Q-function

The optimal Q-value function Q^* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s, a) = \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi \right]$$

Q^* satisfies the following **Bellman equation**:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a \right]$$

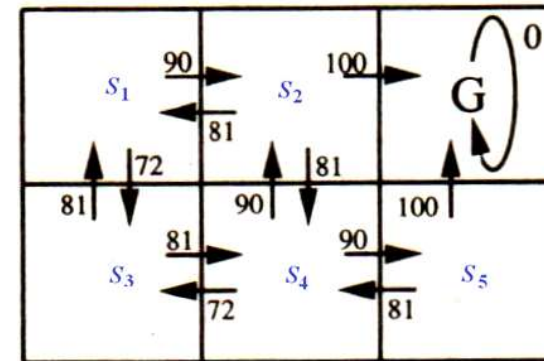
Intuition: if the optimal state-action values for the next time-step $Q^*(s', a')$ are known, then the optimal strategy is to take the action that maximizes the expected value of $r + \gamma Q^*(s', a')$

The optimal policy π^* corresponds to taking the best action in any state as specified by Q^*

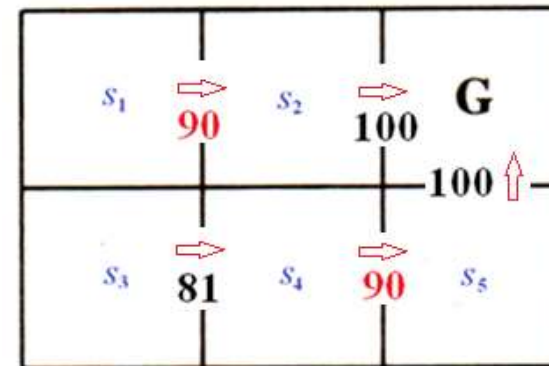
Q-function

$Q(s,a)$ ↑ ↓ ← →

s_1	0	72	0	90
s_2	0	81	81	100
s_3	81	0	0	81
s_4	90	0	72	90
s_5	100	0	81	0
G	0	0	0	0



$Q(s,a)$ values



π^*

Solution

Value iteration algorithm: Use Bellman equation as an iterative update

$$Q_{i+1}(s, a) = \mathbb{E} \left[r + \gamma \max_{a'} Q_i(s', a') | s, a \right]$$

Q_i will converge to Q^* as $i \rightarrow \infty$

What's the problem with this?

Not scalable. Must compute $Q(s, a)$ for every state-action pair. If state is e.g. current game state pixels, computationally infeasible to compute for entire state space!

Solution: use a function approximator to estimate $Q(s, a)$. E.g. a neural network!

Policy Iteration Algorithm

Initialize a policy π' arbitrarily

Repeat

$$\pi \leftarrow \pi'$$

Compute the values using π by
solving the linear equations

$$V^\pi(s) = E[r|s, \pi(s)] + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, \pi(s)) V^\pi(s')$$

Improve the policy at each state

$$\pi'(s) \leftarrow \arg \max_a (E[r|s, a] + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V^\pi(s'))$$

Until $\pi = \pi'$

Exploration - Exploitation

Exploration of unknown states and actions to gather new information

Exploitation of learned states and actions to maximize the cumulative reward

□ ϵ -greedy search:

Explore – with probability ϵ choose uniformly one action among all possible actions.

Exploit – with probability $1-\epsilon$ choose the best action.

Start with a high ϵ and gradually decrease it in order initiate exploitation once enough exploration.

Probabilistic Search

Choose action a according to probability

$$P(a | s) = \frac{\exp Q(s, a)}{\sum_{b \in A} \exp Q(s, b)}$$

Move from exploration to exploitation using

$$P(a | s) = \frac{\exp[Q(s, a)/T]}{\sum_{b=1}^A \exp[Q(s, b)/T]}$$

Start with a large T and gradually decrease it.

T large, $P(a | s) \approx 1/A$ (constant) \Rightarrow exploration

T small, better actions \rightarrow exploitation.

Deep Q-learning

Q-learning: Use a function approximator to estimate the action-value function

$$Q(s, a; \theta) \approx Q^*(s, a)$$

 function parameters (weights)

If the function approximator is a deep neural network => **deep q-learning!**

Deep Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} [(y_i - Q(s, a; \theta_i))^2]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$

Iteratively try to make the Q-value close to the target value (y_i) it should have, if Q-function corresponds to optimal Q^* (and optimal policy π^*)

Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right] \nabla_{\theta_i} Q(s, a; \theta_i)$$

Training

Learning from batches of consecutive samples is problematic:

- Samples are correlated => inefficient learning
- Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand size) => can lead to bad feedback loops

Address these problems using **experience replay**

- Continually update a **replay memory** table of transitions (s_t, a_t, r_t, s_{t+1}) as game (experience) episodes are played
- Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples

Each transition can also contribute
to multiple weight updates
=> greater data efficiency

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$

end for

end for

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

← Initialize replay memory, Q-network

for episode = 1, M **do**

 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$

end for

end for

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$

end for

end for

← Play M episodes (full games)

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$

end for

end for

← Initialize state
(starting game
screen pixels) at the
beginning of each
episode

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$

end for

end for

← For each timestep t
of the game

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$

end for

end for

← With small probability,
select a random
action (explore),
otherwise select
greedy action from
current policy

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$

end for

end for

Take the action (a_t),
and observe the
reward r_t and next
state s_{t+1}

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$

end for

end for

← Store transition in
replay memory

Deep Q-learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$

end for

end for

← Experience Replay:
Sample a random
minibatch of transitions
from replay memory
and perform a gradient
descent step

Swarm Intelligence

The need for new Computing Techniques

The computer revolution changed human societies:

- communication
- transportation
- industrial production
- administration, writing, and bookkeeping
- technological advances / science
- entertainment

However, some problems cannot be tackled with traditional hardware and software!

The need for new Computing Techniques

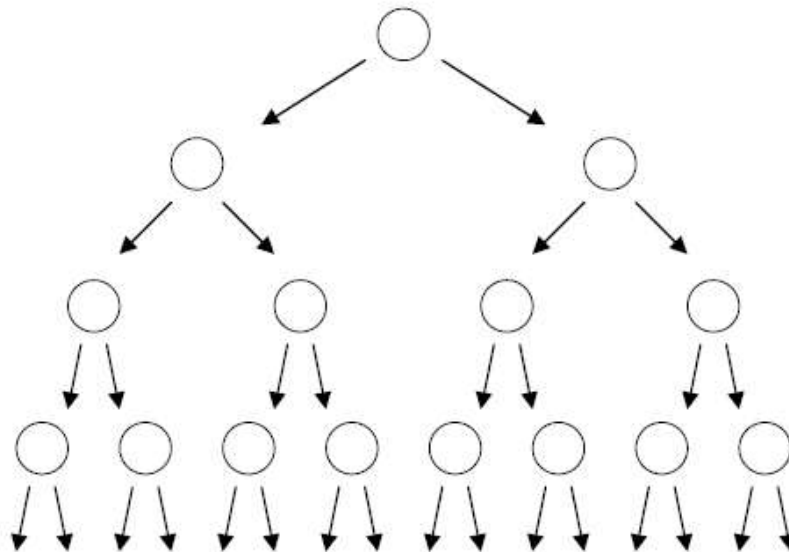
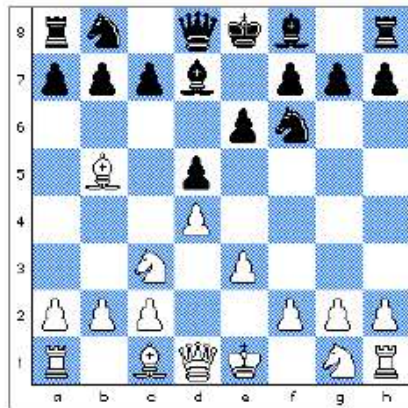
Computing tasks have to be

- well-defined
- fairly predictable
- computable in reasonable time with serial computers

Hard Problems

Well-defined, but computational hard problems

- NP hard problems (Travelling Salesman Problem)
- Action-response planning (Chess playing)

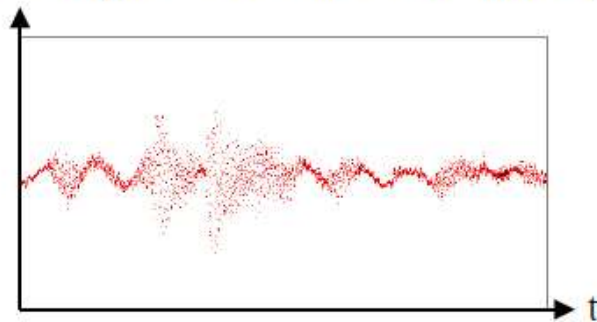


Hard Problems

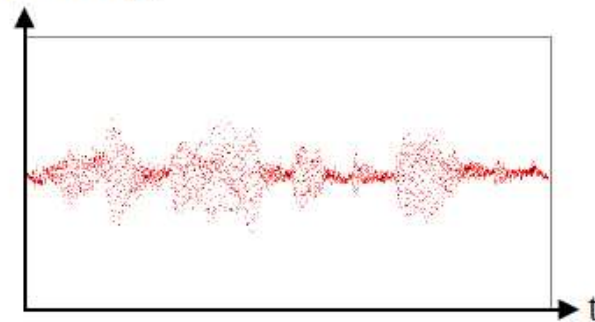
Fuzzy problems

- intelligent human-machine interaction
- natural language understanding

Example: Fuzziness in sound processing



“E-vo-lu-tio-na-ry Com-pu-ta-tion”



“E-vo-lu-tio-na-ry Com-pu-ta-tion”

Hard Problems

Hardly predictable and dynamic problems

- real-world autonomous robots
- management and business planning



Japanese piano robot

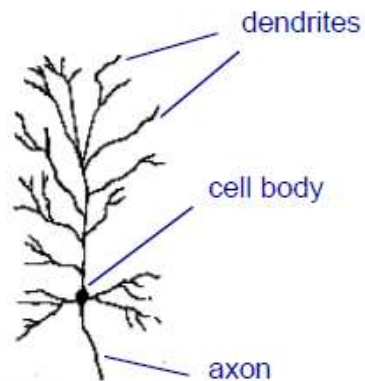


Trade at the stock exchange

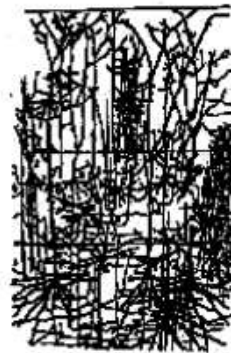
Alternatives

- DNA based computing (chemical computation)
- Quantum computing (quantum-physical computation)
- Bio-computing (simulation of biological mechanisms)

Artificial Networks



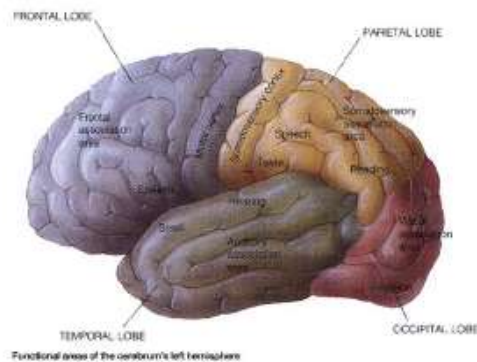
The basic unit - the neurone



Vertical cut through the neocortex of a cat

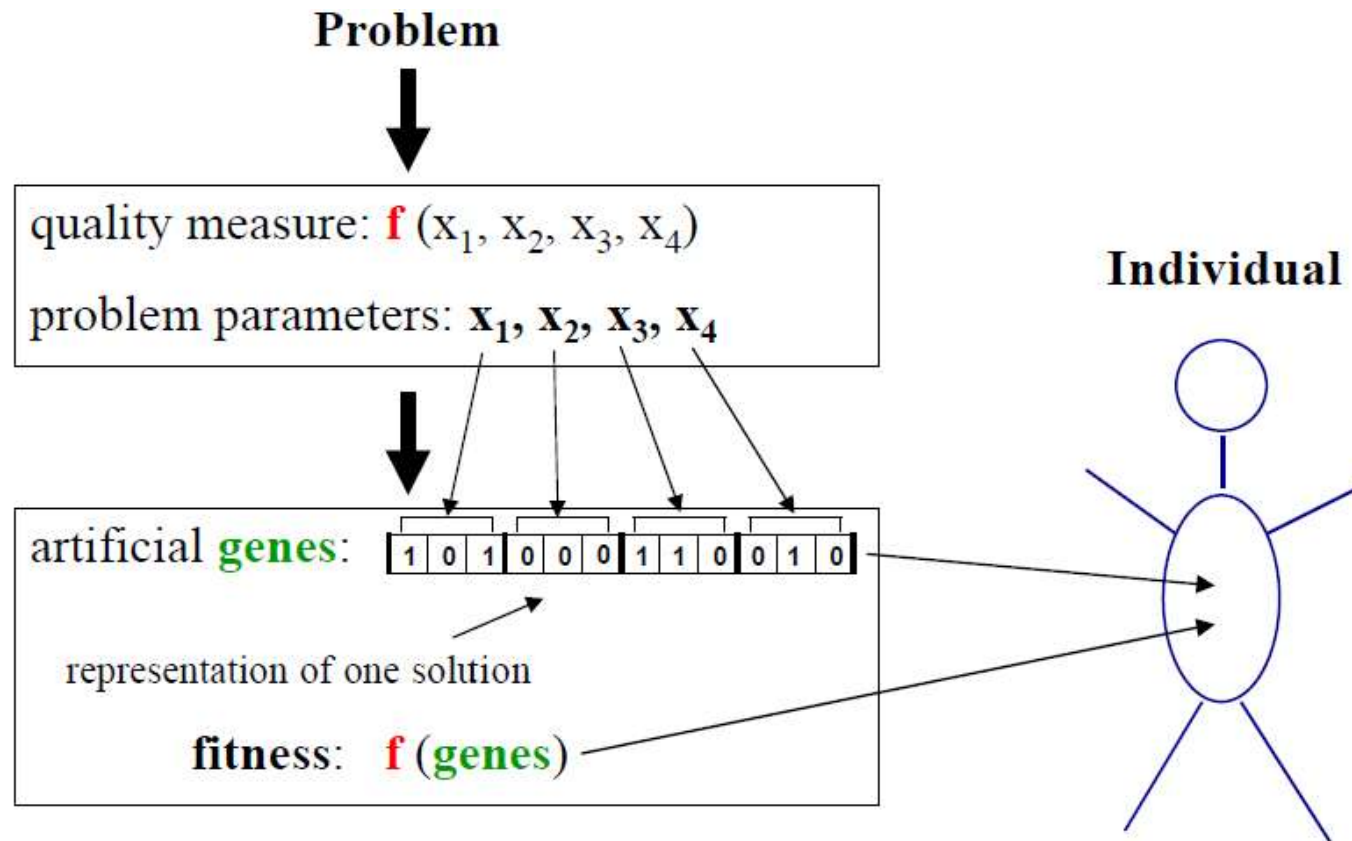
Properties of the brain

- holistic
- parallel
- associative
- learning
- redundancy
- self-organisation

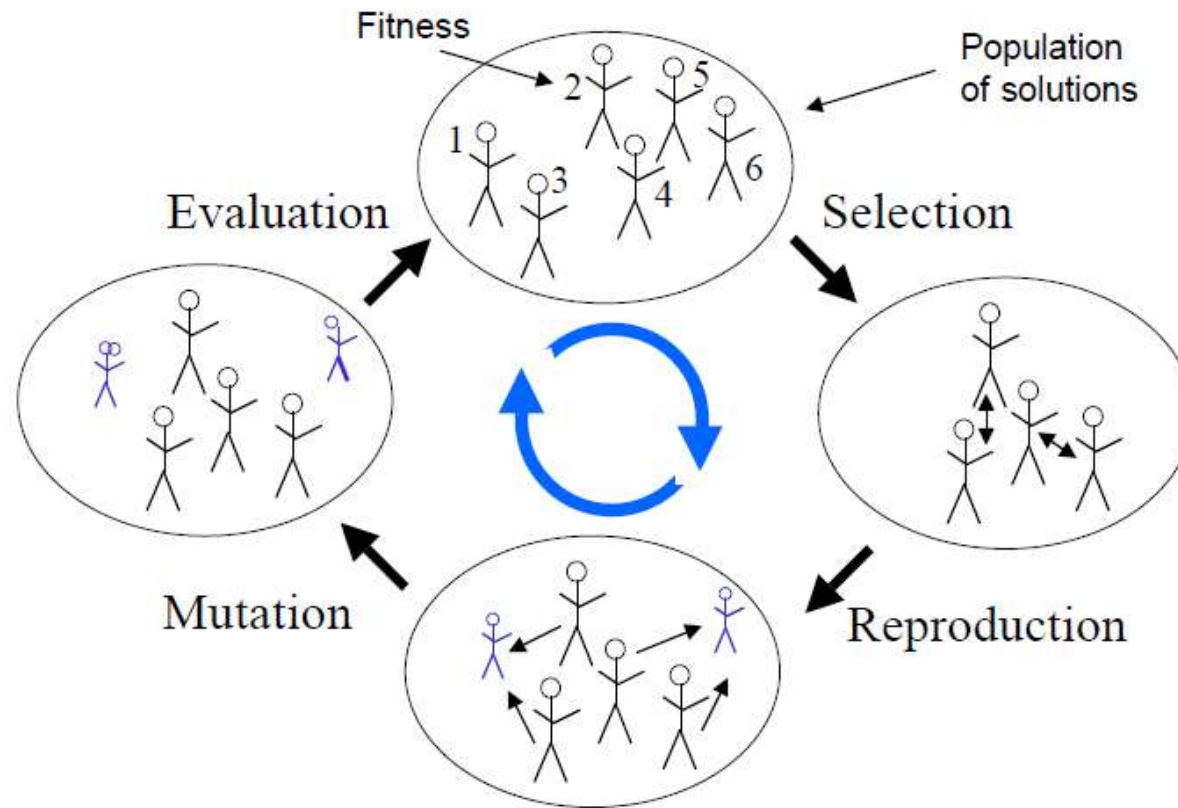


Functional units of the human brain

Evolutionary Computation



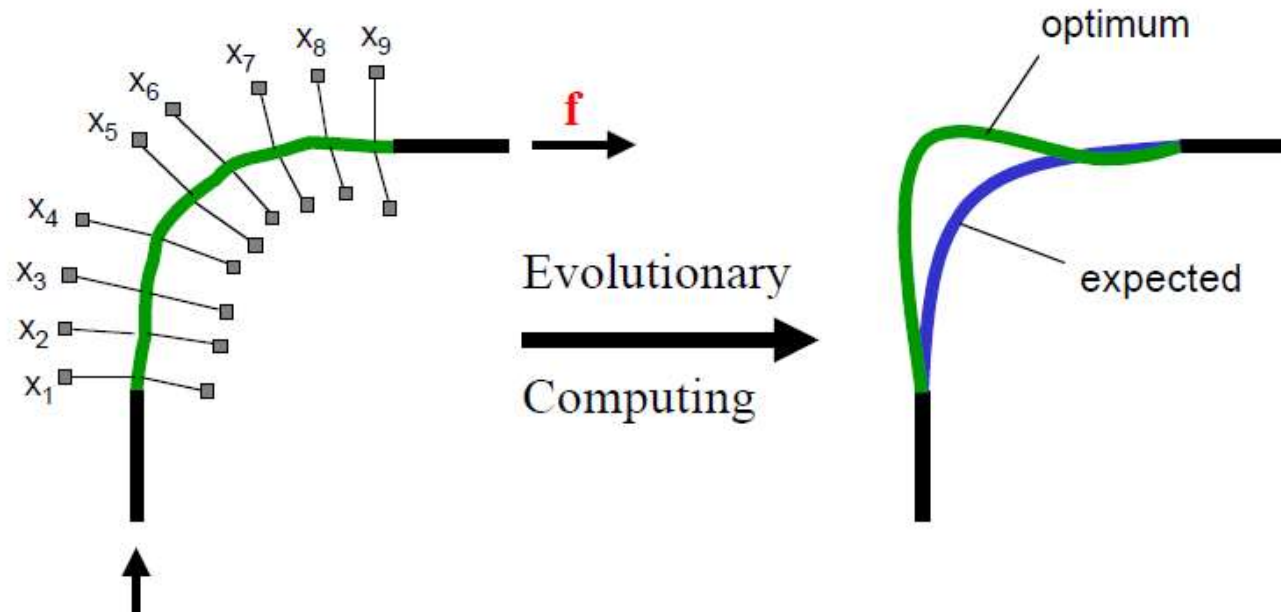
Evolutionary Computation














Evolutionary Computation

The task: Design a bent tube with a maximum flow

Goal: water flow $f(x_1, x_2, \dots, x_9) = f_{\max}$



Bio-Computing

	Inspiration	Identification	Application	Verification
Natural sciences				
Complexity theory				
Adaptive algorithms				
Artificial Life				
Swarm Intelligence				

Applications

- Robotics / Artificial Intelligence
- Process optimisation / Staff scheduling
- Telecommunication companies
- Entertainment



Limitations

- biology makes compromises between different goals
- biology sometimes fails
- some natural mechanisms are not well understood
- well-defined problems can be solved by better means



Swarm Intelligence

“The emergent collective intelligence of groups of simple agents.”
(Bonabeau et al, 1999)

Examples

- group foraging of social insects
- cooperative transportation
- division of labour
- nest-building of social insects
- collective sorting and clustering

Swarm Intelligence

Analogies in IT and social insects

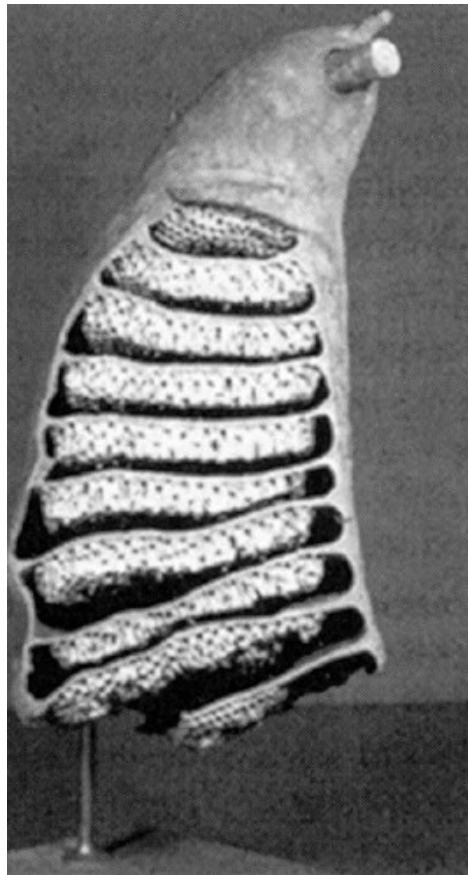
- distributed system of interacting autonomus agents
- goals: performance optimization and robustness
- self-organized control and cooperation (decentralized)
- division of labour and distributed task allocation
- indirect interactions

Swarm Intelligence

The 3 step process

- **identification of analogies:** in swarm biology and IT systems
- **understanding:** computer modelling of realistic swarm biology
- **engineering:** model simplification and tuning for IT applications

Model Examples



Model Examples



Ants

Why are ants interesting?

- ants solve complex tasks by simple local means
- ant productivity is better than the sum of their single activities
- ants are ‘grand masters’ in search and exploitation

Which mechanisms are important?

- cooperation and division of labour
- adaptive task allocation
- work stimulation by cultivation
- pheromones



Self-Organization

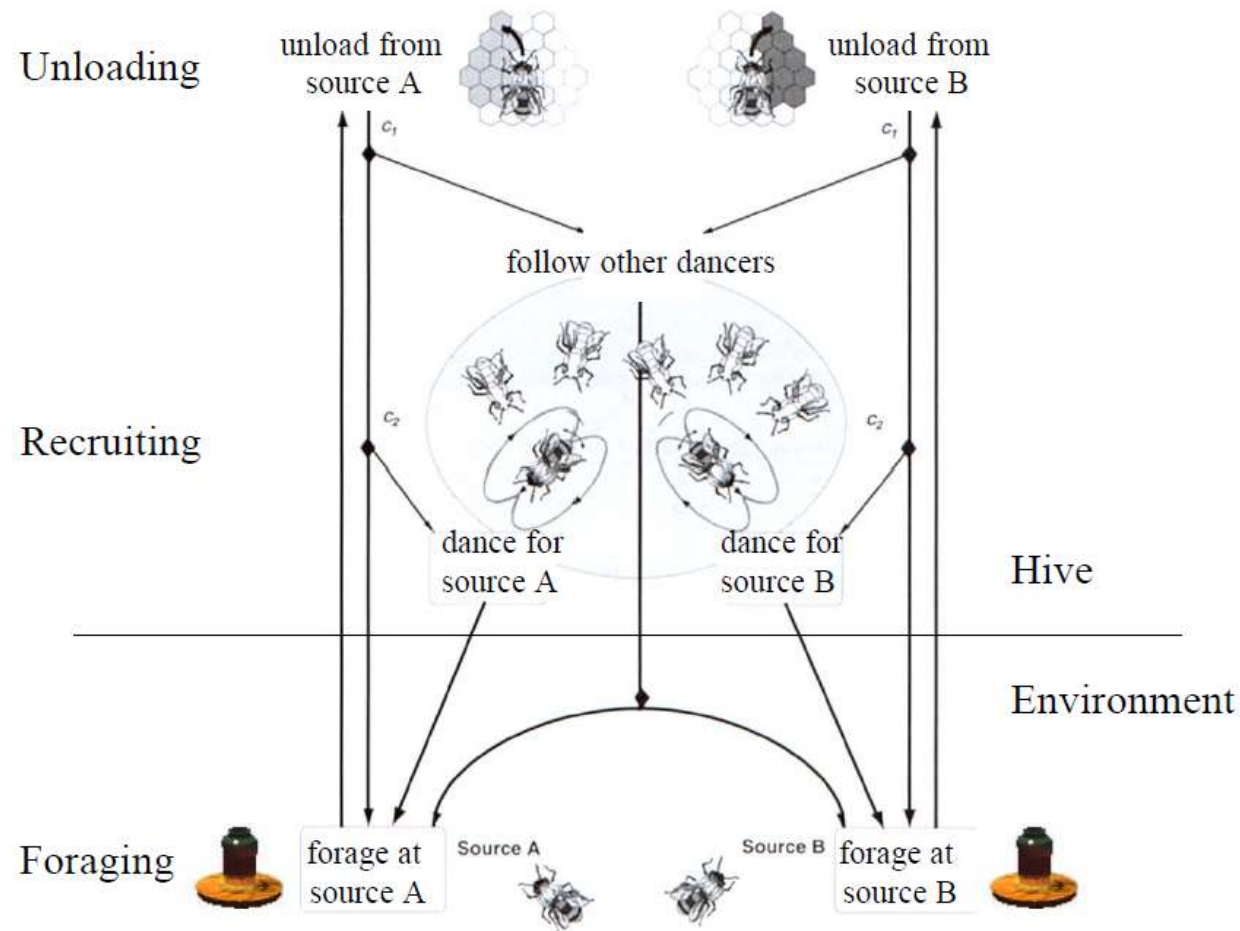
‘Self-organization is a set of dynamical mechanisms whereby structures appear at the global level of a system from interactions of its lower-level components.’

(Bonabeau et al, in Swarm Intelligence, 1999)

Self-Organization

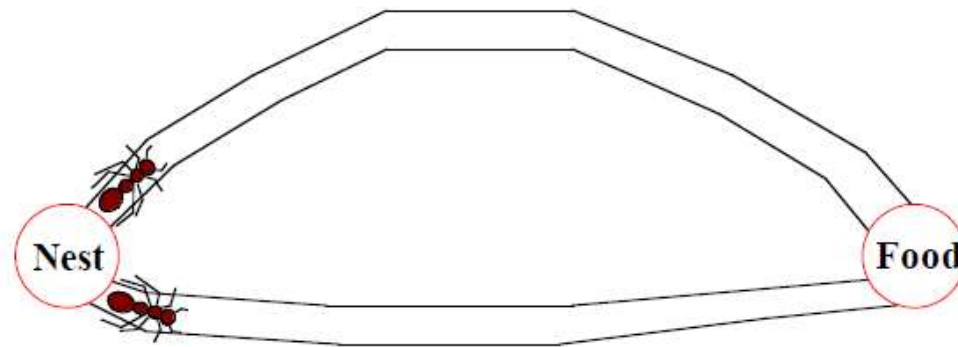
- positive feedback (amplification)
- negative feedback (for counter-balance and stabilization)
- amplification of fluctuations (randomness, errors, random walks)
- multiple interactions

Self-Organization



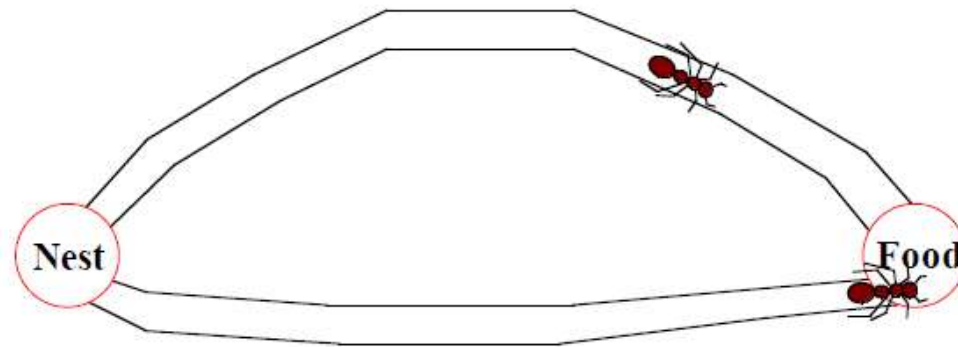
Ant Foraging

Cooperative search by pheromone trails



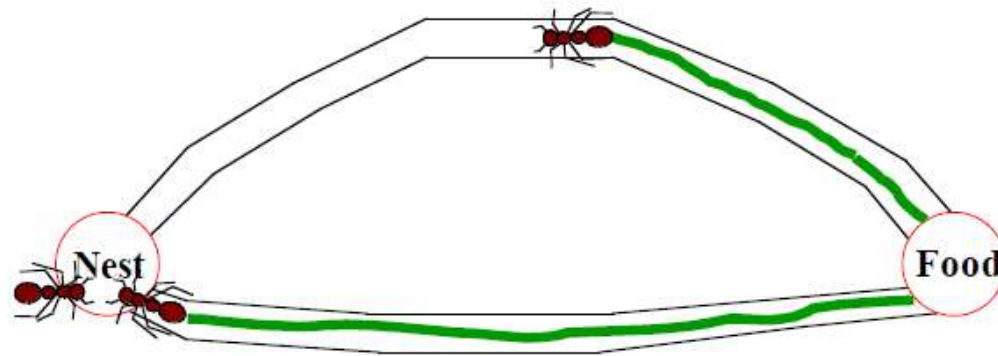
Ant Foraging

Cooperative search by pheromone trails



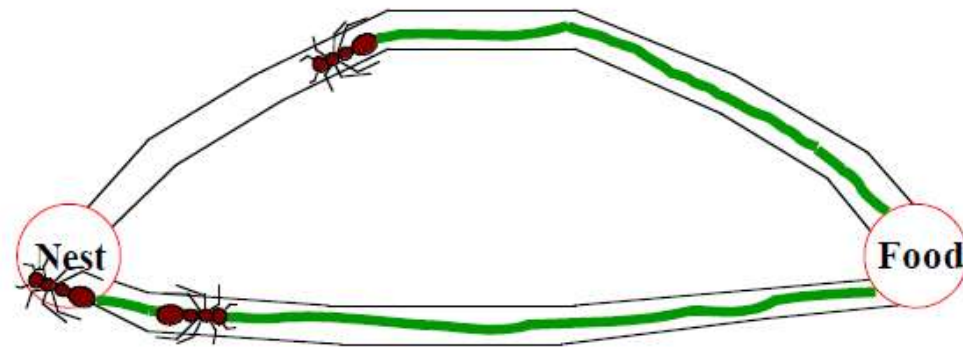
Ant Foraging

Cooperative search by pheromone trails



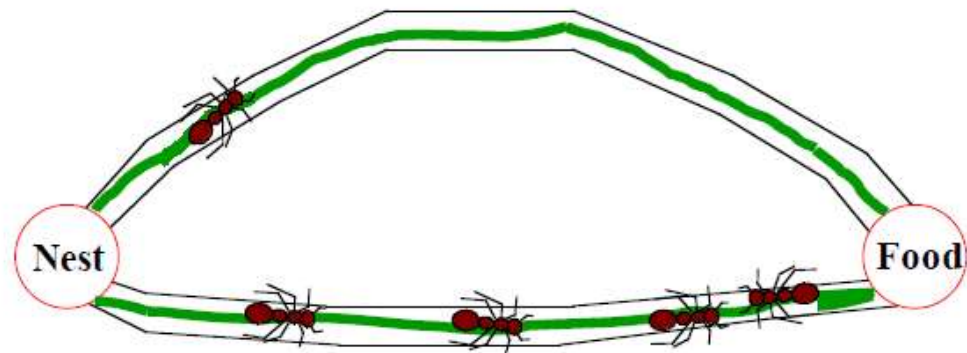
Ant Foraging

Cooperative search by pheromone trails



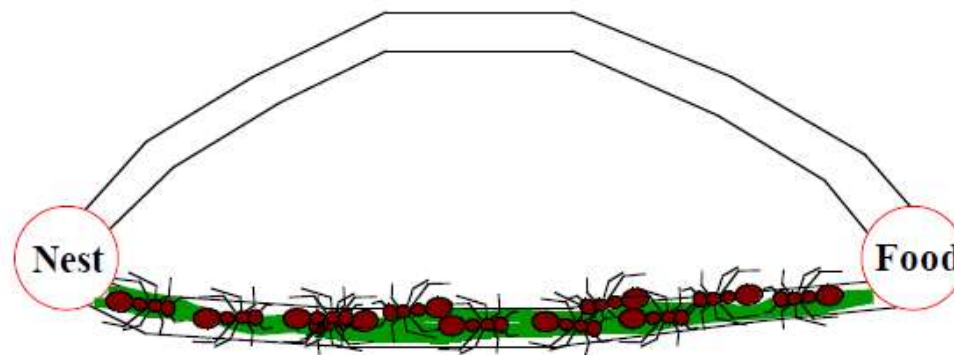
Ant Foraging

Cooperative search by pheromone trails



Ant Foraging

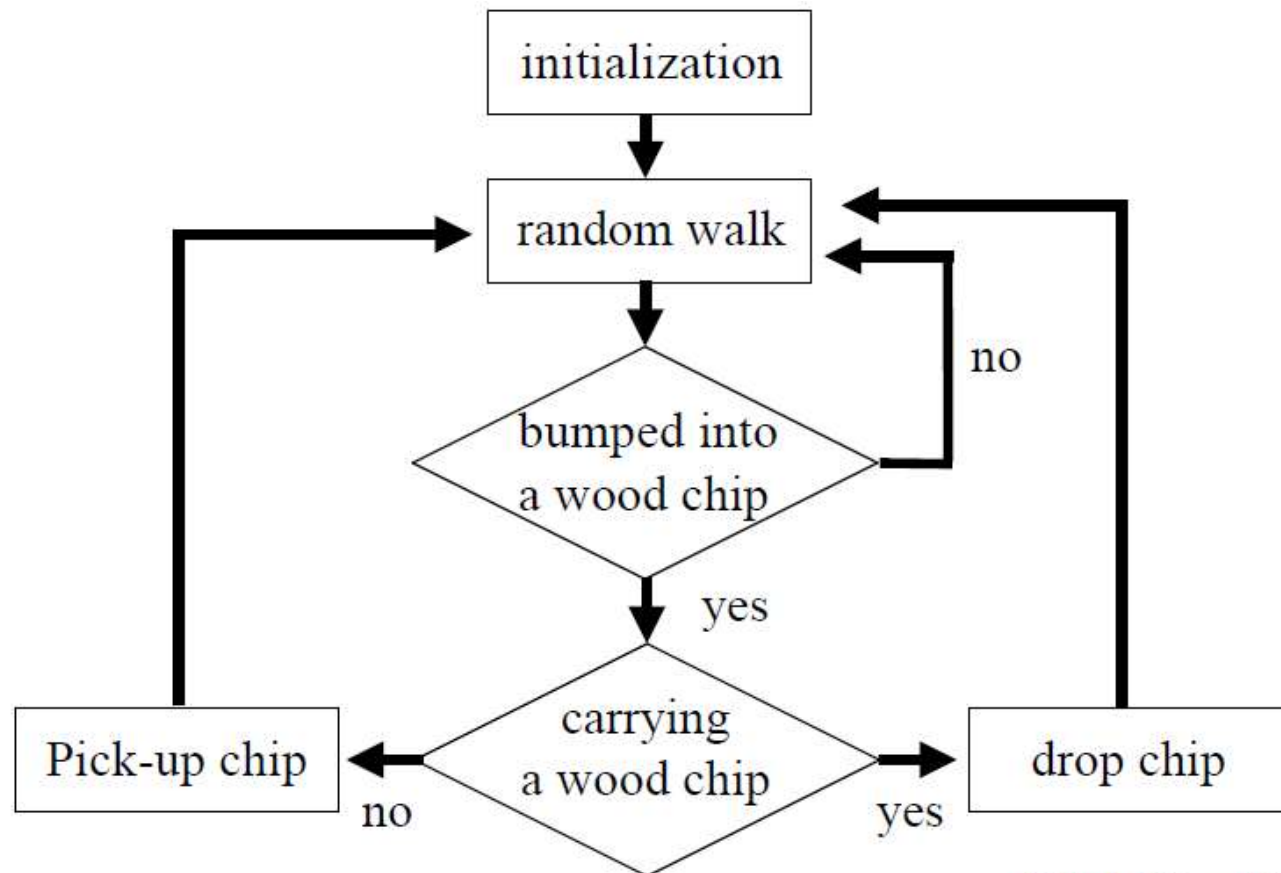
Cooperative search by pheromone trails



Characteristics of Self-Organization

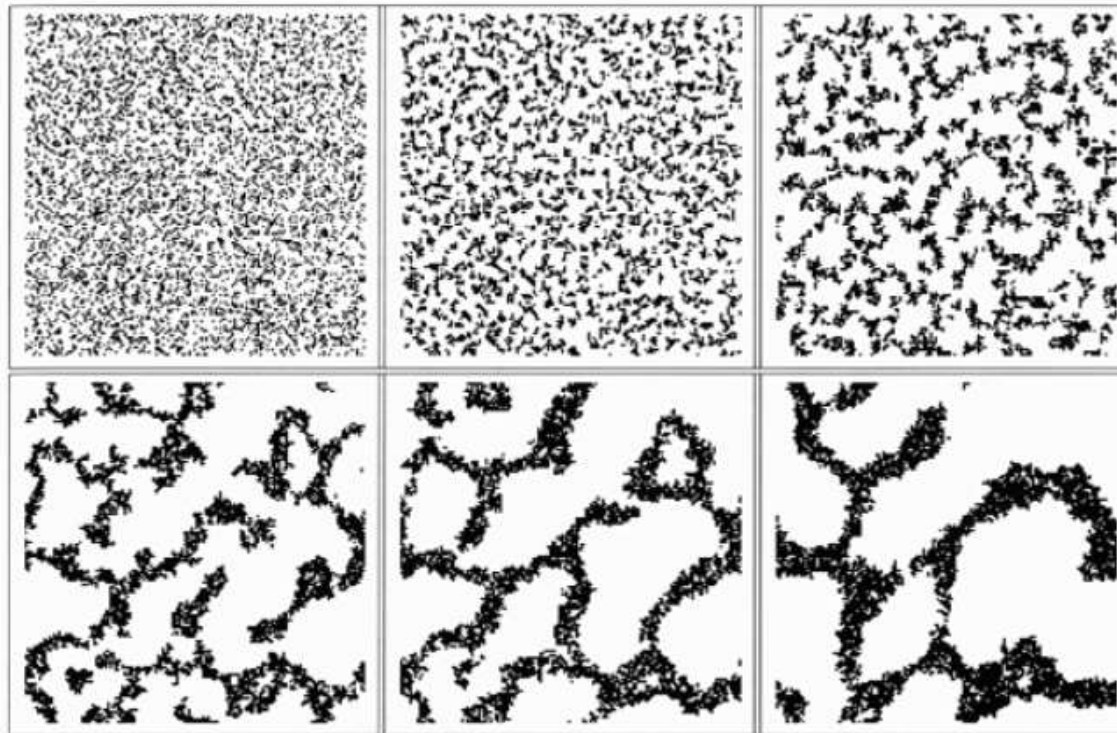
- structure emerging from a homogeneous startup state
- multistability - coexistence of many stable states
- state transitions with a dramatical change of the system behaviour

Termites Simulation



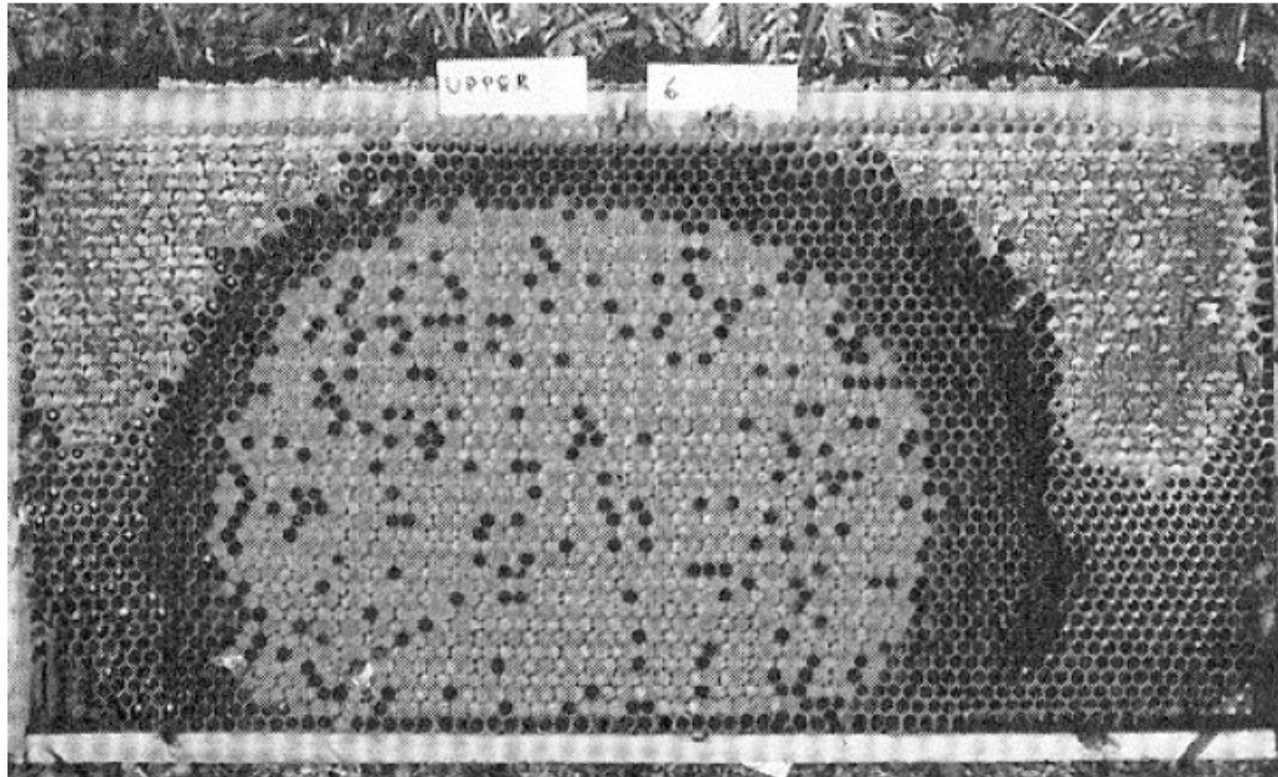
(Mitchel Resnick, 1994)

Termites Simulation



(Mitchel Resnick, 1994)

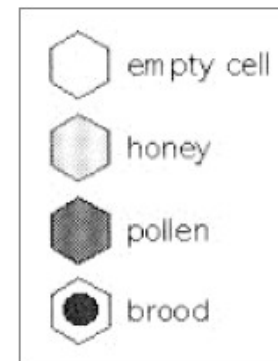
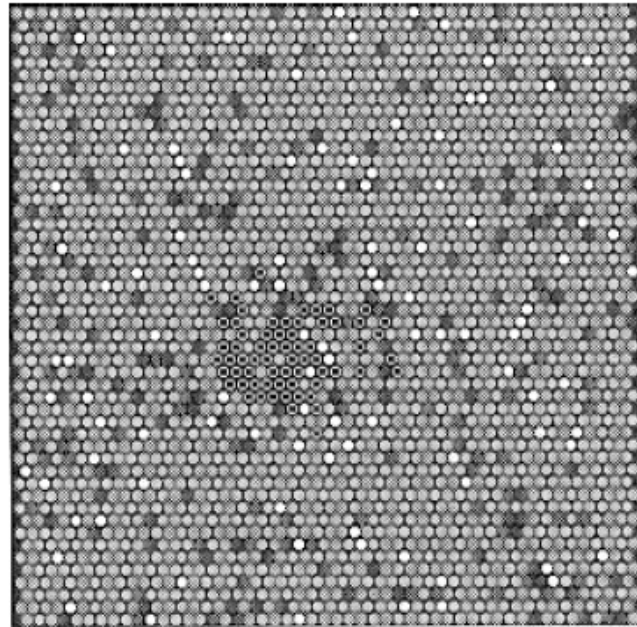
Honey Bees Nest Building



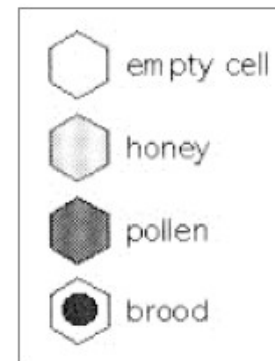
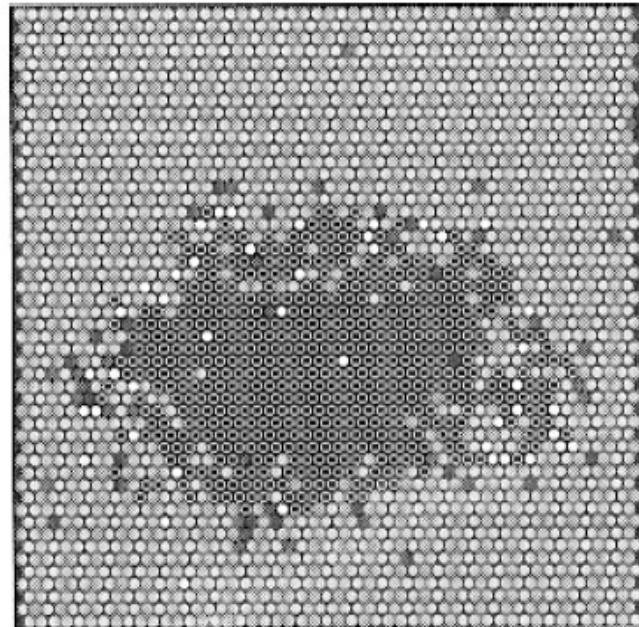
Honey Bees Nest Building

- the queen moves randomly over the combs
- eggs are more likely to be laid in the neighbourhood of brood
- honey and pollen are deposited randomly in empty cells
- four times more honey is brought to the hive than pollen
- removal ratios for honey: 0.95; pollen: 0.6
- removal of honey and pollen is proportional to the number of surrounding cells containing brood

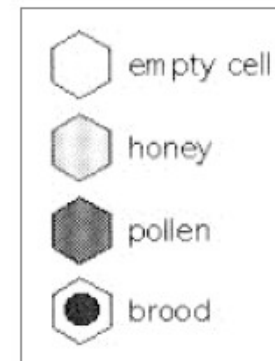
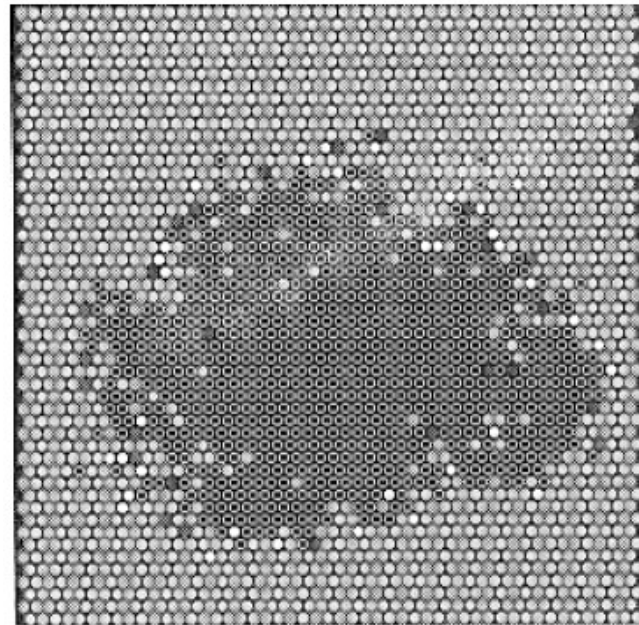
Honey Bees Nest Building



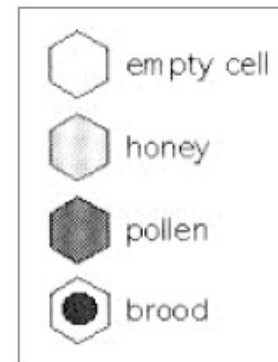
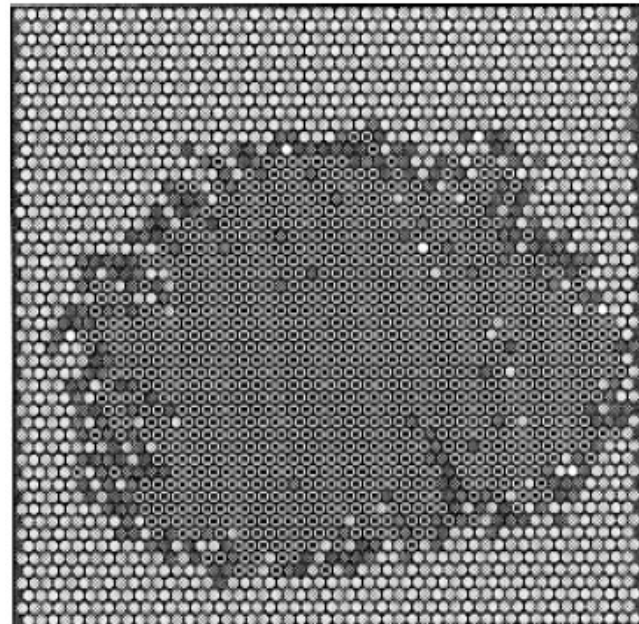
Honey Bees Nest Building



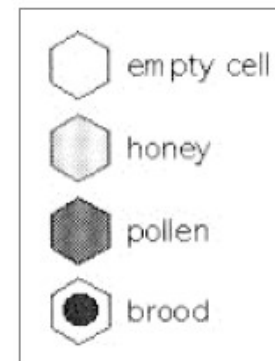
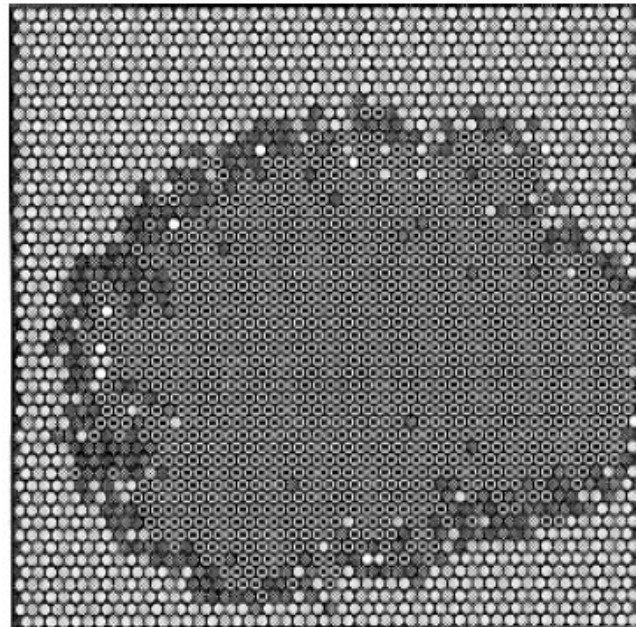
Honey Bees Nest Building



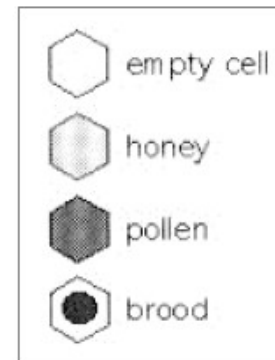
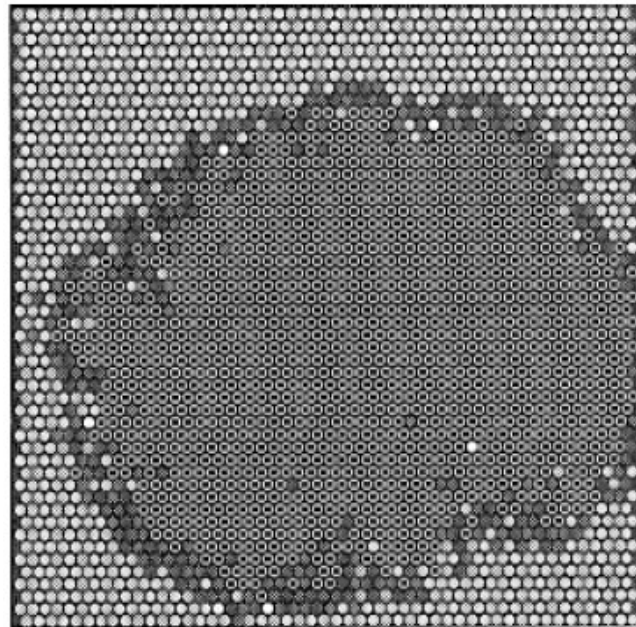
Honey Bees Nest Building



Honey Bees Nest Building



Honey Bees Nest Building



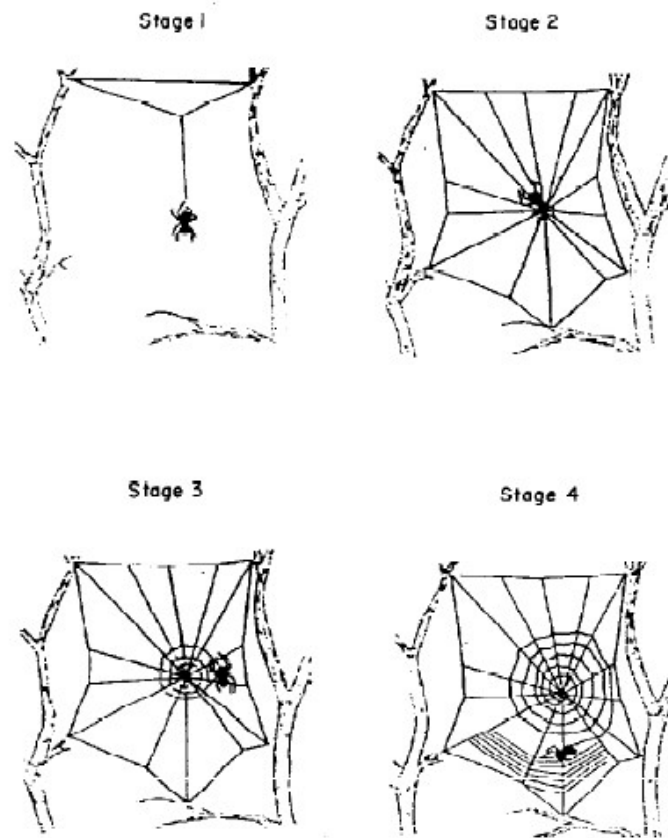
Stigmergy

Stigmergy: *stigma* (sting) + *ergon* (work)
= ‘stimulation by work’

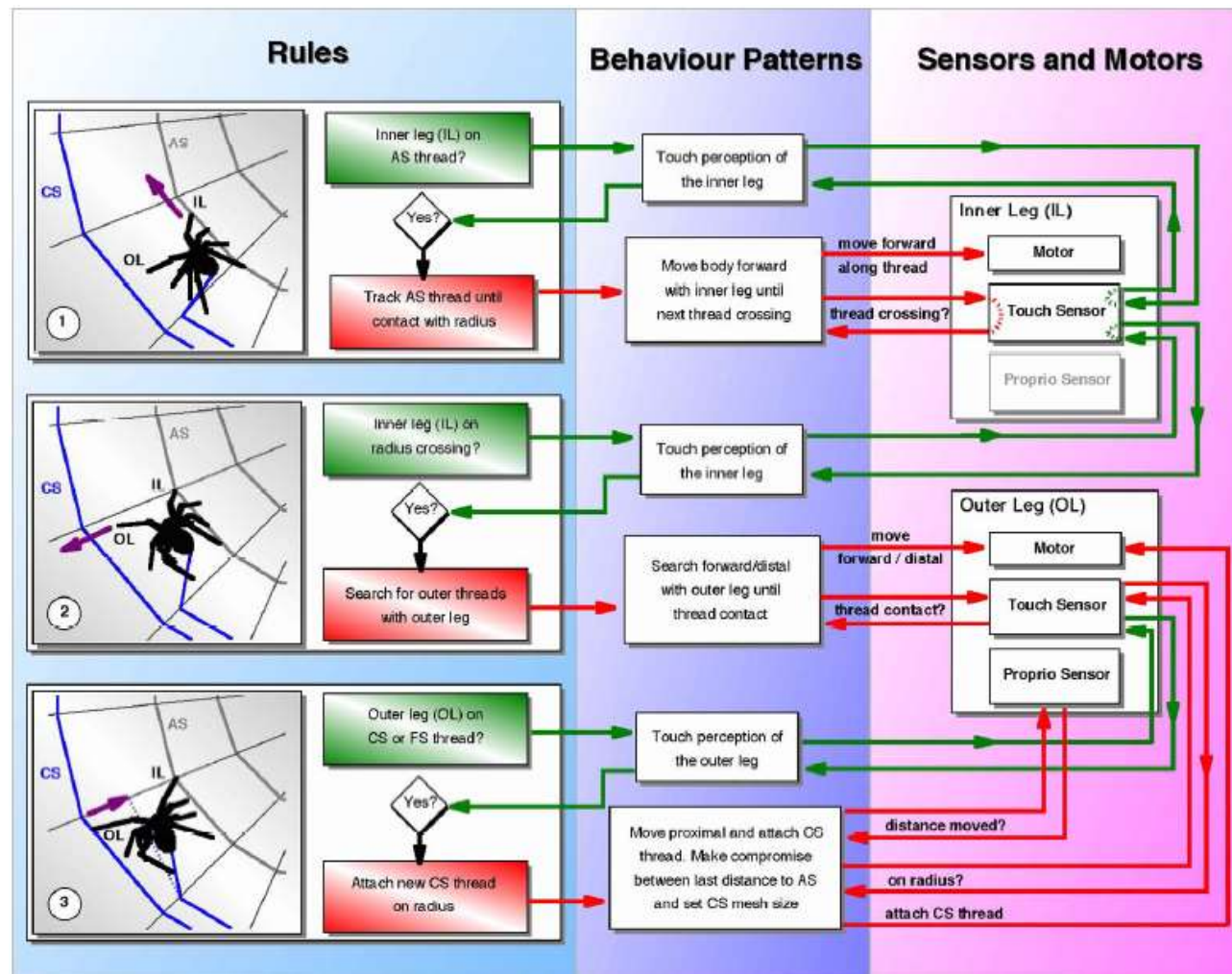
Characteristics of stigmergy

- indirect agent interaction modification of the environment
- environmental modification serves as external memory
- work can be continued by any individual
- the same, simple, behavioural rules can create different designs according to the environmental state

Stigmergy in Spiders



Stigmergy



Motivation

Motivation and methods in biologically inspired IT

- there are analogies in distributed computing and social insects
- biology has found solution to hard computational problems
- biologically inspired computing requires:
 - identification of analogies
 - computer modelling of biological mechanisms
 - adaptation of biological mechanisms for IT applications

Principles

Two principles in swarm intelligence

- self-organization is based on:
 - activity amplification by positive feedback
 - activity balancing by negative feedback
 - amplification of random fluctuations
 - multiple interactions
- stigmergy - stimulation by work - is based on:
 - work as behavioural response to the environmental state
 - an environment that serves as a work state memory
 - work that does not depend on specific agents

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).

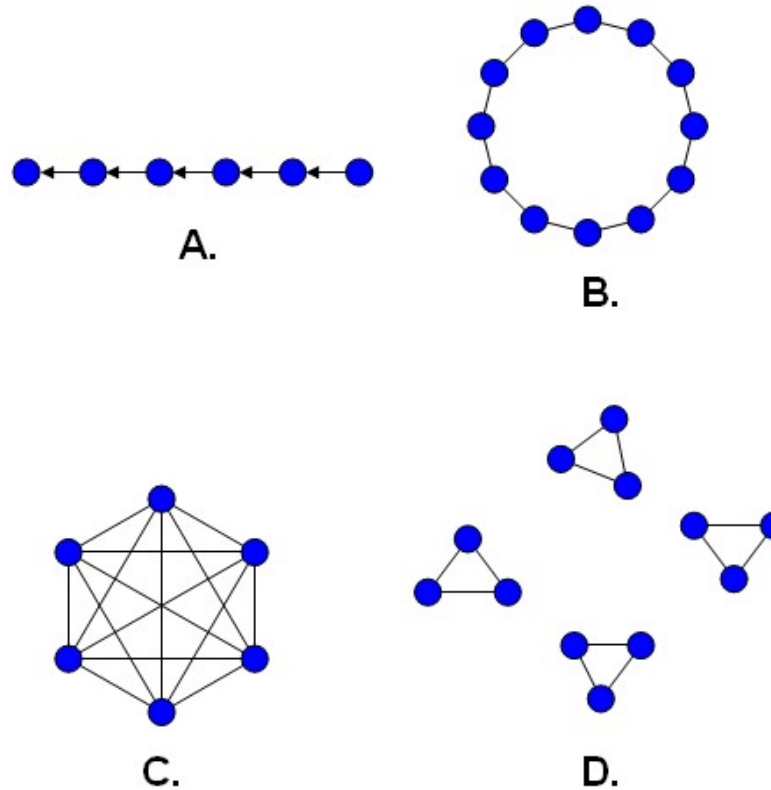
Introduction

- 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.

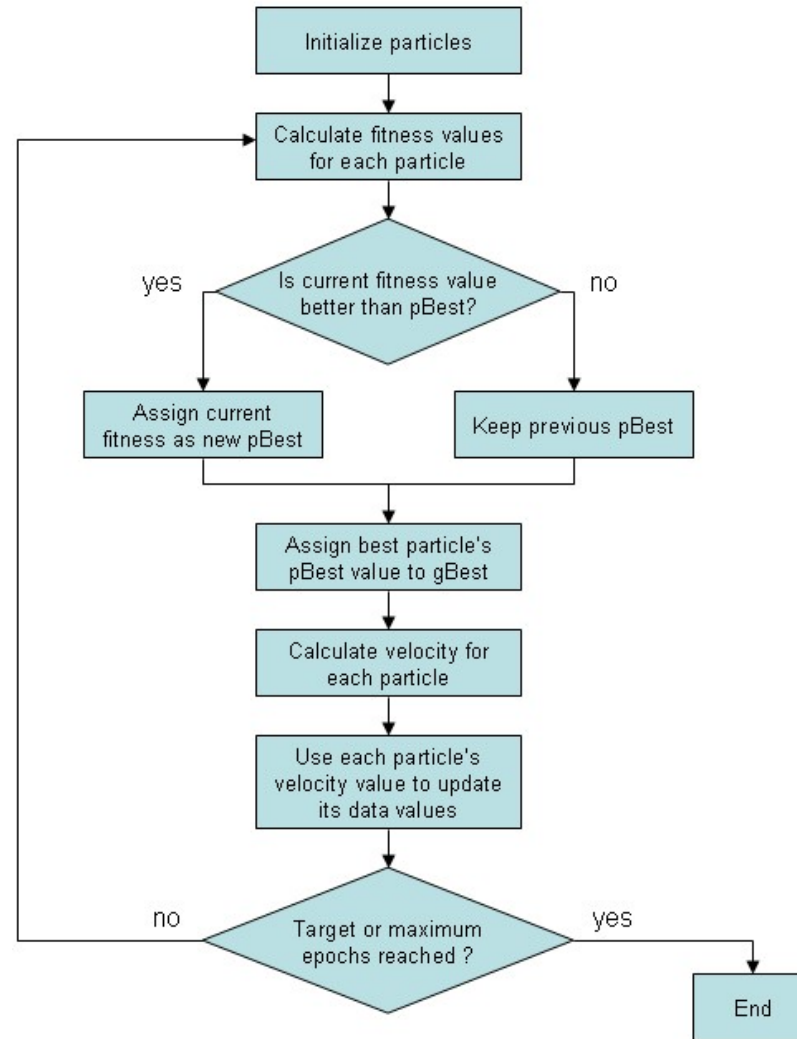
The Algorithm

- 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:

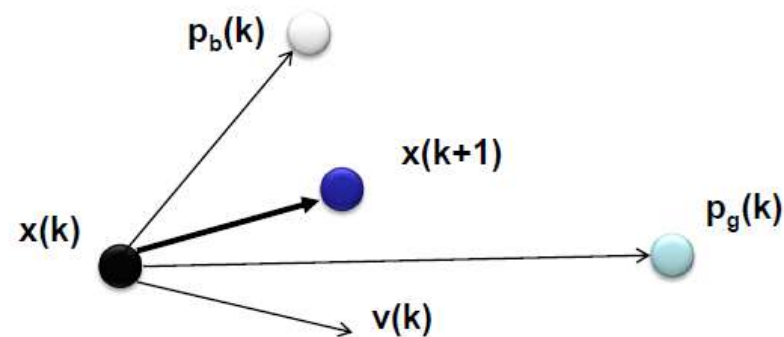
$$\mathbf{v}^{new} = a\mathbf{v}^{old} + bw_1 \times (\mathbf{x}_{my_best} - \mathbf{x}^{old}) + cw_2 \times (\mathbf{x}_{best} - \mathbf{x}^{old})$$

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 $[0,1]$.

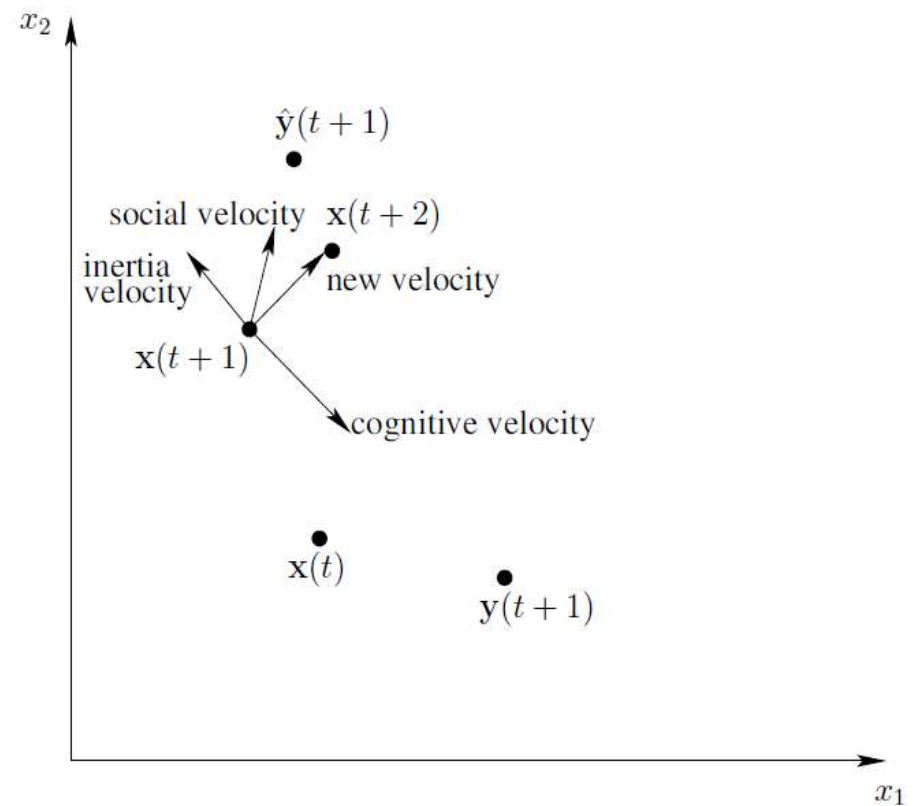
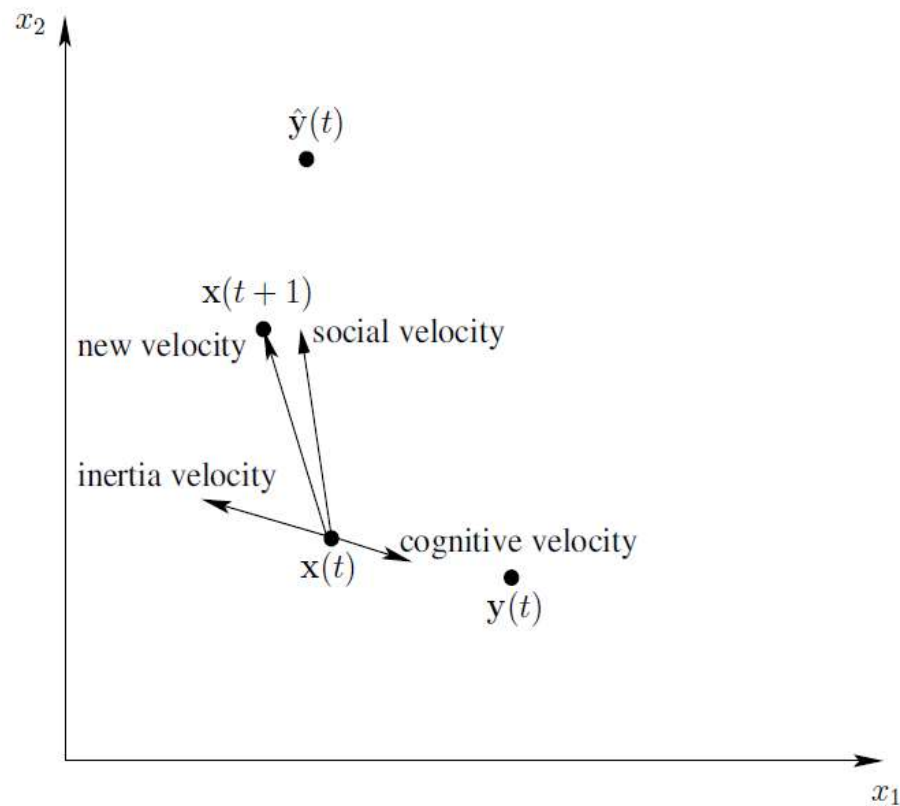
The Algorithm

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

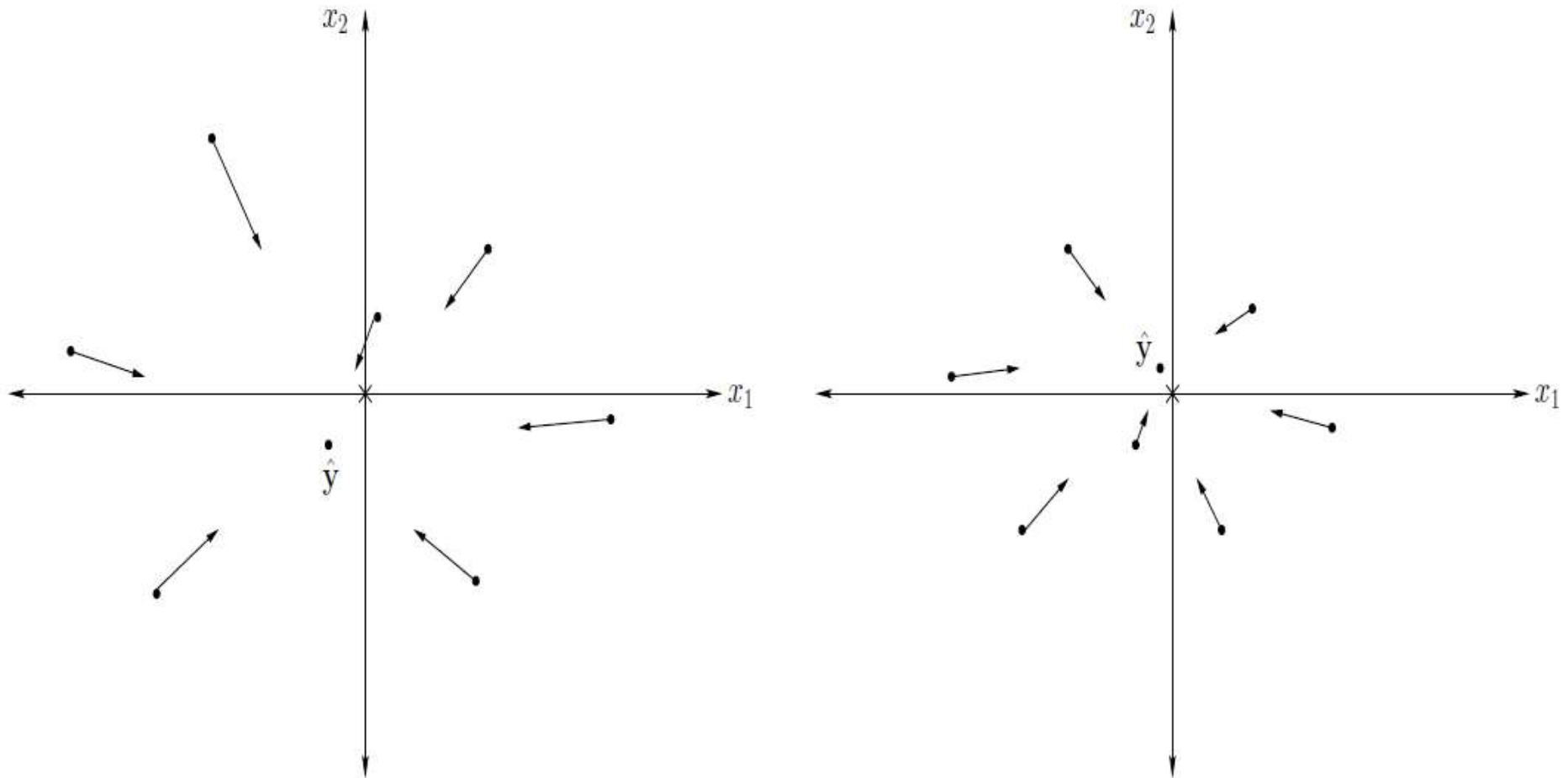
$$\mathbf{x}^{new} = \mathbf{x}^{old} + \mathbf{v}^{new}$$



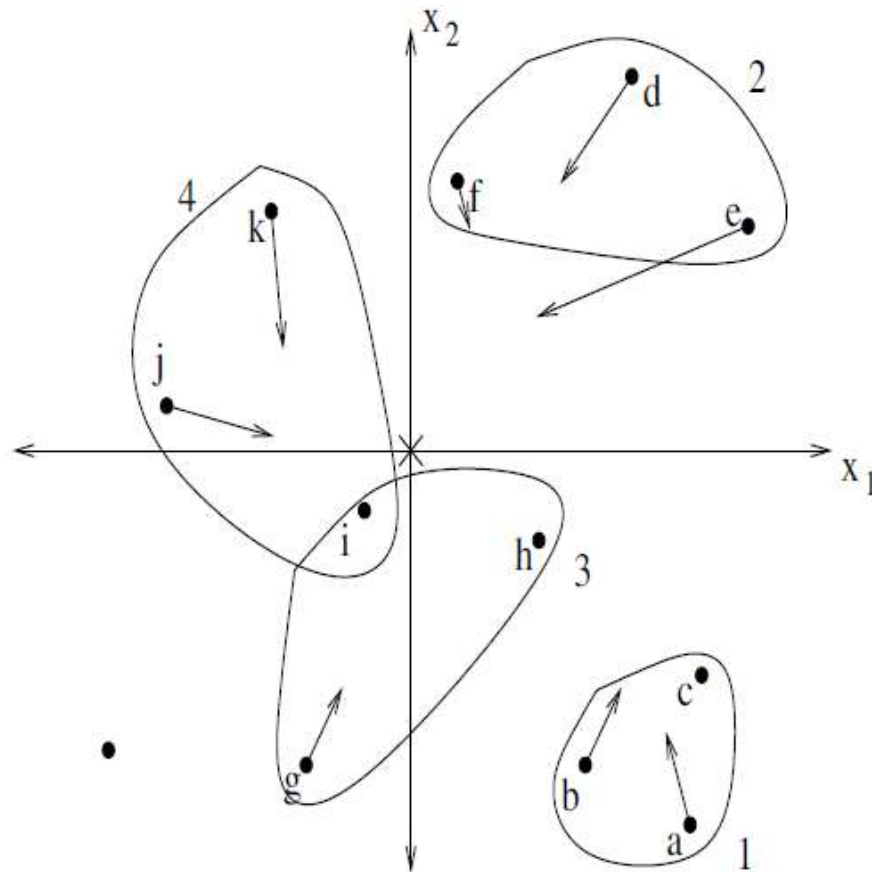
The Algorithm



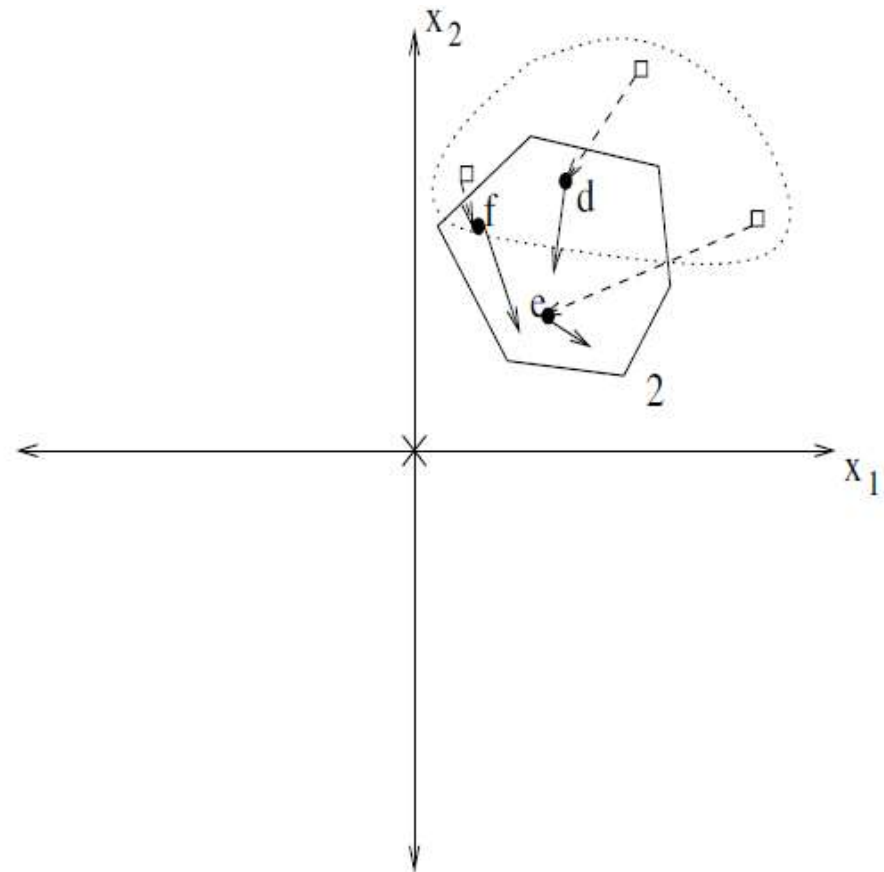
The Algorithm



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., $\sim N(0.72, \sigma)$ 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/>

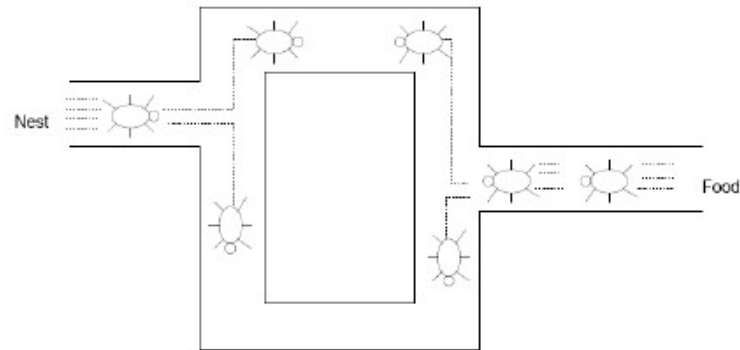
<https://nathanrooy.github.io/posts/2016-08-17/simple-particle-swarm-optimization-with-python/>

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)



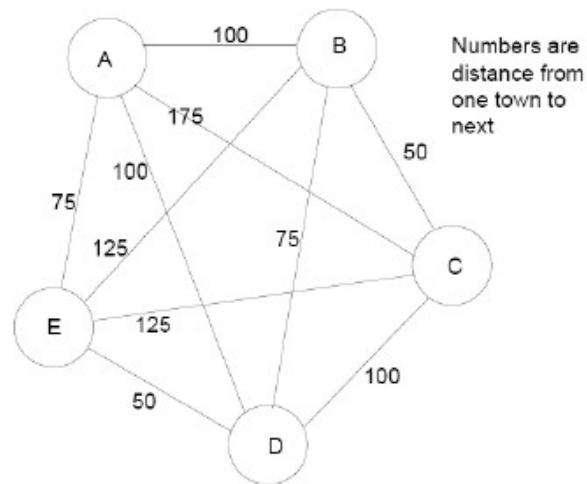
Pierre-Paul Grassé
(1959)

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

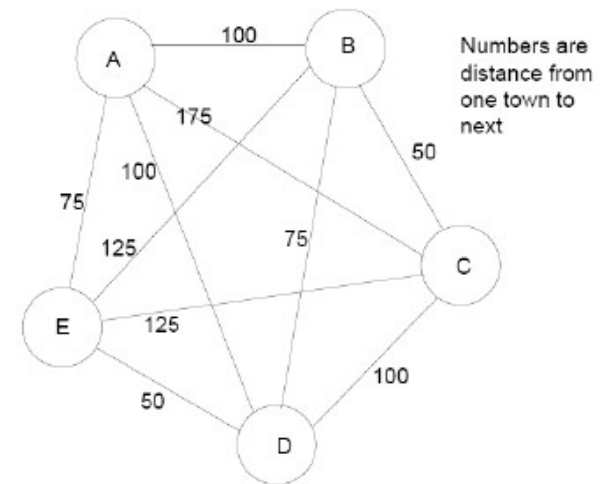
Example in TSP



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

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

- 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 k th 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 $\mathcal{P} = (\mathcal{S}, f, \Omega)$.

InitializePheromoneValues(\mathcal{T})

$s_{bs} \leftarrow \text{NULL}$

init best-so-far solution

while termination conditions not met **do**

$\mathcal{S}_{iter} \leftarrow \emptyset$

set of valid solutions

for $j = 1, \dots, n_a$ **do**

loop over ants

$s \leftarrow \text{ConstructSolution}(\mathcal{T})$

if s is a valid solution **then**

$s \leftarrow \text{LocalSearch}(s)$ {optional}

if $(f(s) < f(s_{bs}))$ or $(s_{bs} = \text{NULL})$ **then** $s_{bs} \leftarrow s$

update best-so-far

$\mathcal{S}_{iter} \leftarrow \mathcal{S}_{iter} \cup \{s\}$

store valid solutions

end if

end for

 ApplyPheromoneUpdate($\mathcal{T}, \mathcal{S}_{iter}, s_{bs}$)

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

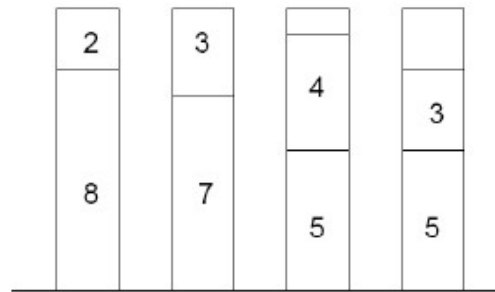
Problem	ACS (avge)	SA (avge)	EN (avge)	SOM (avge)
50-city set 1	5.88	5.88	5.98	6.06
50-city set 2	6.05	6.01	6.03	6.25
50-city set 3	5.58	5.65	5.70	5.83
50-city set 4	5.74	5.81	5.86	5.87
50-city set 5	6.18	6.33	6.49	6.70

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.

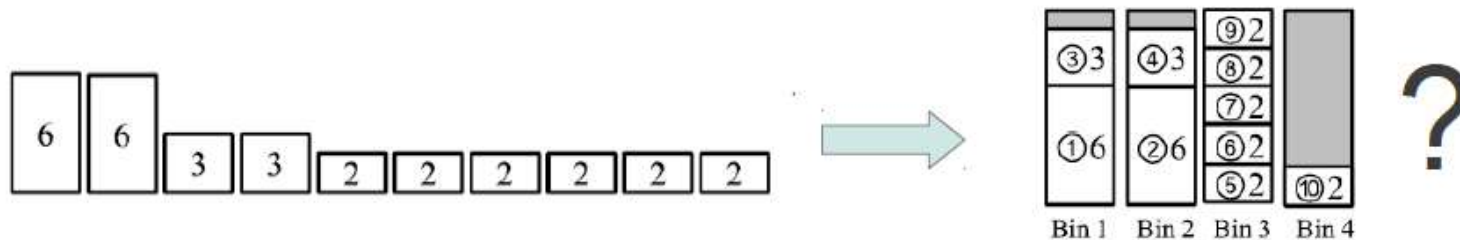
Bin Packing Problems



- 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 recognise 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:

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

- 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)

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

- 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
- $\tau_{\text{min}} = 0.001$ • $\rho = 0.75$
- 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

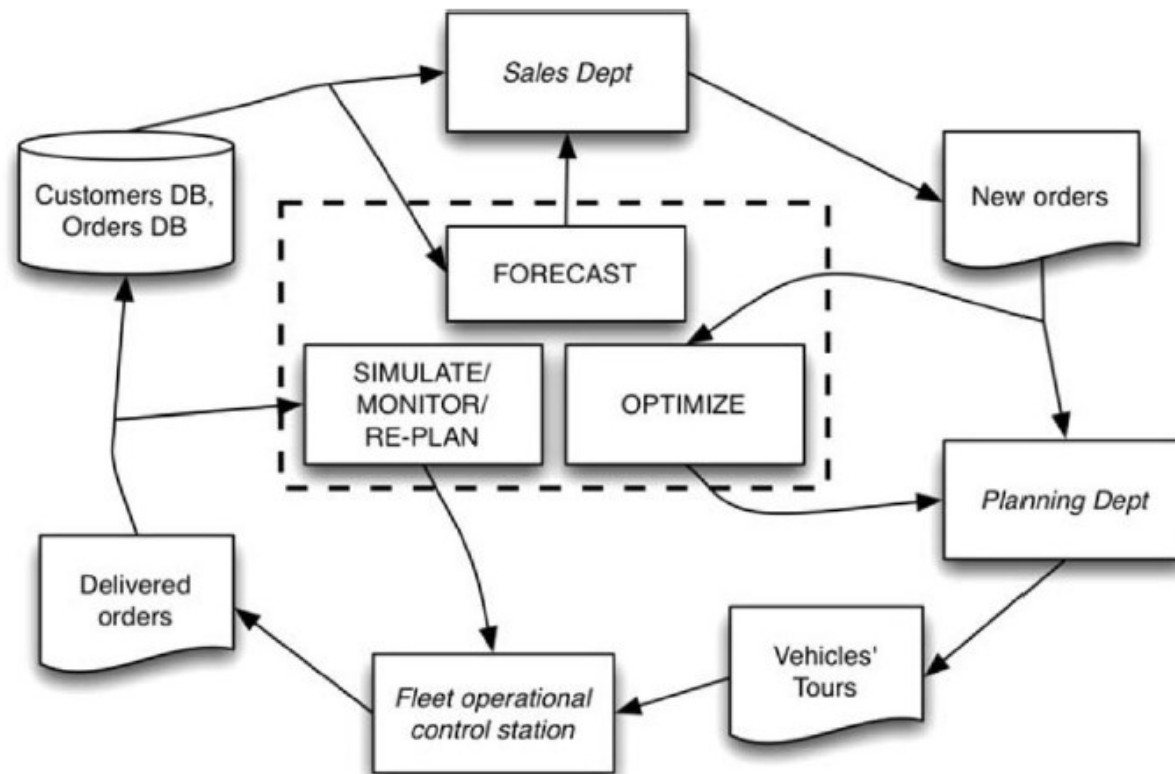
What we need to set up an ACO

- *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

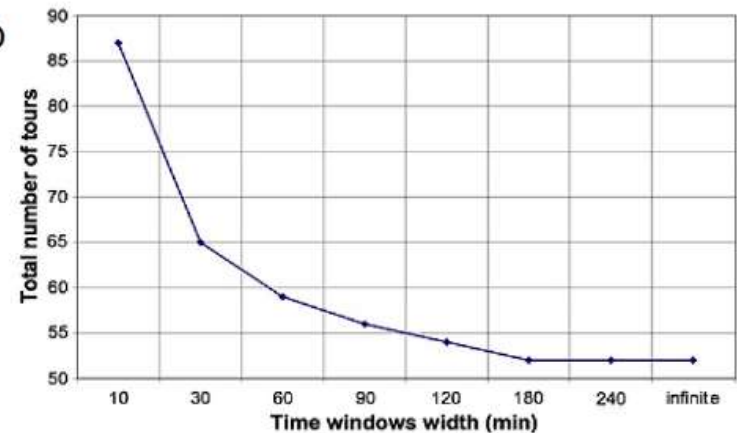
- Best ant laying pheromone (global-best ant or, in some versions of ACO, iteration-best ant) encourage ants to follow the best tour or to search in the neighbourhood of this tour (make sure that $\tau_{\min} > 0$).
- Local updating (the ants lay pheromone as they go along without waiting till end of 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%

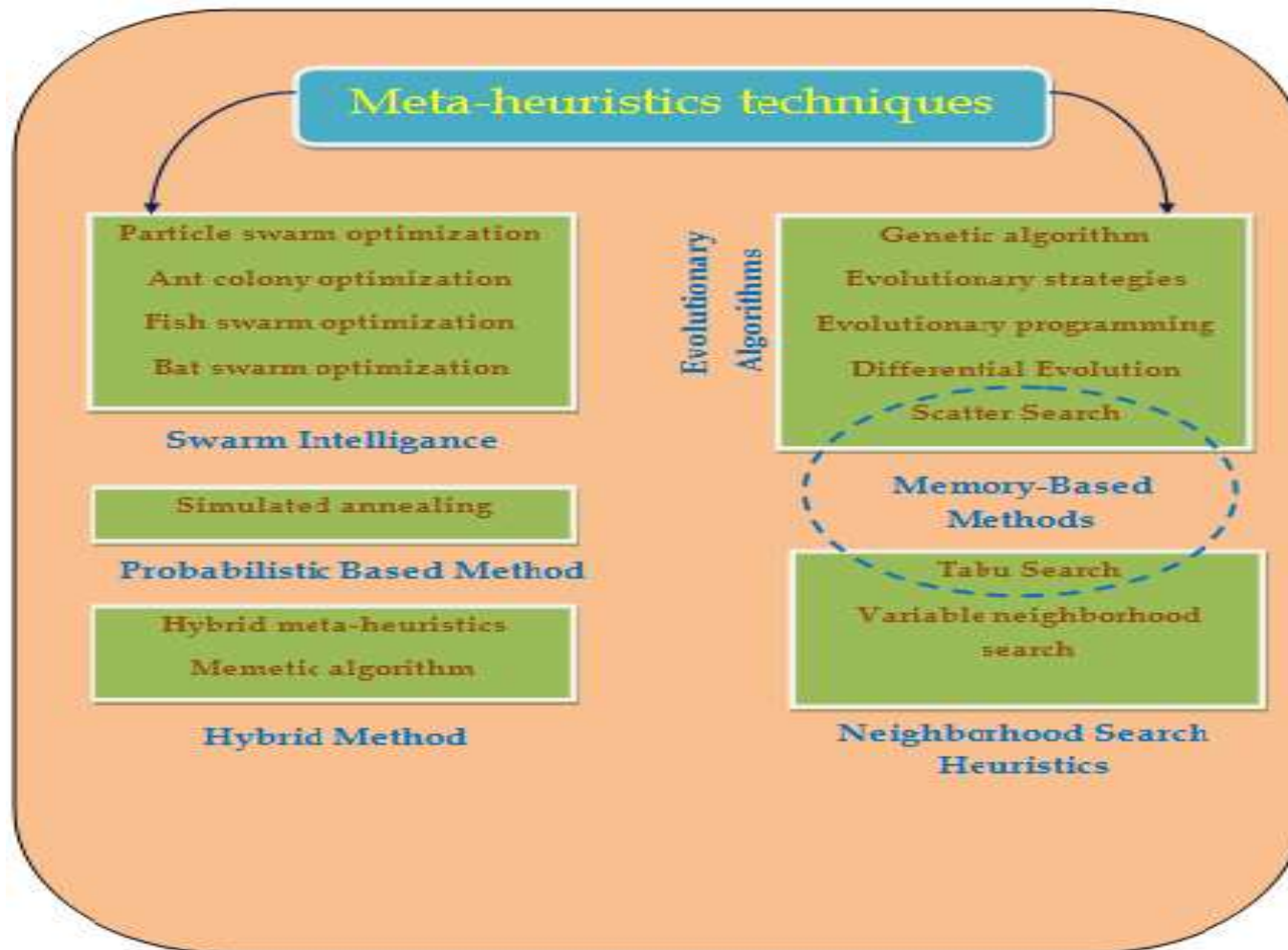
The ACO Algorithm

<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 x_i ($i = 1, 2, \dots, SN$) is a D-dimensional vector.
- After initialization, the population of the positions (solutions) is subjected to repeated cycles, $C = 1, 2, \dots, MCN$, 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 $k \in \{1, 2, \dots, SN\}$ and $j \in \{1, 2, \dots, D\}$ 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.

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

Algorithm

Algorithm 1 Artificial Bee Colony algorithm

- 1: Generate the initial population x_i randomly, $i = 1, \dots, NS$. ▷ Initialization
 - 2: Evaluate the fitness function fit_i of all solutions in the population.
 - 3: Keep the best solution x_{best} in the population. ▷ Memorize the best solution
 - 4: Set $cycle=1$.
 - 5: repeat
 - 6: Generate new solutions v_i from old solutions x_i where $v_{ij} = x_{ij} + \phi_{ij}(x_{ij} - x_{kj})$,
 $\phi_{ij} \in [-1, 1]$, $k \in \{1, 2, \dots, NS\}$, $j \in \{1, 2, \dots, n\}$, and $i \neq k$. ▷ Employed bees
 - 7: Evaluate the fitness function fit_i of all new solutions in the population.
 - 8: Keep the best solution between current and candidate solutions. ▷ Greedy
 selection
 - 9: Calculate the probability P_i , for the solutions x_i where $P_i = fit_i / \sum_{j=1}^{NS} fit_j$.
 - 10: Generate the new solutions v_i from the solutions selecting depending on its P_i .
 ▷ Onlookers bees
 - 11: Evaluate the fitness function fit_i of all new solutions in the population.
 - 12: Keep the best solution between current and candidate solutions. ▷ Greedy
 selection
 - 13: Determine the abandoned solution if exist, replace it with a new randomly
 solution x_i . ▷ Scout bee
 - 14: Keep the best solution x_{best} found so far in the population.
 - 15: $cycle = cycle + 1$
 - 16: 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:

$$\text{Minimize } f(x) = x_1^2 + x_2^2 \quad -5 \leq x_1, x_2 \leq 5$$

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 $(-5, 5)$.

$x =$ 1.4112 -2.5644
 0.4756 1.4338
 -0.1824 -1.0323

$f(x)$ values are: 8.5678
 2.2820
 1.0990

Example

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

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

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

with this formula, produce a new solution.

k=1 k is a random selected index.

j=0 j is a random selected index.

Example

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

$u_0 = 2.1644 \quad -2.5644$

Calculate $f(u_0)$ and the fitness of u_0 .

$f(u_0) = 11.2610$ and the fitness value is 0.0816 .

Apply greedy selection between x_0 and u_0

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

Example

2nd employed bee

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

with this formula produce a new solution.

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

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

$u_1 = 0.4756 \quad 1.6217$

Calculate $f(u_1)$ and the fitness of u_1 .

$f(u_1) = 2.8560$ and the fitness value is 0.2593.

Apply greedy selection between x_1 and u_1

$0.2593 < 0.3047$, the solution 1 couldn't be improved, increase its trial counter.

Example

3rd employed bee

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

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.

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

$$u_2 = -0.0754 \quad -1.0323$$

Calculate $f(u_2)$ and the fitness of u_2 .

$f(u_2) = 1.0714$ and the fitness value is 0.4828.

Apply greedy selection between x_2 and u_2 .

$0.4828 > 0.4764$, the solution 2 was improved, set its trial counter as 0 and replace the solution x_2 with u_2 .

Example

$x =$

1.4112	-2.5644
0.4756	1.4338
-0.0754	-1.0323

$f(x)$ 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;

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

$p = 0.1172$
0.3416
0.5412

Example

Onlooker bees phase

Produce new solutions u_i for the onlookers from the solutions x_i selected depending on p_i and evaluate them.

1st onlooker bee

$i=2$

$u_2 = -0.0754 \quad -2.2520$

Calculate $f(u_2)$ and the fitness of u_2 .

$f(u_2) = 5.0772$ and the fitness value is 0.1645.

Apply greedy selection between x_2 and u_2

$0.1645 < 0.4828$, the solution 2 couldn't be improved, increase its trial counter.

Example

2nd onlooker bee

$i=1$

$u_1 = 0.1722 \quad 1.4338$

Calculate $f(u_1)$ and the fitness of u_1 .

$f(u_1) = 2.0855$ and the fitness value is 0.3241 .

Apply greedy selection between x_1 and u_1

$0.3241 > 0.3047$, the solution 1 was improved, set its trial counter as 0 and replace the solution x_1 with u_1 .

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

$i=2$

$u_2 = 0.0348 \quad -1.0323$

Calculate $f(u_2)$ and the fitness of u_2 .

$f(u_2) = 1.0669$ and the fitness value is 0.4838.

Apply greedy selection between x_2 and u_2

$0.4838 > 0.4828$, the solution 2 was improved, set its trial counter as 0 and replace the solution x_2 with u_2 .

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

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 $p \in [0,1]$.
- 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.

Lévy 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

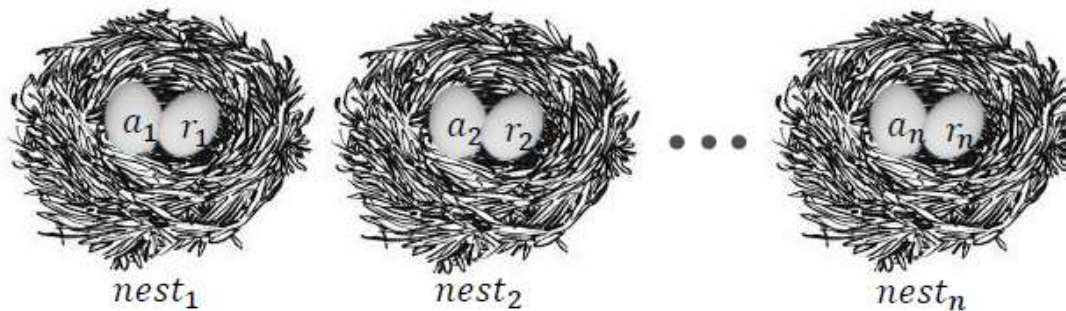
Algorithm 1 Cuckoo search algorithm

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

Steps

The following steps describe the main concepts of Cuckoo search algorithm

Step1. Generate initial population of n host nests.



(a_i, r_i) : 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

$ak' = ak + \text{Randomwalk}$ (Lèvy flight) ak

$rk' = rk + \text{Randomwalk}$ (Lèvy flight) rk



Steps

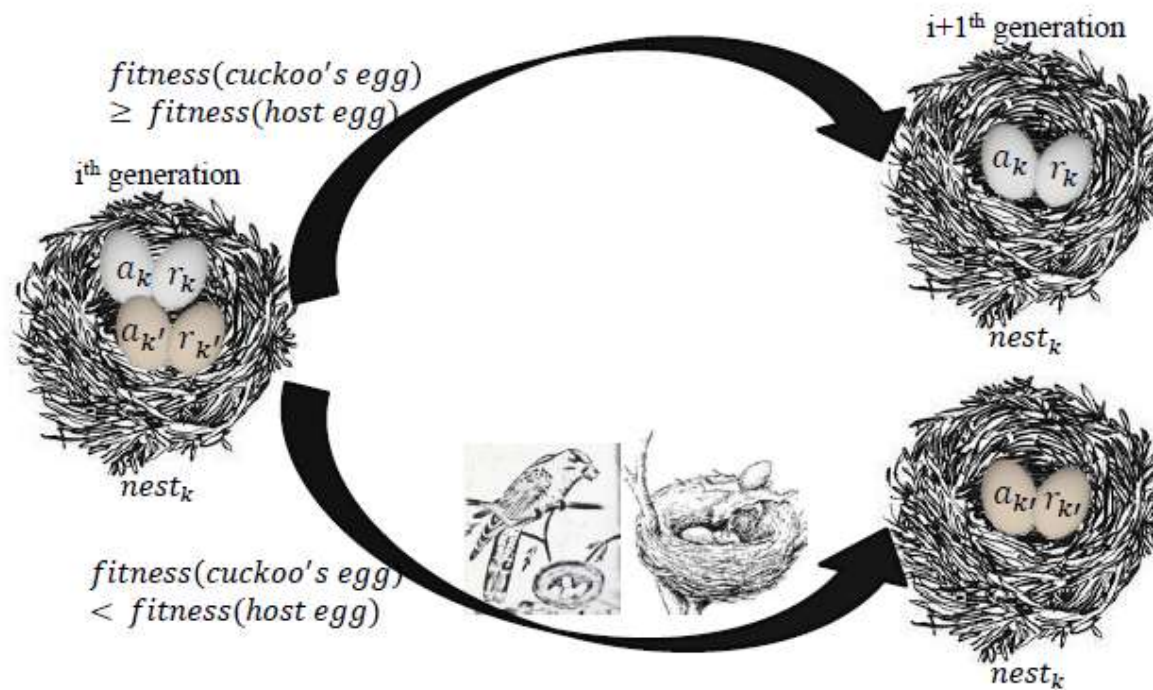
Step3. Compare the fitness of cuckoo's egg with the fitness of the host egg.

- Root Mean Square Error (RMSE)



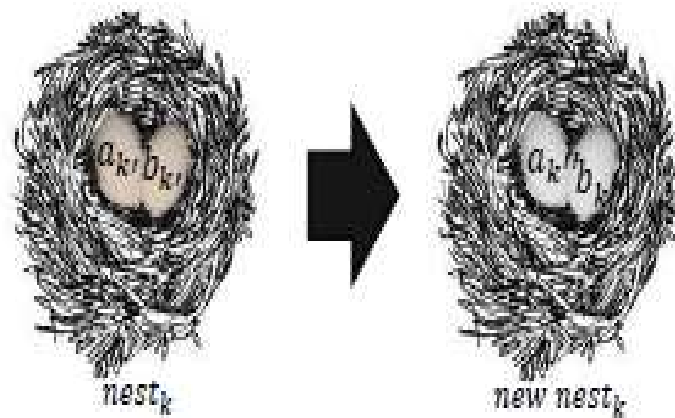
Steps

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

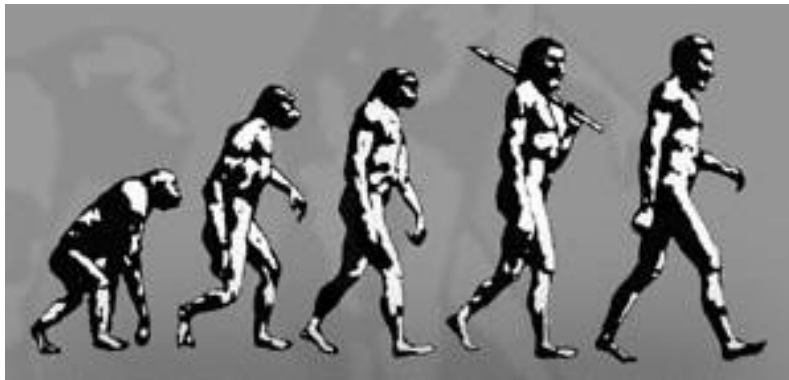
- Engineering optimization problems
- NP hard combinatorial optimization problems
- Data fusion in wireless sensor networks
- Nanoelectronic technology based operation-amplifier (OP-AMP)
- Train neural network
- Manufacturing scheduling
- Nurse scheduling problem

Evolutionary Computation

Adopted from Madhu, Natraj, Bhavish, Sanjay & Antoine CORNUÉJOLS - Christine Martin

Introduction

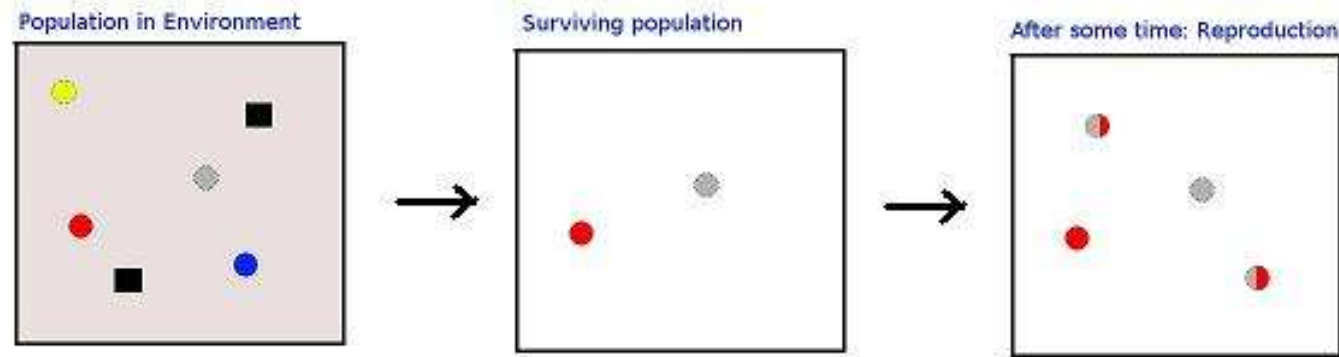
- Evolution is the change in the inherited traits of a population from one generation to the next.



- 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, $f(x_i)$, for all x_i belonging to $P(t)$;

 Perform cross-over to produce offspring;

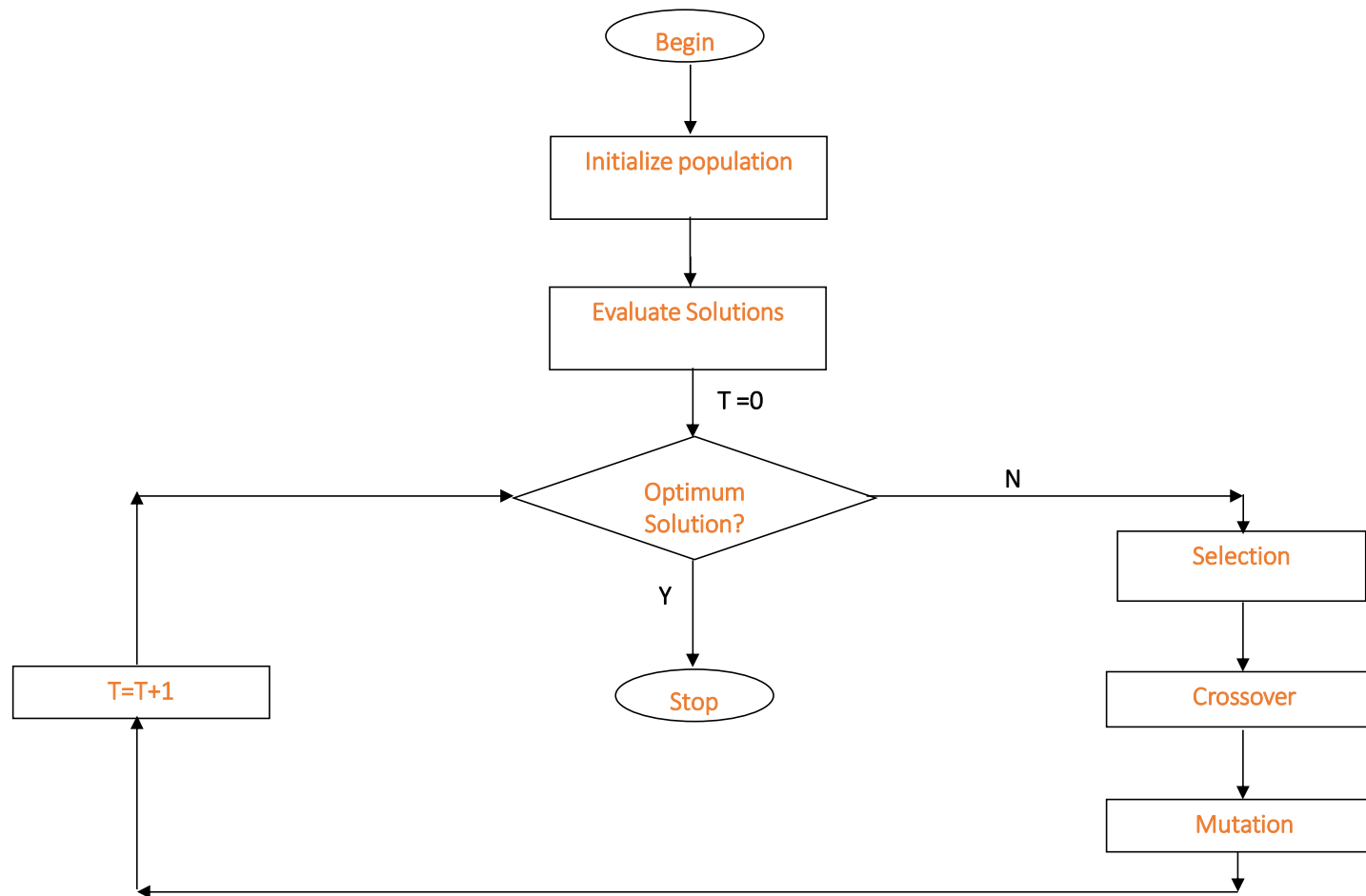
 Perform mutation on offspring;

 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



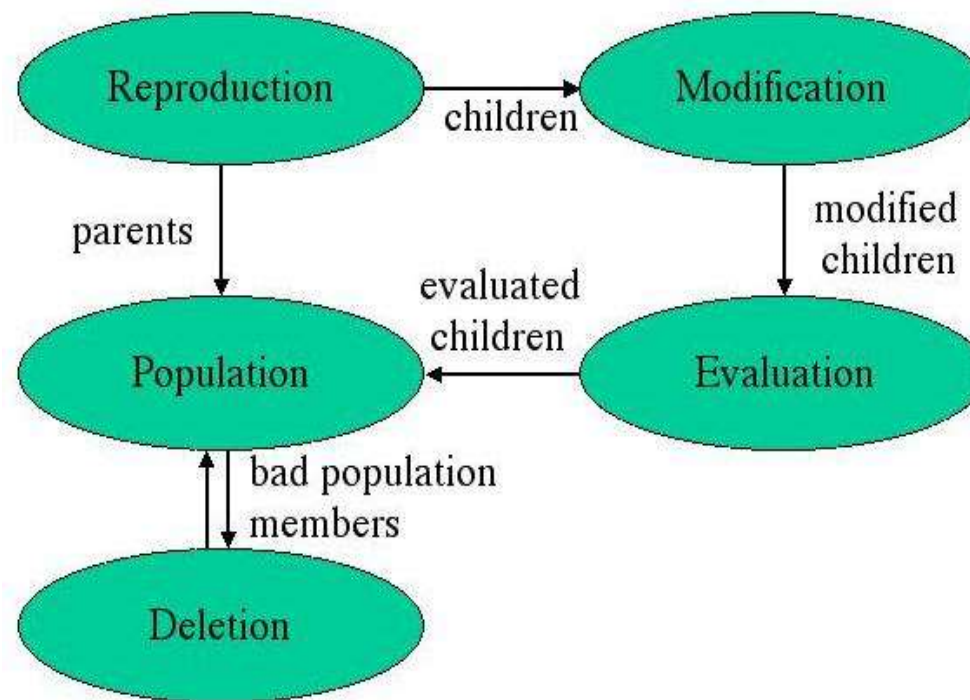
Genetic Algorithms

- 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

<i>Natural Evolution</i>	<i>Evolutionary Computation</i>
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



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 $[\sim 50 - \sim 1000]$

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:

$$P(x_i) = f(x_i) / \sum f(x_j) \quad \text{for all } j$$

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

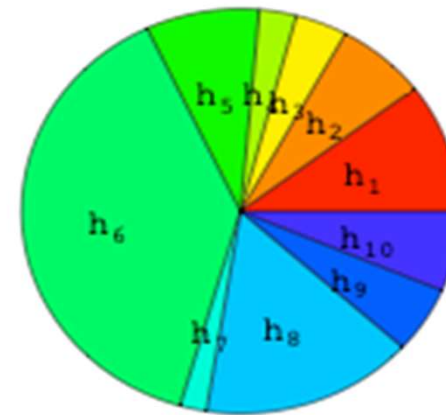
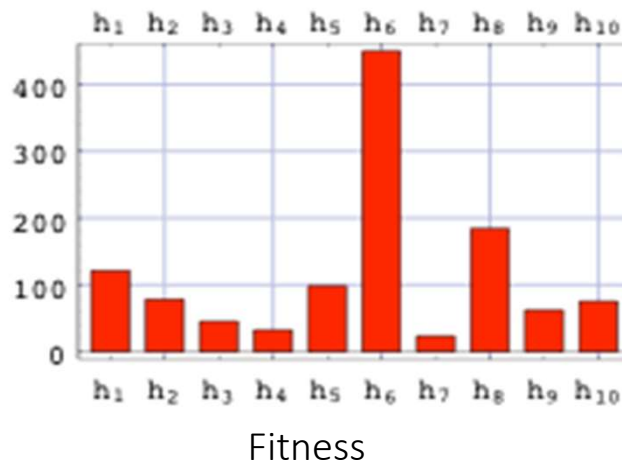
$$P(x_i) = (1/\beta)(1 - e^{r(x_i)})$$

Wheel Selection

- Let $i = 1$, where i denotes chromosome index;
- Calculate $P(x_i)$ using proportional selection;
- $sum = P(x_i)$;
- choose $r \sim U(0,1)$;
 - while $sum < r$ do
 - $i = i + 1$; i.e. next chromosome
 - $sum = sum + P(x_i)$;
 - end
 - return x_i as one of the selected parent;
 - repeat until all parents are selected

Wheel Selection

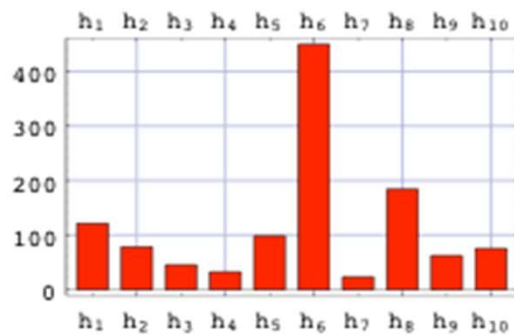
The probability of selecting an individual is proportional to its **fitness**



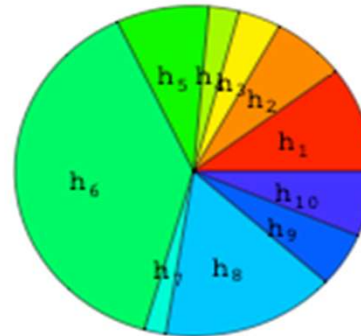
Probability of selection

Wheel Selection

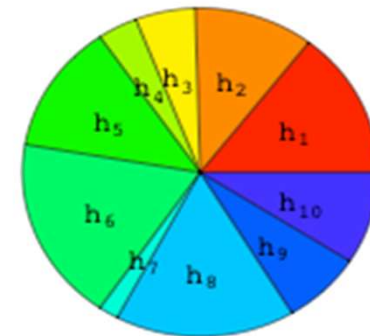
The probability of selecting an individual is proportional to its **rank**



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 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	00110110	00110110	00110110
Parent2	11011011	11011011	11011011
Off-spring1	01110111	00111011	01011010
Off-spring2	10011010	11010110	10110111

Mutation

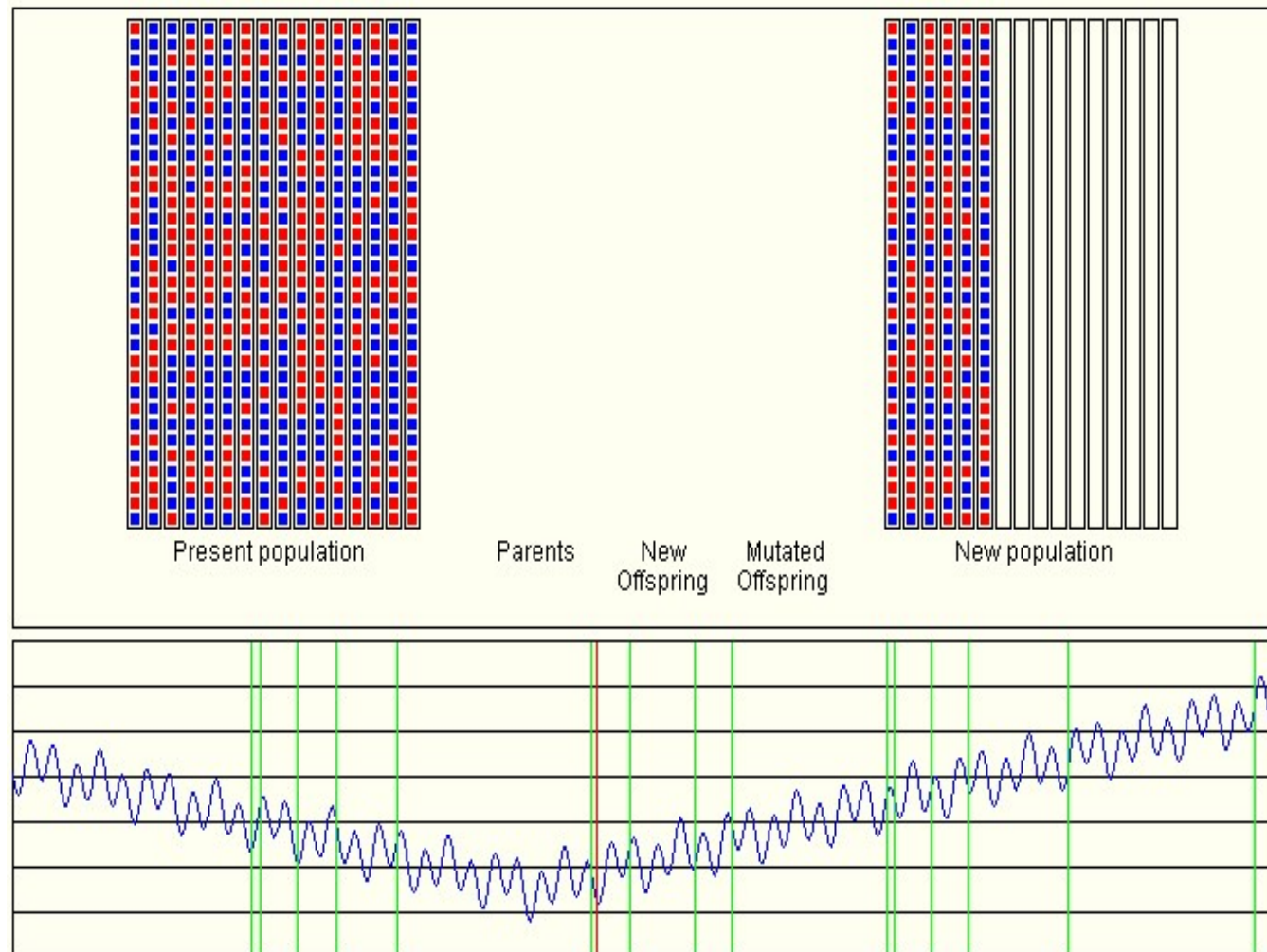
- Mutation procedures depend upon the representation schema of the chromosomes.
- This is to prevent falling all solutions in population into a local optimum.
- For a bit-vector representation:
 - random mutation : randomly negates bits
 - in-order mutation : performs random mutation between two randomly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

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

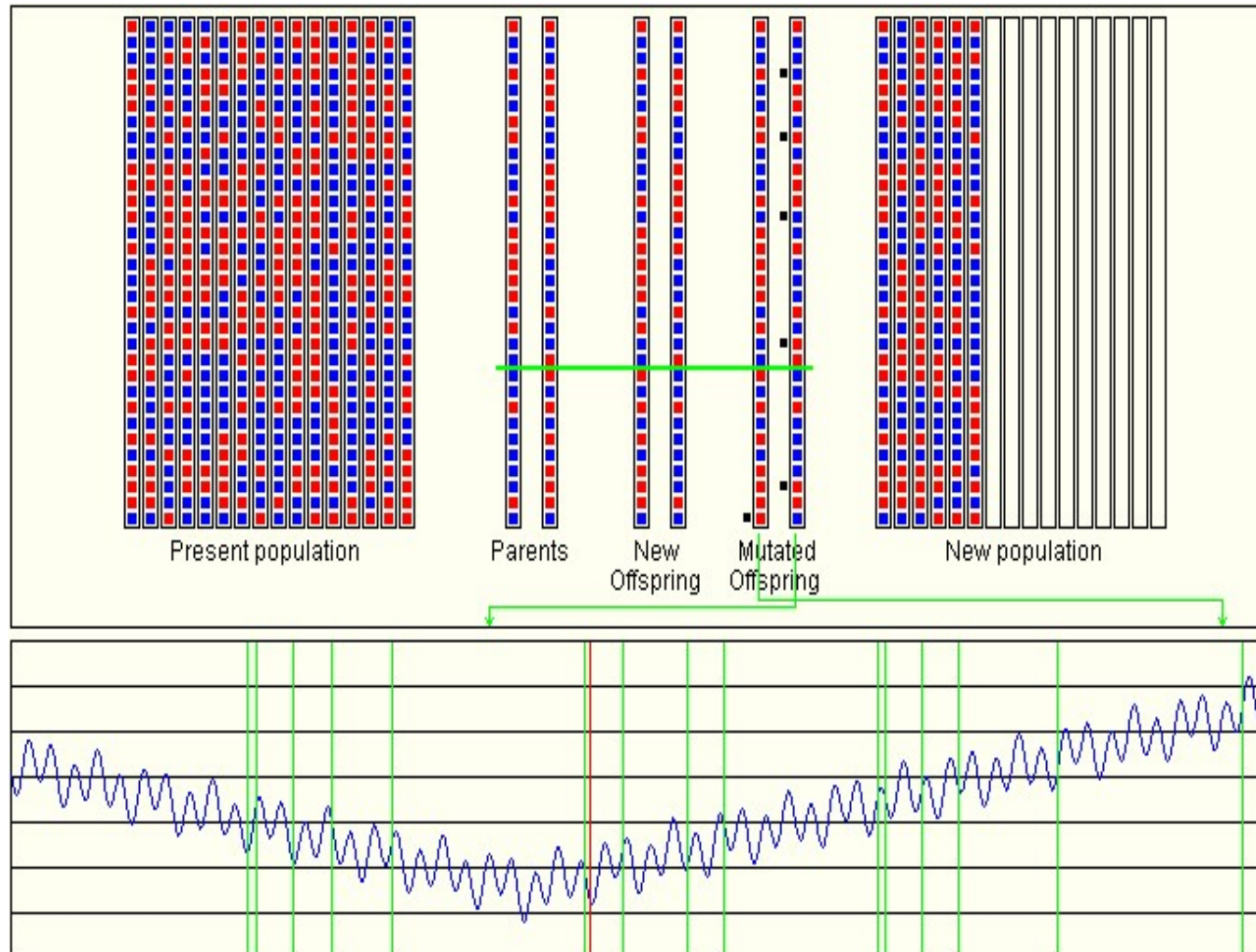
Operators

- 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

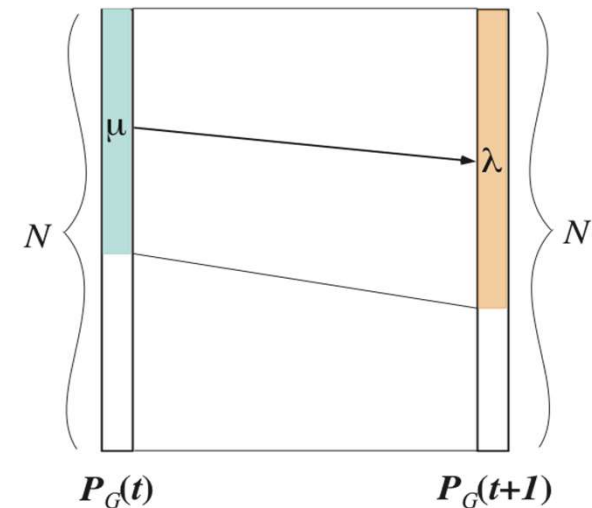


Introduction



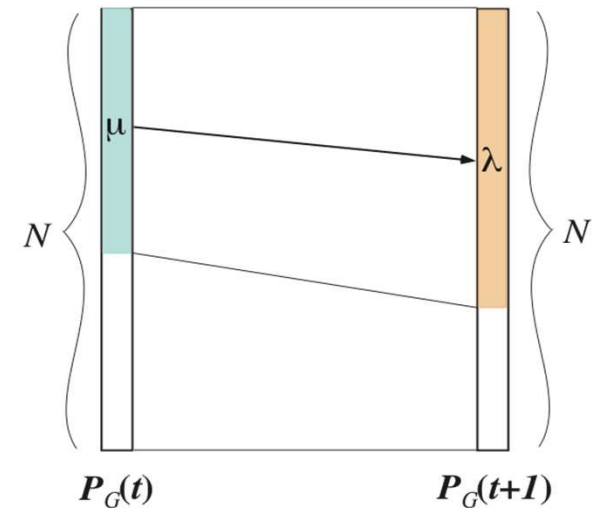
Replacements

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



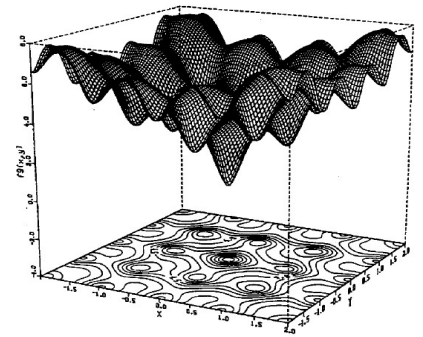
Strategies

1. Completely replace the previous population (called *(m,l) replacement*)
 - Risk: losing 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 $\{0, \dots, 31\}$
- GA approach
 - Representation: binary code (e.g. 0 1 1 0 1 \leftrightarrow 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

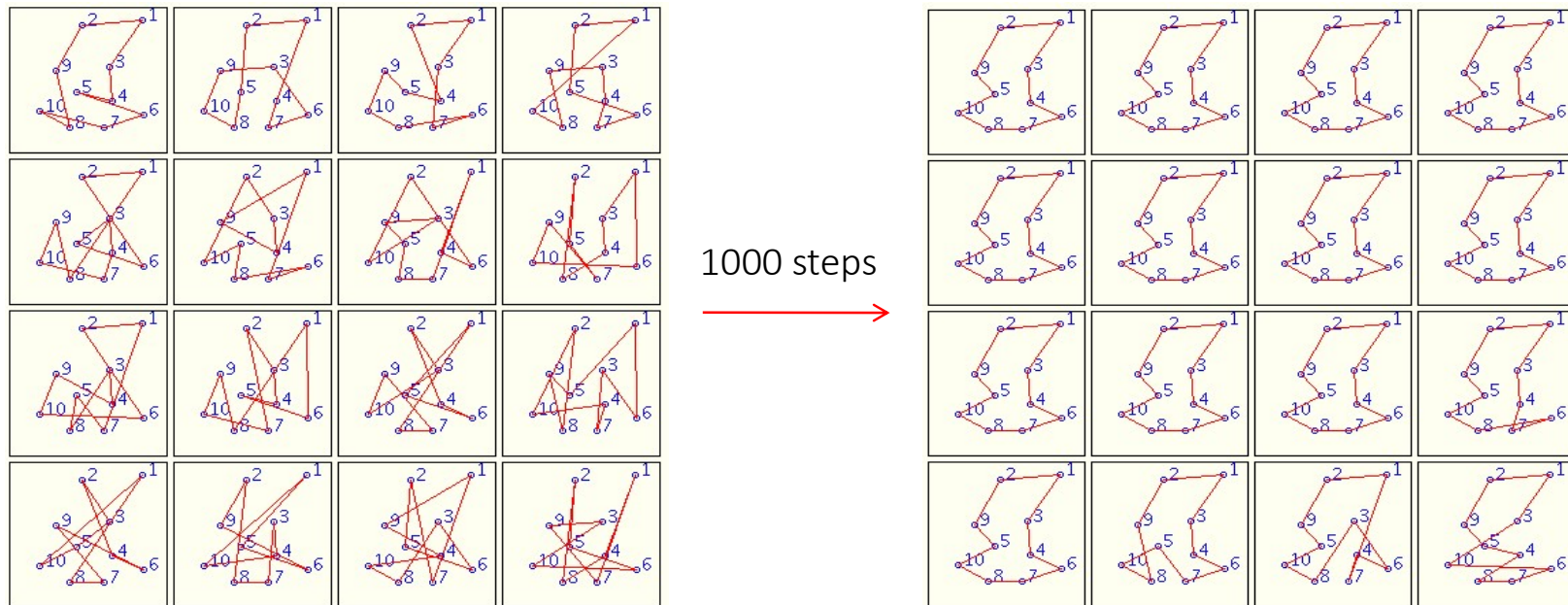
Mutation

String no.	Offspring after xover	Offspring after mutation	x Value	Fitness $f(x) = x^2$
1	0 1 1 0 0	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 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.

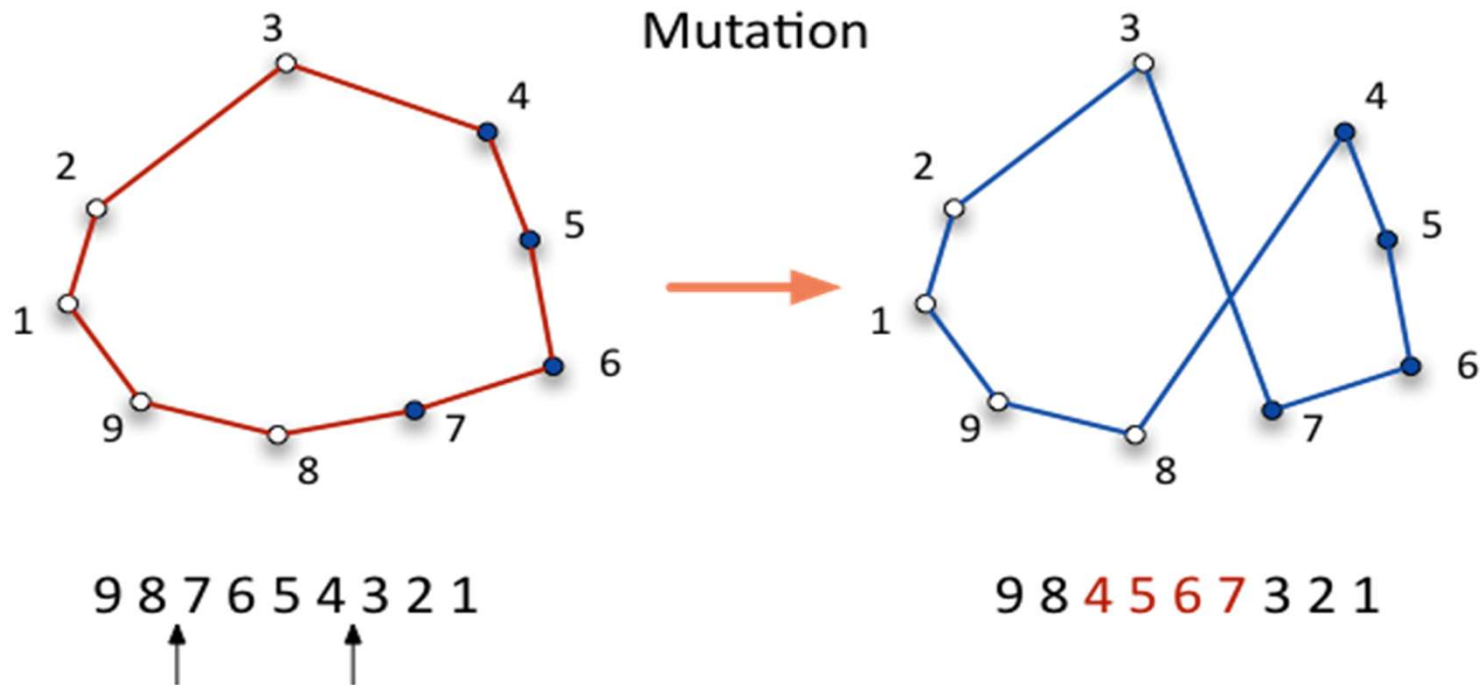
TSP



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

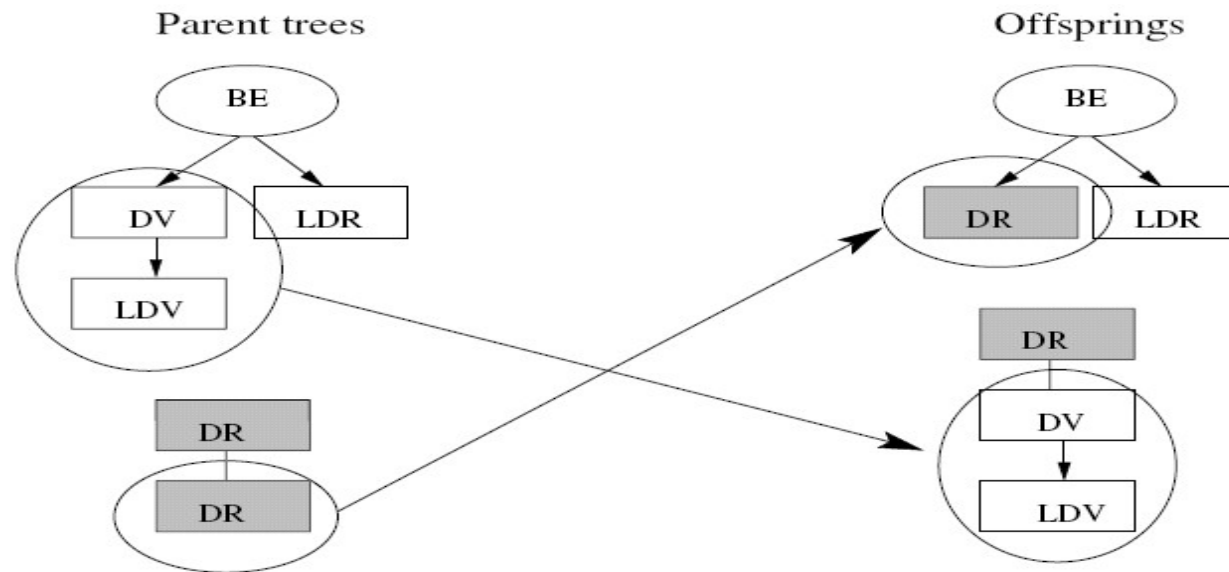
- Branch by Size (BS) : this primitive is used to select different sorting paths based on the size of the partition
- Branch by Entropy (BE): this primitive is used to select different paths based on the entropy of the input

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 $\sum_i -P_i * \log P_i$ 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

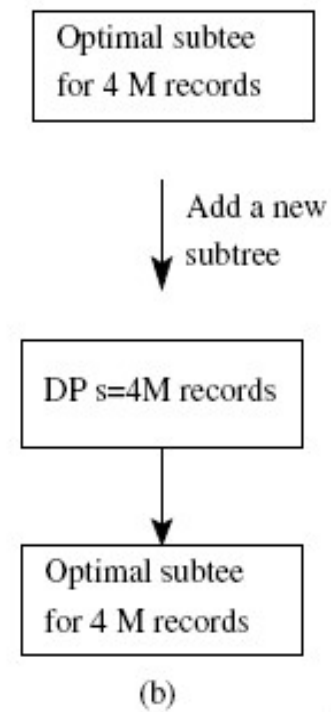
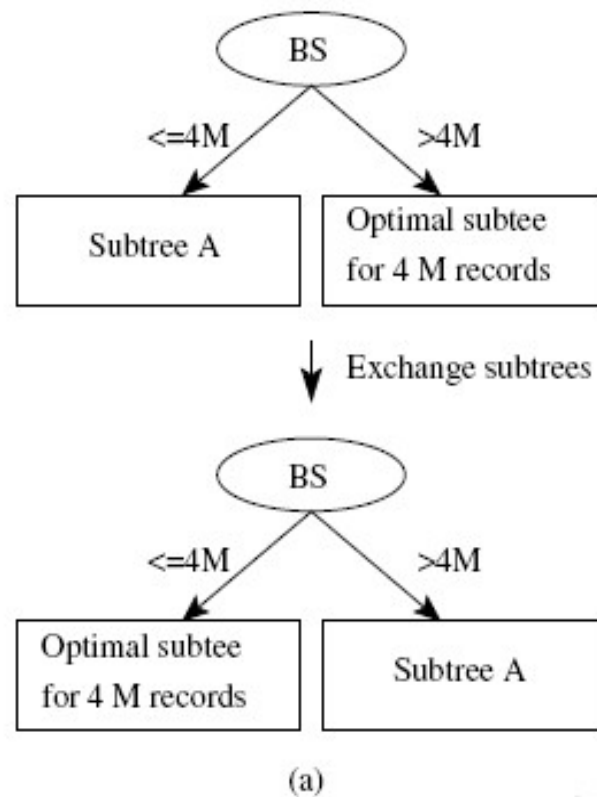
New offspring are generated using random single point crossovers



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

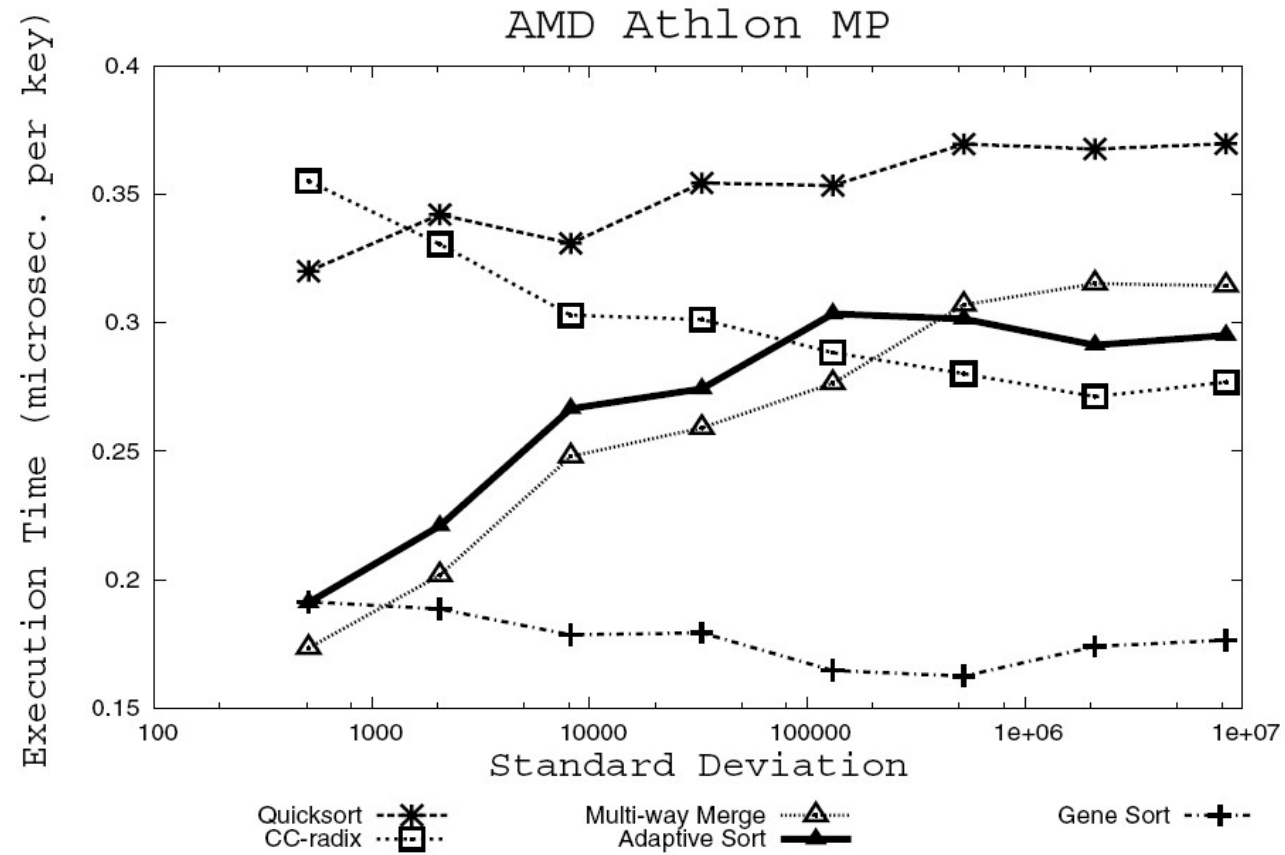
Optimizing Sorting



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



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.

Fuzzy Systems

Introduction

- Fuzzy logic is a mathematical language to **express** something.
This means it has grammar, syntax, semantic like a language for communication.
- There are some other mathematical languages also known
 - **Relational algebra** (operations on sets)
 - **Boolean algebra** (operations on Boolean variables)
 - **Predicate logic** (operations on well formed formulae (wff), also called predicate propositions)
- Fuzzy logic deals with **Fuzzy set**.

Introduction

- First time introduced by [Lotfi Abdelli Zadeh](#) (1965), University of California, Berkley, USA (1965).



- He is fondly nick-named as [LAZ](#)

Introduction

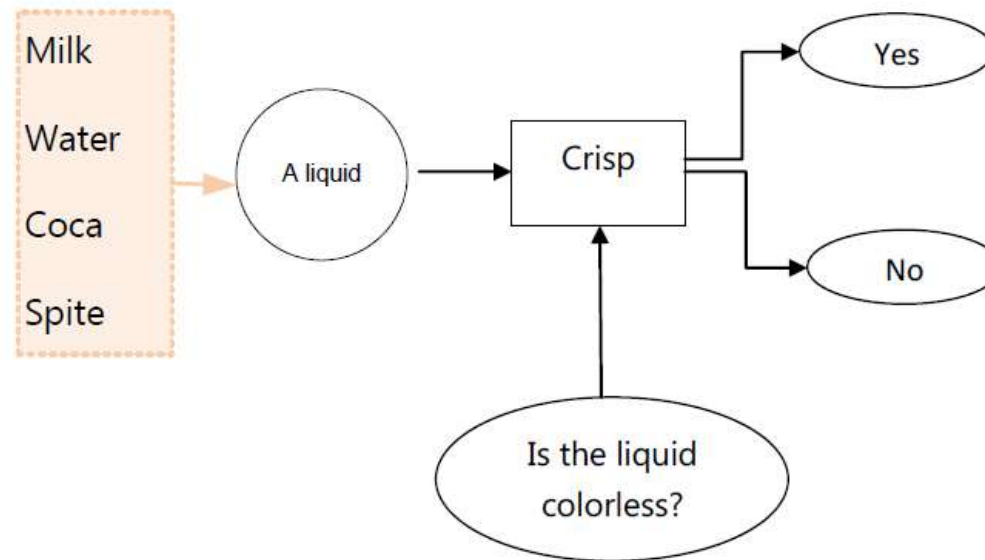
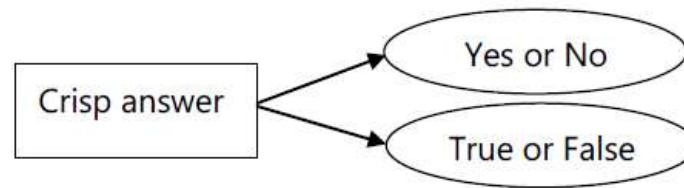
- 1 Dictionary meaning of **fuzzy** is not clear, noisy etc.

Example: Is the picture on this slide is fuzzy?

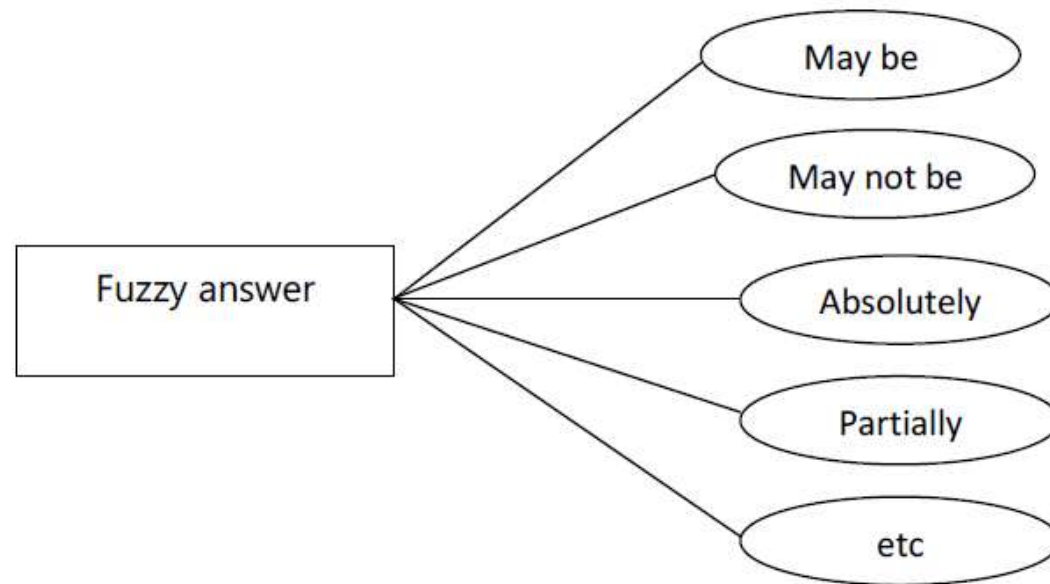
- 2 Antonym of fuzzy is **crisp**

Example: Are the chips crisp?

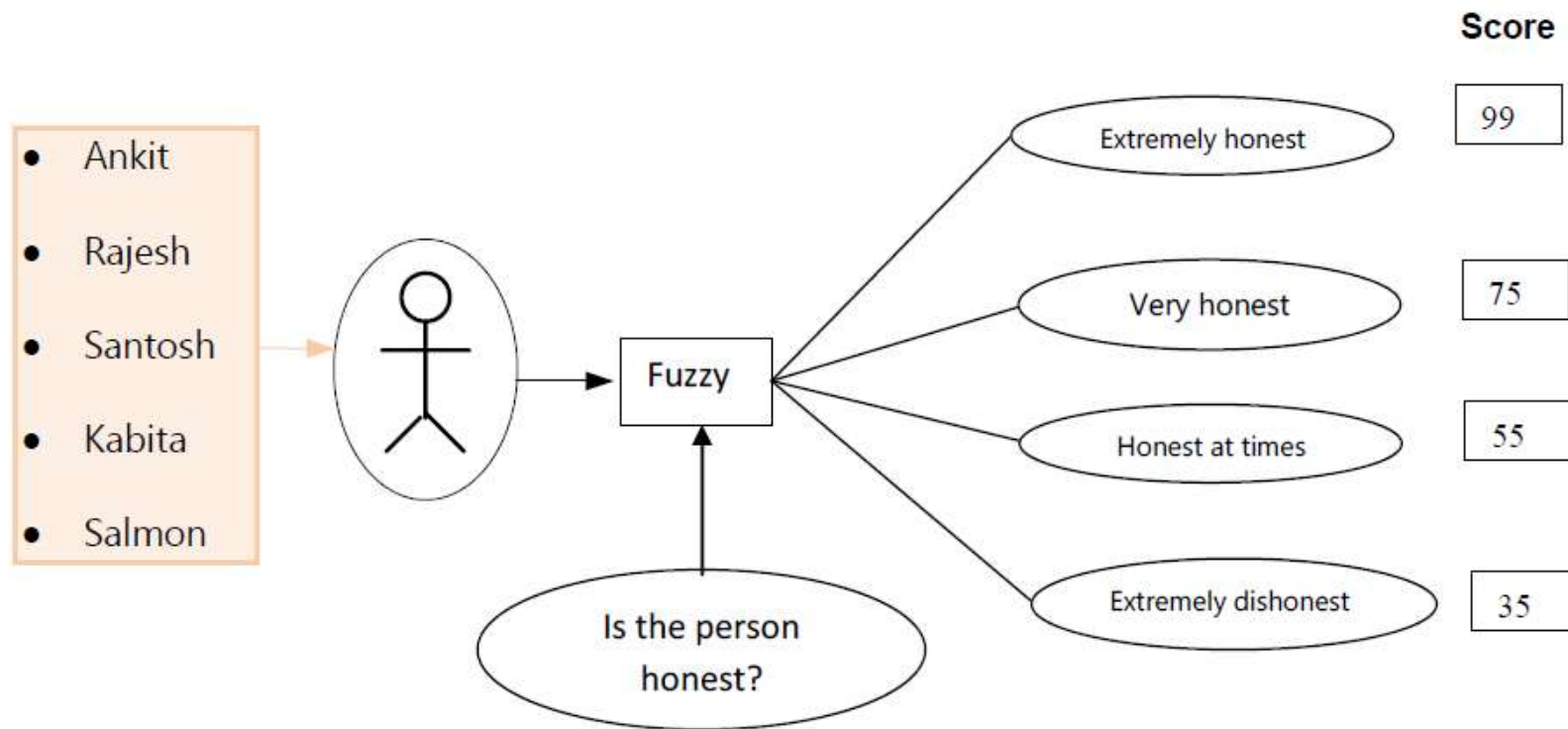
Introduction



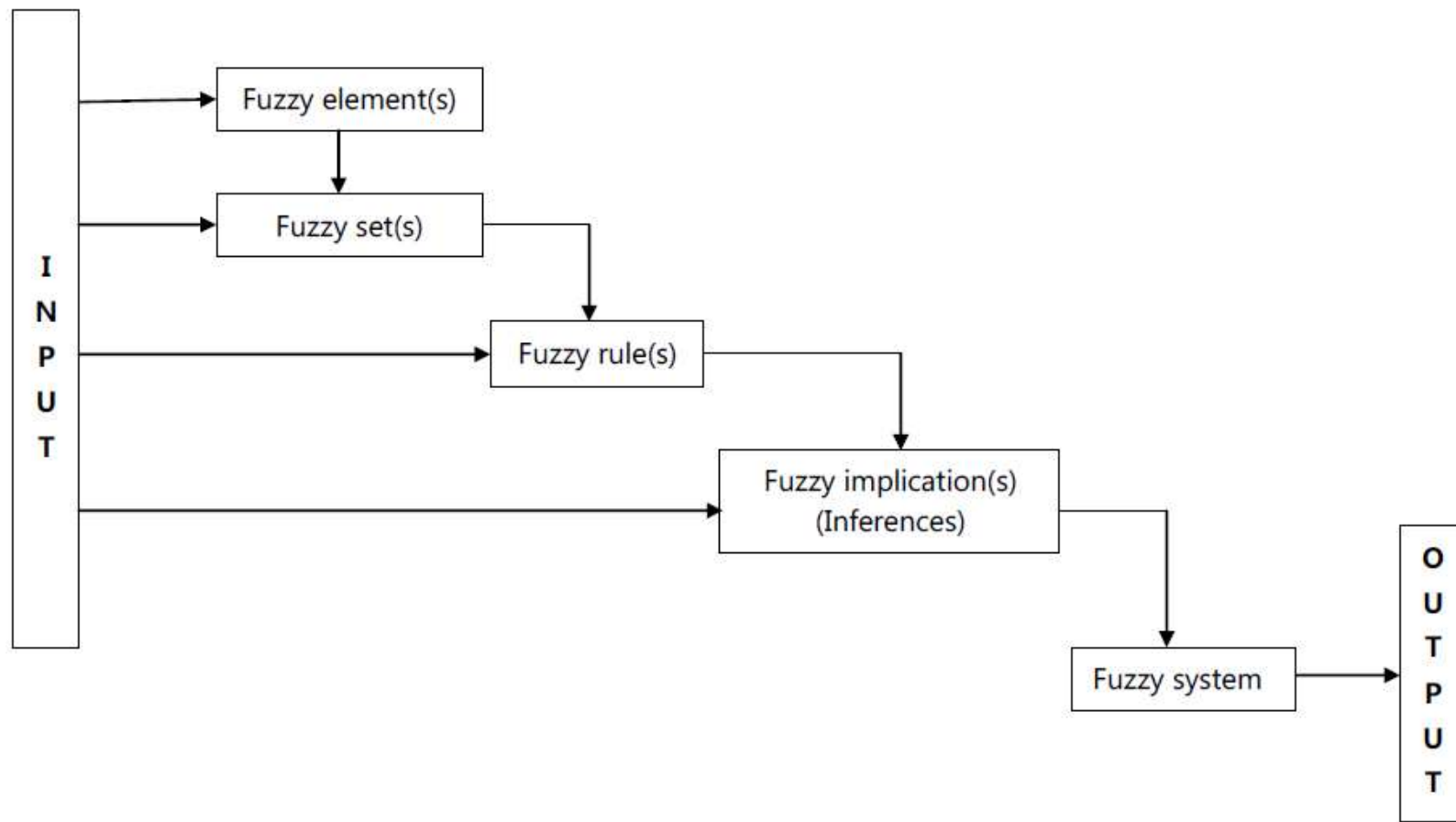
Introduction



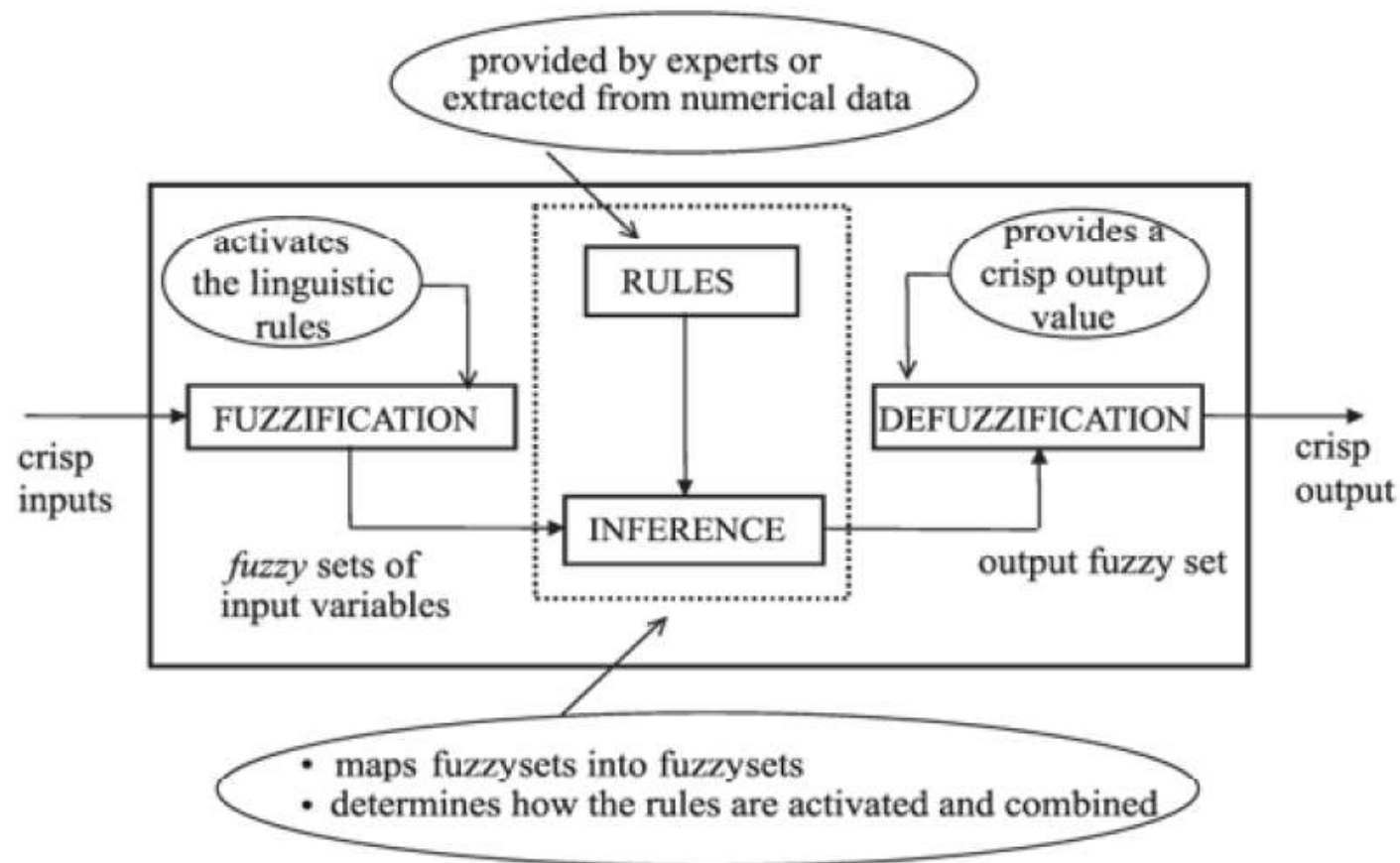
Introduction



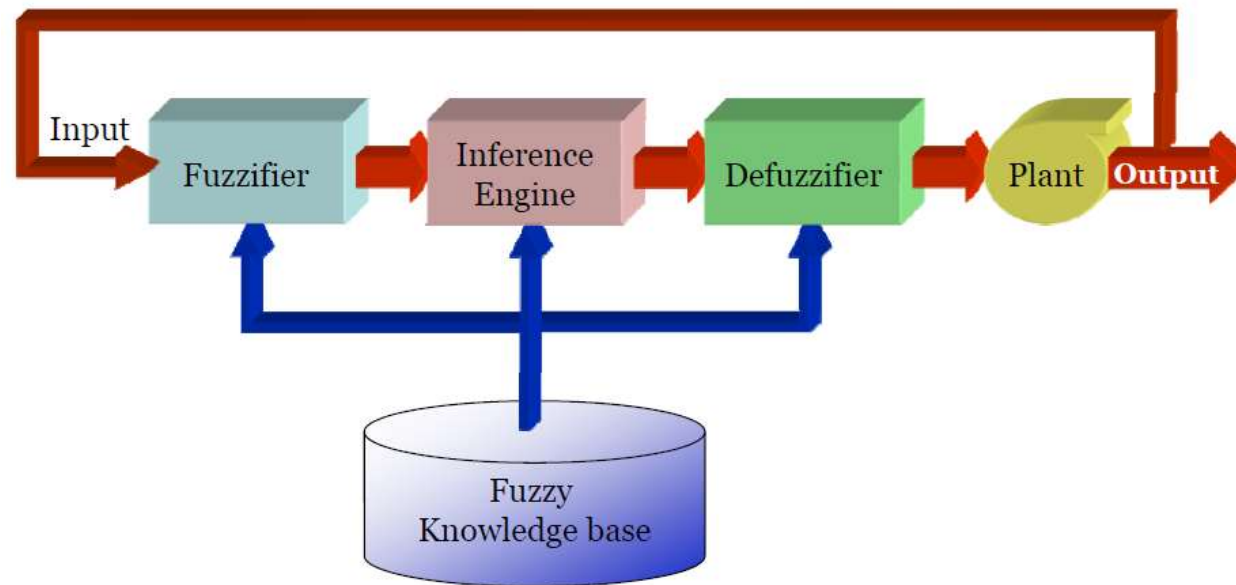
Phases



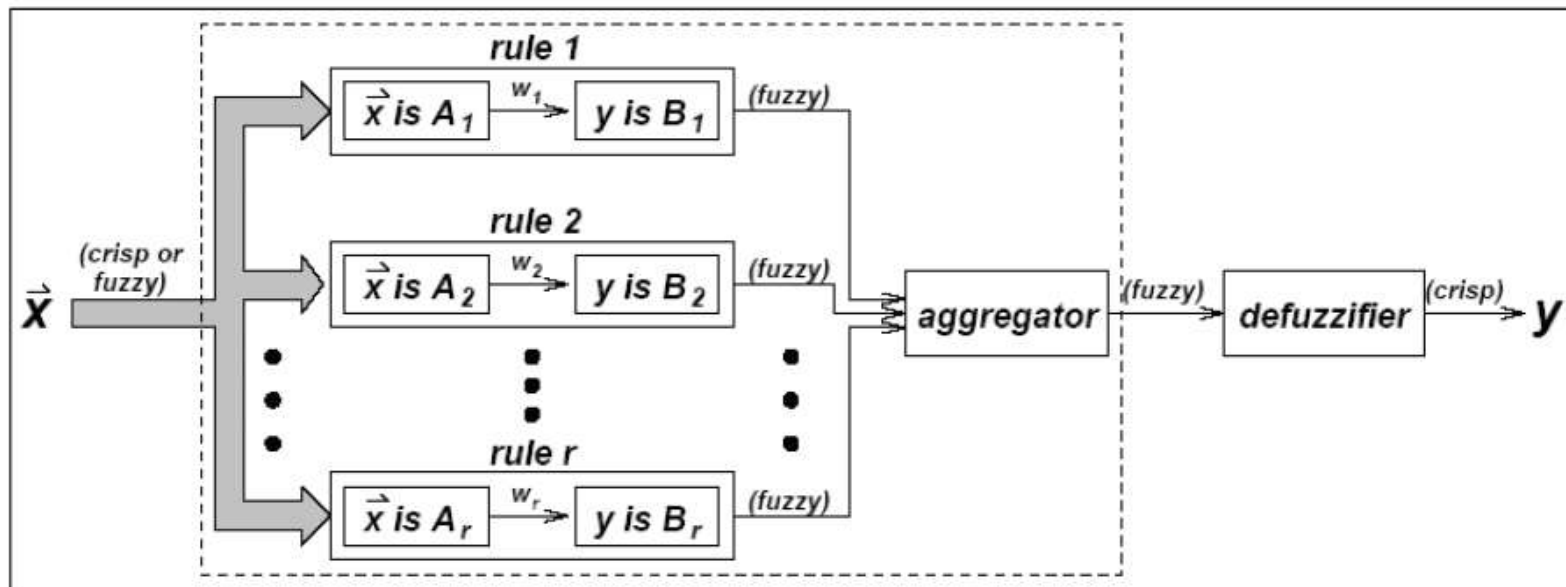
System



System

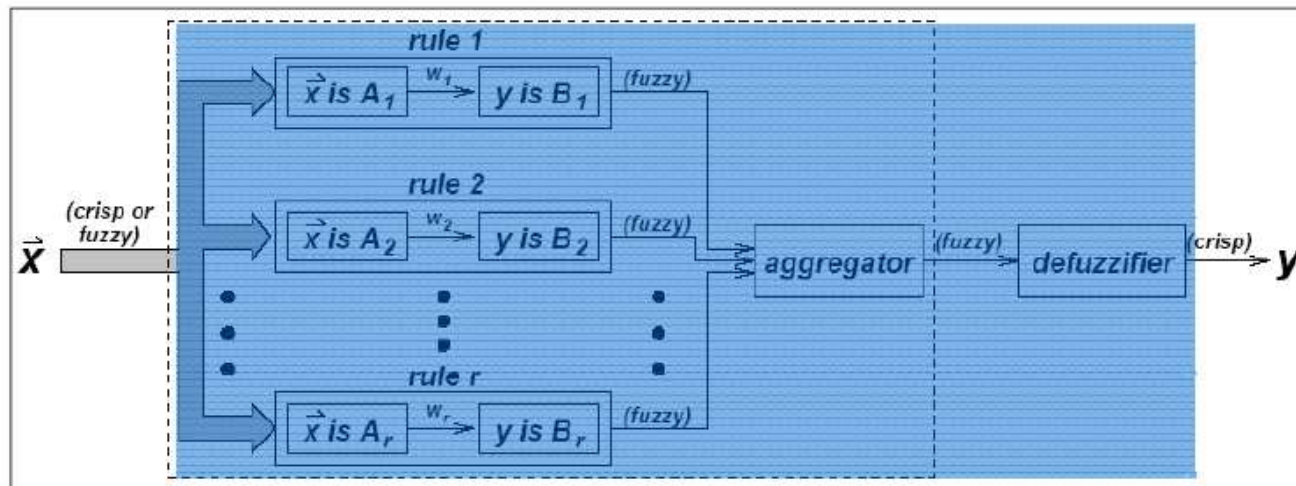


System



Mapping

In the case of crisp inputs & outputs, a fuzzy inference system implements a **nonlinear mapping** from its input space to output space.



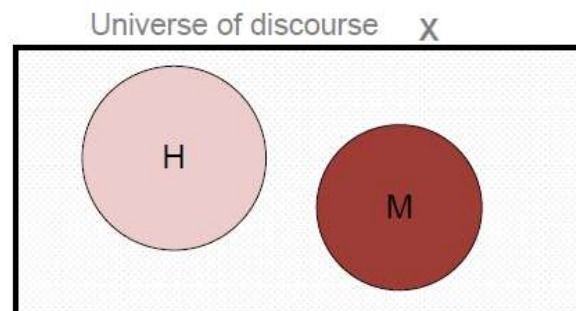
Fuzzy Sets

To understand the concept of **fuzzy set** it is better, if we first clear our idea of **crisp set**.

X = The entire population of India.

H = All Hindu population = $\{ h_1, h_2, h_3, \dots, h_L \}$

M = All Muslim population = $\{ m_1, m_2, m_3, \dots, m_N \}$



Here, All are the sets of finite numbers of individuals.

Such a set is called **crisp set**.

Fuzzy Sets

Let us discuss about fuzzy set.

X = All students in IT60108.

S = All **Good students**.

$S = \{ (s, g) \mid s \in X \}$ and $g(s)$ is a measurement of goodness of the student s .

Example:

$S = \{ (\text{Rajat}, 0.8), (\text{Kabita}, 0.7), (\text{Salman}, 0.1), (\text{Ankit}, 0.9) \}$ etc.

Fuzzy Sets

Crisp Set	Fuzzy Set
1. $S = \{ s \mid s \in X \}$	1. $F = (s, \mu) \mid s \in X \text{ and } \mu(s) \text{ is the degree of } s.$
2. It is a collection of elements.	2. It is collection of ordered pairs.
3. Inclusion of an element $s \in X$ into S is crisp, that is, has strict boundary yes or no .	3. Inclusion of an element $s \in X$ into F is fuzzy, that is, if present, then with a degree of membership .

Fuzzy Sets

Note: A crisp set is a fuzzy set, but, a fuzzy set is not necessarily a crisp set.

Example:

$$H = \{ (h_1, 1), (h_2, 1), \dots, (h_L, 1) \}$$

$$\text{Person} = \{ (p_1, 1), (p_2, 0), \dots, (p_N, 1) \}$$

In case of a crisp set, the elements are with extreme values of degree of membership namely either 1 or 0.

How to decide the degree of memberships of elements in a fuzzy set?

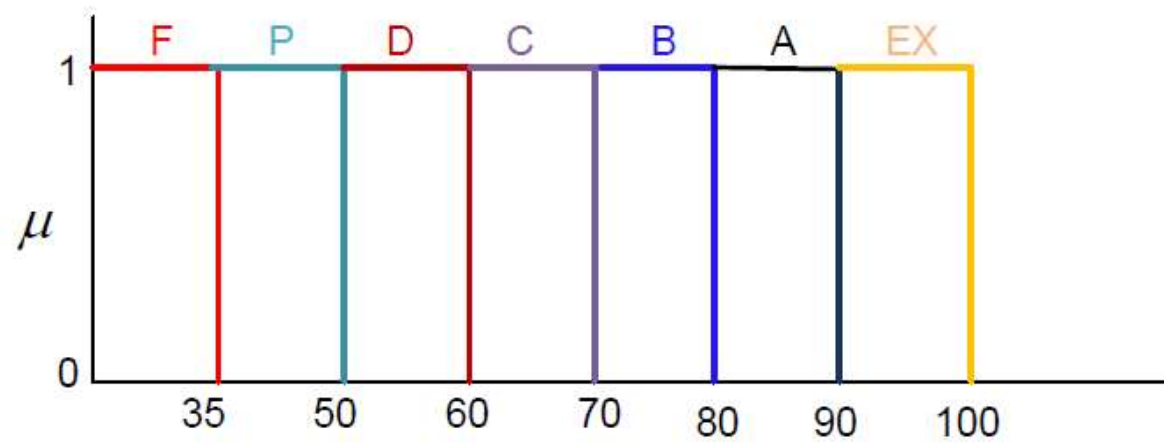
City	Bangalore	Bombay	Hyderabad	Kharagpur	Madras	Delhi
DoM	0.95	0.90	0.80	0.01	0.65	0.75

How the cities of **comfort** can be judged?

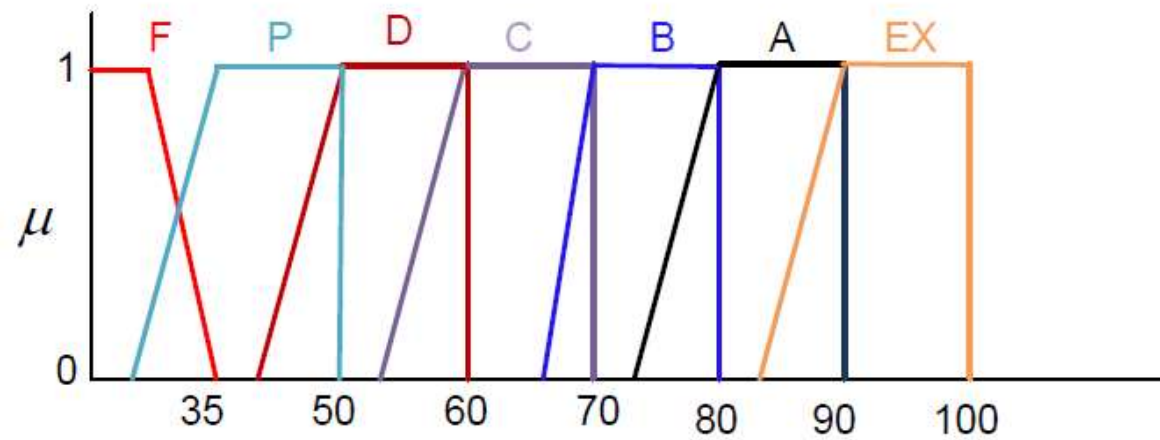
Fuzzy Sets

- ① $EX = \text{Marks} \geq 90$
- ② $A = 80 \leq \text{Marks} < 90$
- ③ $B = 70 \leq \text{Marks} < 80$
- ④ $C = 60 \leq \text{Marks} < 70$
- ⑤ $D = 50 \leq \text{Marks} < 60$
- ⑥ $P = 35 \leq \text{Marks} < 50$
- ⑦ $F = \text{Marks} < 35$

Fuzzy Sets



Fuzzy Sets



Examples

- High Temperature
- Low Pressure
- Color of Apple
- Sweetness of Orange
- Weight of Mango

Note: Degree of membership values lie in the range $[0...1]$.

Fuzzy Sets

Definition 1: Membership function (and Fuzzy set)

If X is a universe of discourse and $x \in X$, then a fuzzy set A in X is defined as a set of ordered pairs, that is

$A = \{(x, \mu_A(x)) | x \in X\}$ where $\mu_A(x)$ is called the **membership function** for the fuzzy set A .

Note:

$\mu_A(x)$ map each element of X onto a membership grade (or membership value) between 0 and 1 (both inclusive).

Question:

How (and who) decides $\mu_A(x)$ for a Fuzzy set A in X ?

Fuzzy Sets

Example:

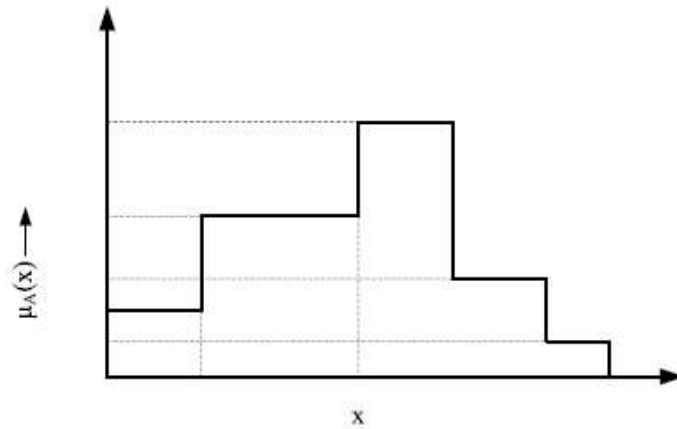
X = All cities in India

A = City of comfort

$A = \{(\text{New Delhi}, 0.7), (\text{Bangalore}, 0.9), (\text{Chennai}, 0.8), (\text{Hyderabad}, 0.6), (\text{Kolkata}, 0.3), (\text{Kharagpur}, 0)\}$

Fuzzy Sets

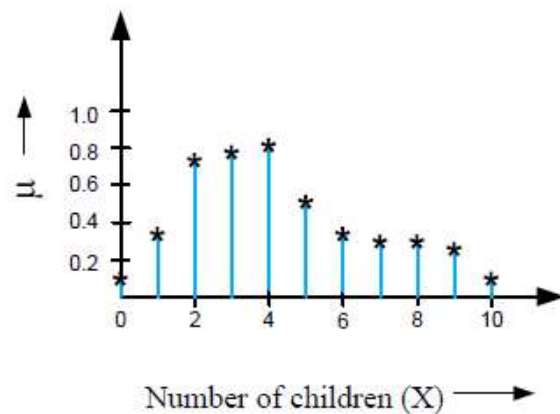
The membership values may be of discrete values.



A fuzzy set with discrete values of μ

Fuzzy Sets

Either elements or their membership values (or both) also may be of discrete values.



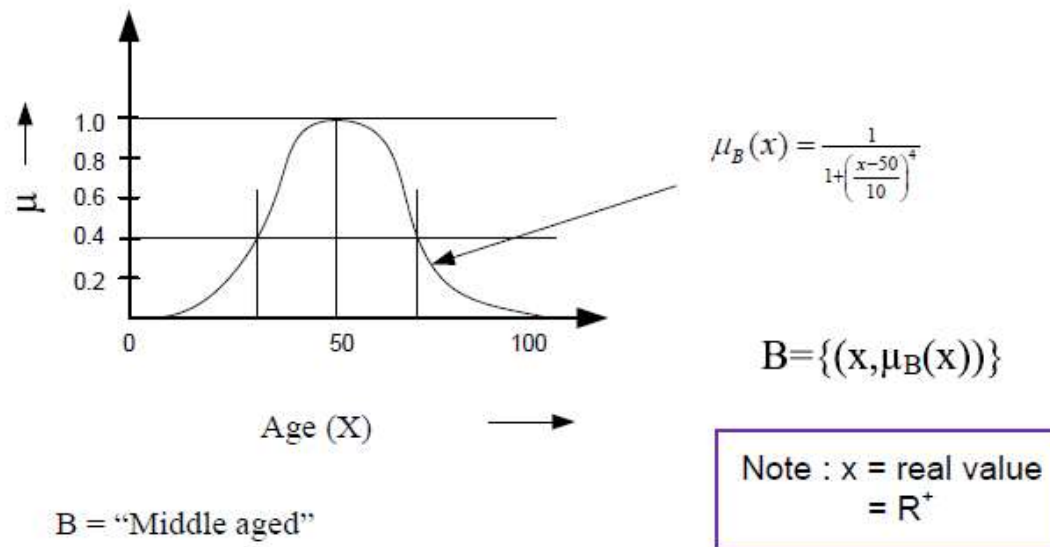
$$A = \{(0, 0.1), (1, 0.30), (2, 0.78), \dots, (10, 0.1)\}$$

Note : X = discrete value

How you measure happiness ??

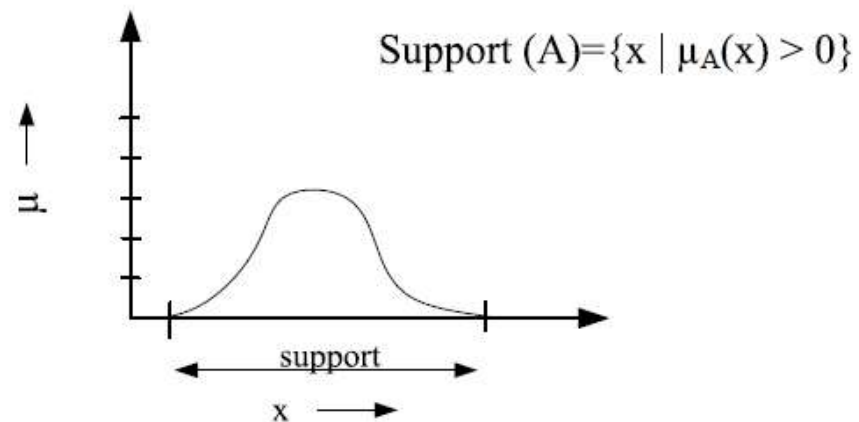
A = "Happy family"

Fuzzy Sets



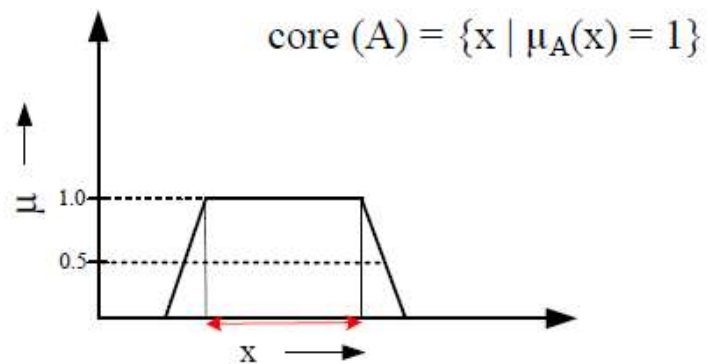
Fuzzy Sets

Support: The support of a fuzzy set A is the set of all points $x \in X$ such that $\mu_A(x) > 0$



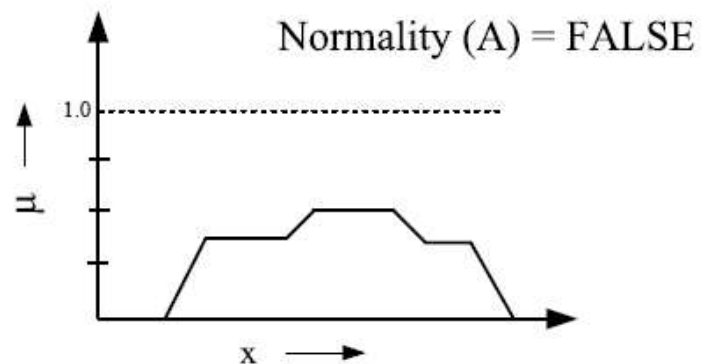
Fuzzy Sets

Core: The core of a fuzzy set A is the set of all points x in X such that $\mu_A(x) = 1$



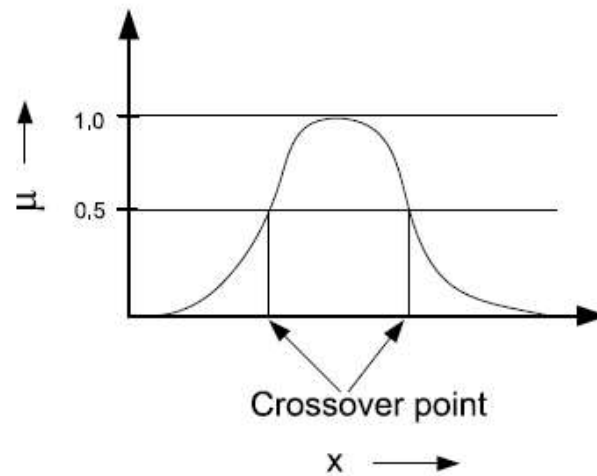
Fuzzy Sets

Normality : A fuzzy set A is a normal if its core is non-empty. In other words, we can always find a point $x \in X$ such that $\mu_A(x) = 1$.



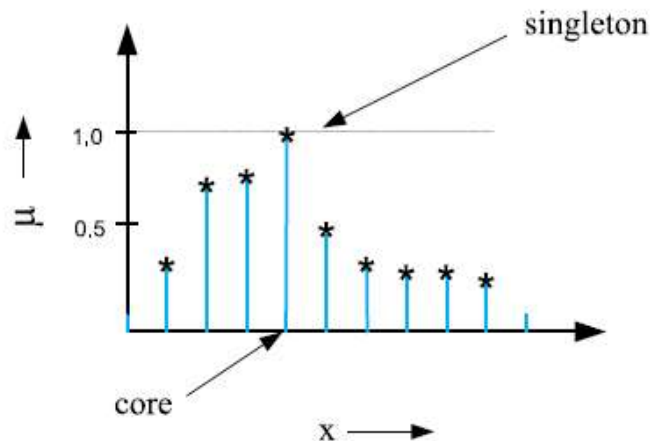
Fuzzy Sets

Crossover point : A crossover point of a fuzzy set A is a point $x \in X$ at which $\mu_A(x) = 0.5$. That is
 $\text{Crossover}(A) = \{x | \mu_A(x) = 0.5\}$.



Fuzzy Sets

Fuzzy Singleton : A fuzzy set whose support is a single point in X with $\mu_A(x) = 1$ is called a fuzzy singleton. That is $|A| = |\{x \mid \mu_A(x) = 1\}| = 1$. Following fuzzy set is not a fuzzy singleton.



Fuzzy Sets

α -cut and strong α -cut :

The α -cut of a fuzzy set A is a crisp set defined by

$$A_{\alpha} = \{x \mid \mu_A(x) \geq \alpha \}$$

Strong α -cut is defined similarly :

$$A_{\alpha}' = \{x \mid \mu_A(x) > \alpha \}$$

Note : $\text{Support}(A) = A_0'$ and $\text{Core}(A) = A_1$.

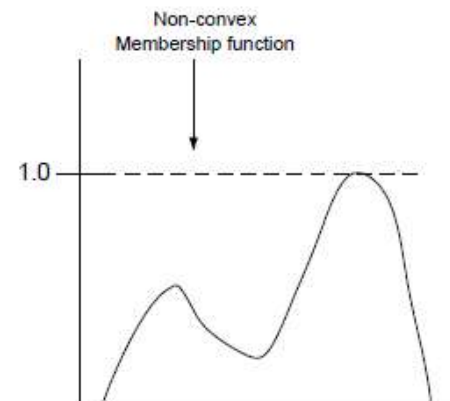
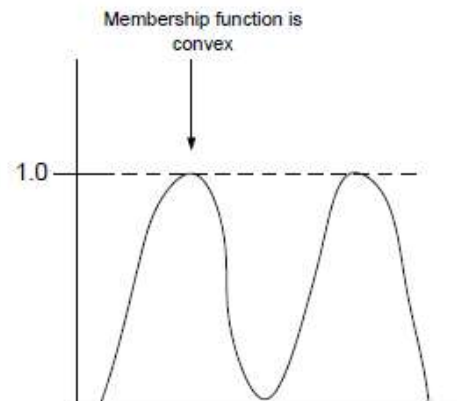
Fuzzy Sets

Convexity : A fuzzy set A is convex if and only if for any x_1 and $x_2 \in X$ and any $\lambda \in [0, 1]$

$$\mu_A(\lambda x_1 + (1 - \lambda)x_2) \geq \min(\mu_A(x_1), \mu_A(x_2))$$

Note :

- A is convex if all its α - level sets are convex.
- Convexity (A_α) $\implies A_\alpha$ is composed of a single line segment only.



Fuzzy Sets

Bandwidth :

For a normal and convex fuzzy set, the bandwidth (or width) is defined as the distance the two unique crossover points:

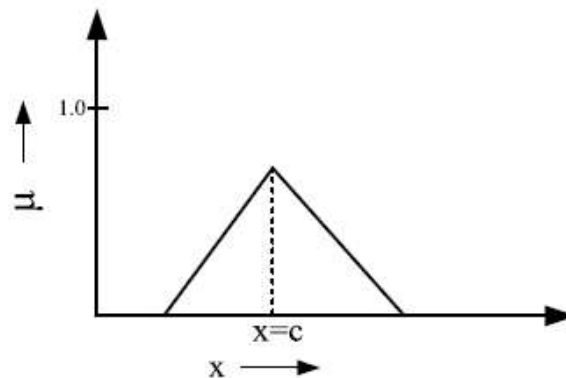
$$\text{Bandwidth}(A) = |x_1 - x_2|$$

where $\mu_A(x_1) = \mu_A(x_2) = 0.5$

Fuzzy Sets

Symmetry :

A fuzzy set A is symmetric if its membership function around a certain point $x = c$, namely $\mu_A(x + c) = \mu_A(x - c)$ for all $x \in X$.



Fuzzy Sets

A fuzzy set A is

Open left

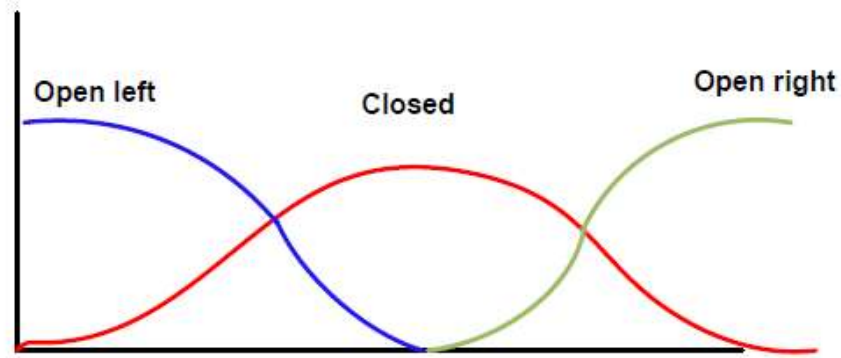
If $\lim_{x \rightarrow -\infty} \mu_A(x) = 1$ and $\lim_{x \rightarrow +\infty} \mu_A(x) = 0$

Open right:

If $\lim_{x \rightarrow -\infty} \mu_A(x) = 0$ and $\lim_{x \rightarrow +\infty} \mu_A(x) = 1$

Closed

If : $\lim_{x \rightarrow -\infty} \mu_A(x) = \lim_{x \rightarrow +\infty} \mu_A(x) = 0$



Fuzzy vs Probability

Fuzzy : When we say about certainty of a thing

Example: A patient come to the doctor and he has to diagnose so that medicine can be prescribed.

Doctor prescribed a medicine with certainty 60% that the patient is suffering from flue. So, the disease will be cured with certainty of 60% and uncertainty 40%. Here, in stead of flue, other diseases with some other certainties may be.

Probability: When we say about the chance of an event to occur

Example: India will win the T20 tournament with a chance 60% means that out of 100 matches, India own 60 matches.

Prediction vs Forecasting

The Fuzzy vs. Probability is analogical to Prediction vs. Forecasting

Prediction : When you start guessing about things.

Forecasting : When you take the information from the past job and apply it to new job.

The main difference:

Prediction is based on the best guess from experiences.

Forecasting is based on data you have actually recorded and packed from previous job.

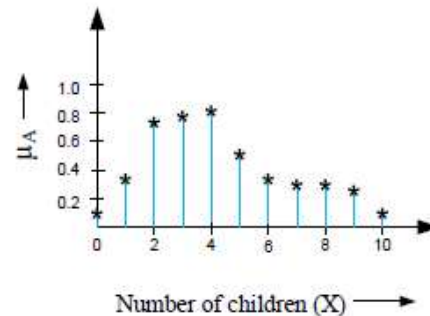
Membership Functions

A fuzzy set is completely characterized by its membership function (sometimes abbreviated as *MF* and denoted as μ). So, it would be important to learn how a membership function can be expressed (mathematically or otherwise).

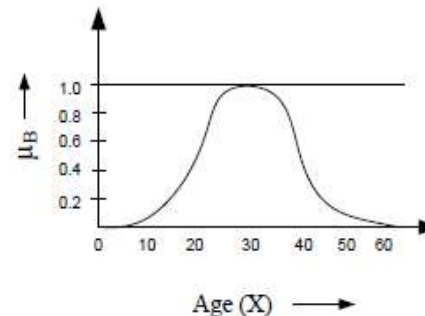
Note: A membership function can be on

- (a) a discrete universe of discourse and
- (b) a continuous universe of discourse.

Example:



A = Fuzzy set of "Happy family"

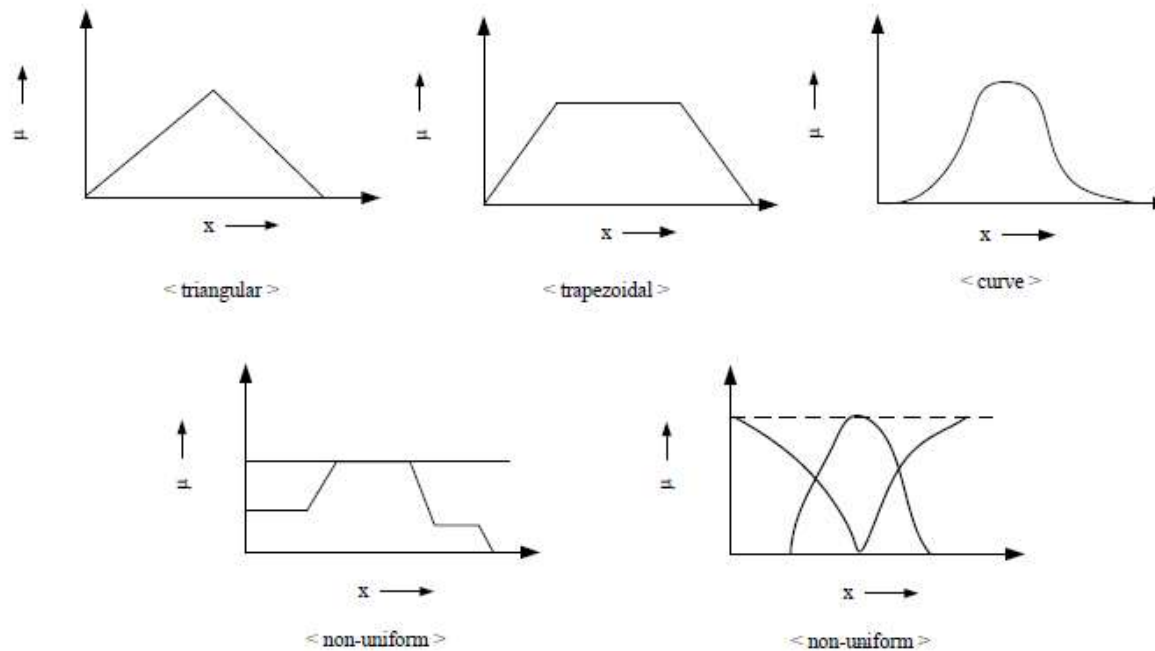


B = "Young age"

Membership Functions

So, membership function on a discrete universe of course is trivial. However, a membership function on a continuous universe of discourse needs a special attention.

Following figures shows a typical examples of membership functions.

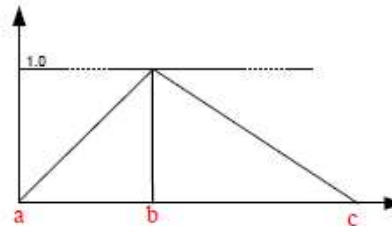


Membership Functions

In the following, we try to parameterize the different MFs on a continuous universe of discourse.

Triangular MFs : A triangular MF is specified by three parameters $\{a, b, c\}$ and can be formulated as follows.

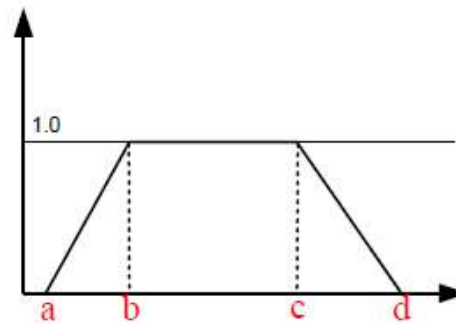
$$\text{triangle}(x; a, b, c) = \begin{cases} 0 & \text{if } x \leq a \\ \frac{x-a}{b-a} & \text{if } a \leq x \leq b \\ \frac{c-x}{c-b} & \text{if } b \leq x \leq c \\ 0 & \text{if } c \leq x \end{cases}$$



Membership Functions

A trapezoidal MF is specified by four parameters $\{a, b, c, d\}$ and can be defined as follows:

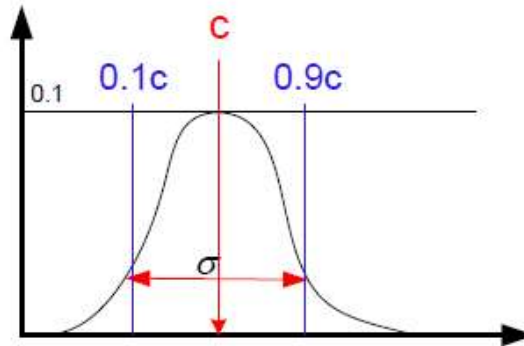
$$\text{trapezoid}(x; a, b, c, d) = \begin{cases} 0 & \text{if } x \leq a \\ \frac{x-a}{b-a} & \text{if } a \leq x \leq b \\ 1 & \text{if } b \leq x \leq c \\ \frac{d-x}{d-c} & \text{if } c \leq x \leq d \\ 0 & \text{if } d \leq x \end{cases}$$



Membership Functions

A Gaussian MF is specified by two parameters $\{c, \sigma\}$ and can be defined as below:

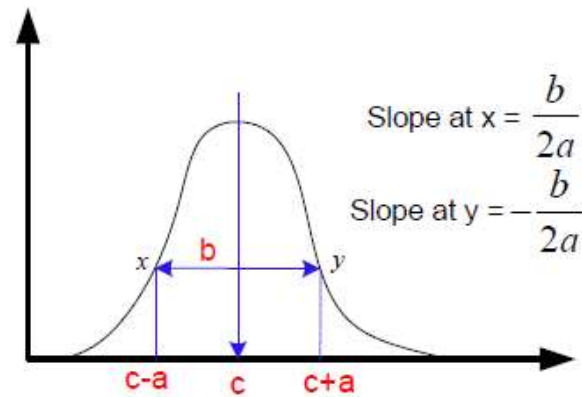
$$\text{gaussian}(x; c, \sigma) = e^{-\frac{1}{2}\left(\frac{x-c}{\sigma}\right)^2}.$$



Membership Functions

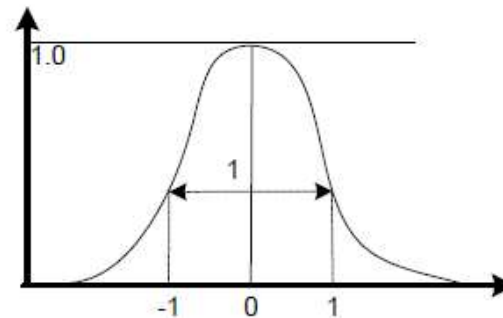
It is also called **Cauchy MF**. A generalized bell MF is specified by three parameters $\{a, b, c\}$ and is defined as:

$$\text{bell}(x; a, b, c) = \frac{1}{1 + \left| \frac{x-c}{a} \right|^{2b}}$$

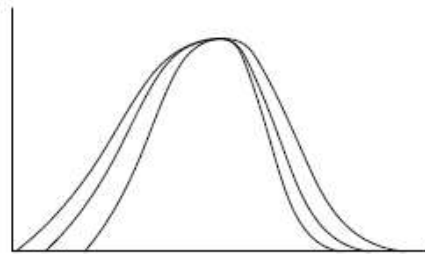


Membership Functions

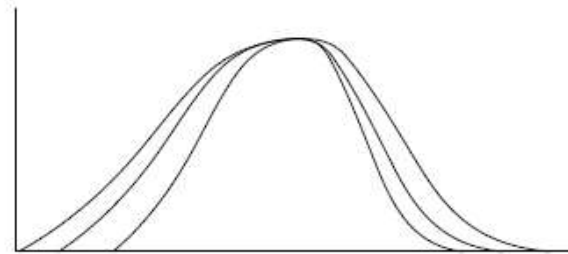
Example: $\mu(x) = \frac{1}{1+x^2}$;
 $a = b = 1$ and $c = 0$;



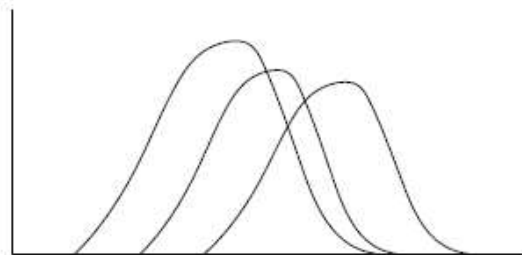
Membership Functions



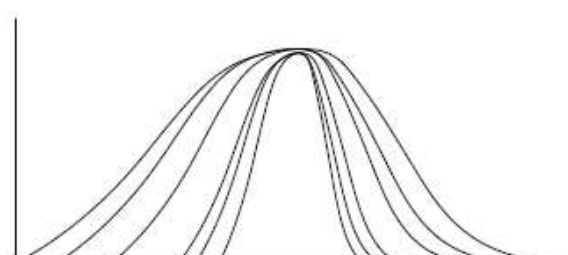
Changing a



Changing b



Changing a

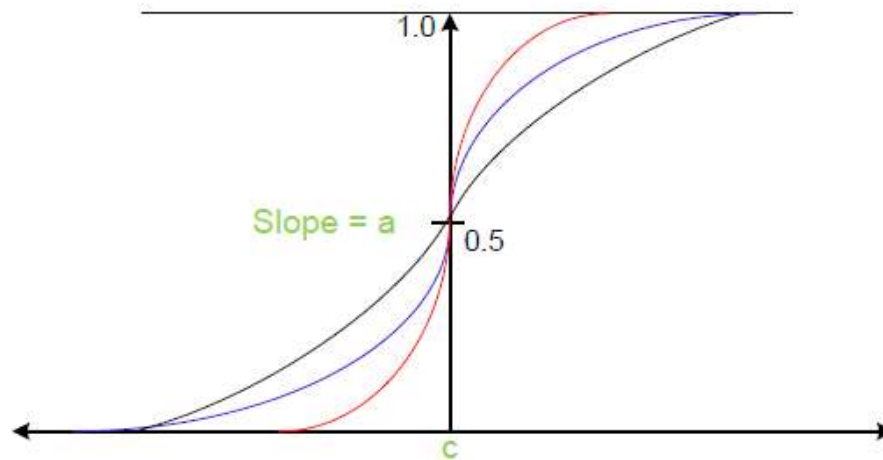


Changing a and b

Membership Functions

Parameters: $\{a, c\}$; where c = crossover point and a = slope at c ;

$$\text{Sigmoid}(x;a,c) = \frac{1}{1 + e^{-[\frac{a}{x-c}]}}$$



Membership Functions

Example : Consider the following grading system for a course.

Excellent = Marks ≤ 90

Very good = $75 \leq \text{Marks} \leq 90$

Good = $60 \leq \text{Marks} \leq 75$

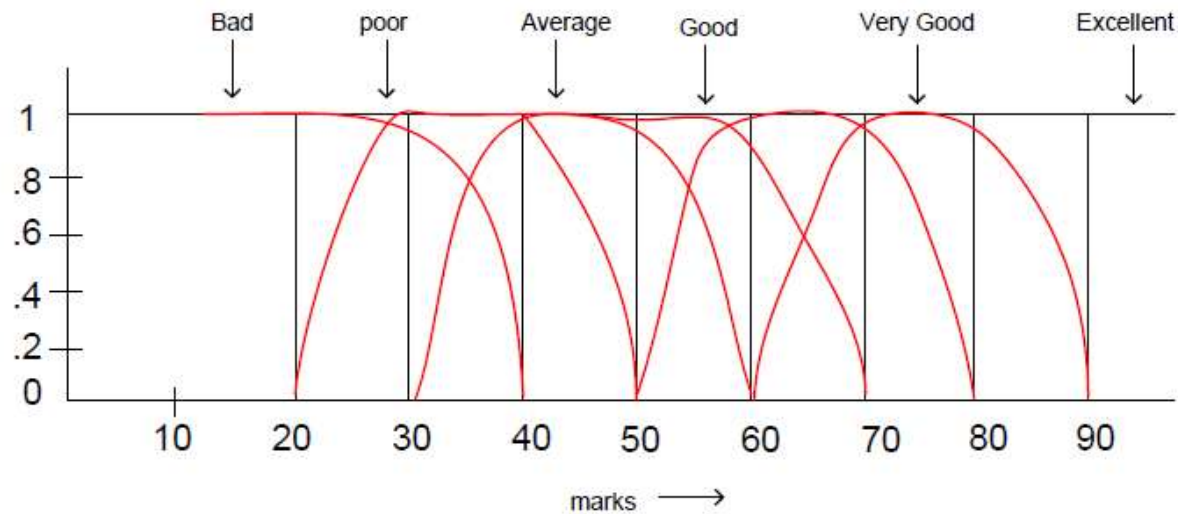
Average = $50 \leq \text{Marks} \leq 60$

Poor = $35 \leq \text{Marks} \leq 50$

Bad = Marks ≤ 35

Membership Functions

A fuzzy implementation will look like the following.



Operations

Union ($A \cup B$):

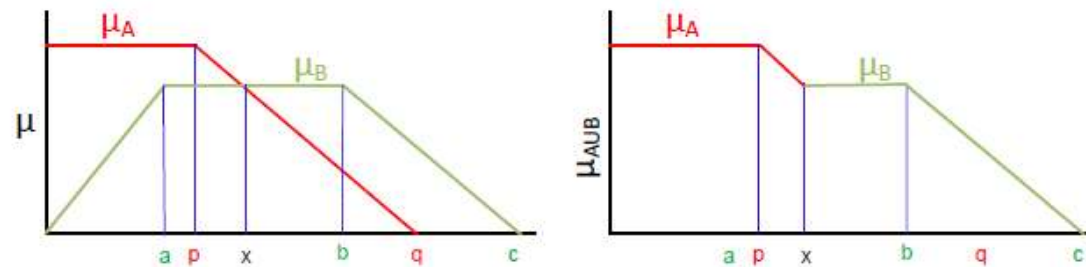
$$\mu_{A \cup B}(x) = \max\{\mu_A(x), \mu_B(x)\}$$

Example:

$A = \{(x_1, 0.5), (x_2, 0.1), (x_3, 0.4)\}$ and

$B = \{(x_1, 0.2), (x_2, 0.3), (x_3, 0.5)\};$

$C = A \cup B = \{(x_1, 0.5), (x_2, 0.3), (x_3, 0.5)\}$



Operations

Intersection ($A \cap B$):

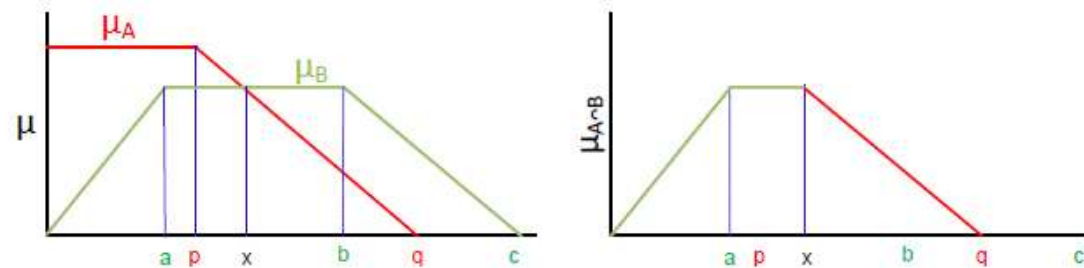
$$\mu_{A \cap B}(x) = \min\{\mu_A(x), \mu_B(x)\}$$

Example:

$A = \{(x_1, 0.5), (x_2, 0.1), (x_3, 0.4)\}$ and

$B = \{(x_1, 0.2), (x_2, 0.3), (x_3, 0.5)\};$

$C = A \cap B = \{(x_1, 0.2), (x_2, 0.1), (x_3, 0.4)\}$



Operations

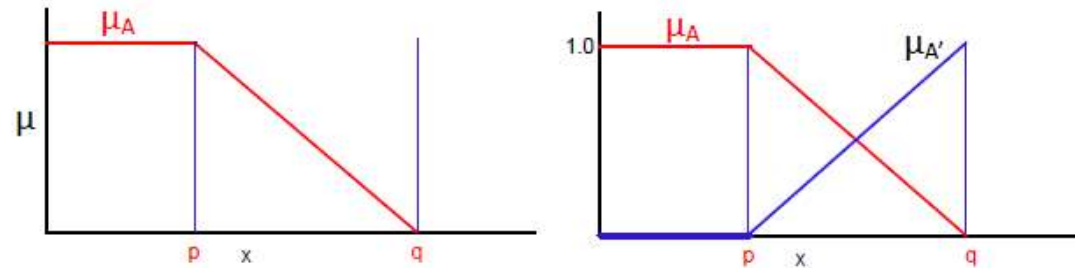
Complement (A^C):

$$\mu_{A^C}(X) = 1 - \mu_A(X)$$

Example:

$$A = \{(x_1, 0.5), (x_2, 0.1), (x_3, 0.4)\}$$

$$C = A^C = \{(x_1, 0.5), (x_2, 0.9), (x_3, 0.6)\}$$



Operations

Algebraic product or Vector product ($A \bullet B$):

$$\mu_{A \bullet B}(X) = \mu_A(X) \bullet \mu_B(X)$$

Scalar product ($\alpha \times A$):

$$\mu_{\alpha A}(X) = \alpha \cdot \mu_A(X)$$

Operations

Sum ($A + B$):

$$\mu_{A+B}(X) = \mu_A(X) + \mu_B(X) - \mu_A(X) \cdot \mu_B(X)$$

Difference ($A - B = A \cap B^C$):

$$\mu_{A-B}(X) = \mu_{A \cap B^C}(X)$$

Disjunctive sum: $A \oplus B = (A^C \cap B) \cup (A \cap B^C)$

Bounded Sum: $|A(x) \oplus B(x)|$

$$\mu_{|A(x) \oplus B(x)|} = \min\{1, \mu_A(X) + \mu_B(X)\}$$

Bounded Difference: $|A(x) \ominus B(x)|$

$$\mu_{|A(x) \ominus B(x)|} = \max\{0, \mu_A(X) + \mu_B(X) - 1\}$$

Operations

Equality ($A = B$):

$$\mu_A(X) = \mu_B(X)$$

Power of a fuzzy set A^α :

$$\mu_{A^\alpha}(X) = \{\mu_A(X)\}^\alpha$$

- If $\alpha < 1$, then it is called *dilation*
- If $\alpha > 1$, then it is called *concentration*

Operations

Cartesian Product ($A \times B$):

$$\mu_{A \times B}(x, y) = \min\{\mu_A(x), \mu_B(y)\}$$

Example 3:

$$A(x) = \{(x_1, 0.2), (x_2, 0.3), (x_3, 0.5), (x_4, 0.6)\}$$

$$B(y) = \{(y_1, 0.8), (y_2, 0.6), (y_3, 0.3)\}$$

$$A \times B = \min\{\mu_A(x), \mu_B(y)\} =$$

	y_1	y_2	y_3
x_1	0.2	0.2	0.2
x_2	0.3	0.3	0.3
x_3	0.5	0.5	0.3
x_4	0.6	0.6	0.3

Operations

Commutativity :

$$A \cup B = B \cup A$$

$$A \cap B = B \cap A$$

Associativity :

$$A \cup (B \cup C) = (A \cup B) \cup C$$

$$A \cap (B \cap C) = (A \cap B) \cap C$$

Distributivity :

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

Operations

Idempotence :

$$A \cup A = A$$

$$A \cap A = \emptyset$$

$$A \cup \emptyset = A$$

$$A \cap \emptyset = \emptyset$$

Transitivity :

$$\text{If } A \subseteq B, B \subseteq C \text{ then } A \subseteq C$$

Involution :

$$(A^c)^c = A$$

De Morgan's law :

$$(A \cap B)^c = A^c \cup B^c$$

$$(A \cup B)^c = A^c \cap B^c$$

Operations

Given a membership function of a fuzzy set representing a linguistic hedge, we can derive many more MFs representing several other linguistic hedges using the concept of Concentration and Dilation.

- **Concentration:**

$$A^k = [\mu_A(x)]^k ; k > 1$$

- **Dilation:**

$$A^k = [\mu_A(x)]^k ; k < 1$$

Example : Age = { Young, Middle-aged, Old }

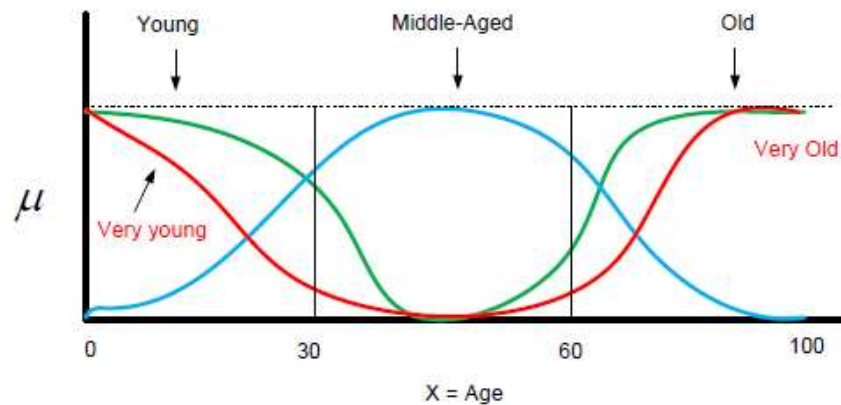
Thus, corresponding to Young, we have : Not young, Very young, Not very young and so on.

Similarly, with Old we can have : old, very old, very very old, extremely old etc.

Thus, Extremely old = $((old)^2)^2$ and so on

Or, More or less old = $A^{0.5} = (old)^{0.5}$

Operations



$$\mu_{\text{young}}(x) = \text{bell}(x, 20, 2, 0) = \frac{1}{1 + (\frac{x}{20})^4}$$

$$\mu_{\text{old}}(x) = \text{bell}(x, 30, 3, 100) = \frac{1}{1 + (\frac{x-100}{30})^6}$$

$$\mu_{\text{middle-aged}} = \text{bell}(x, 30, 60, 50)$$

$$\text{Not young} = \overline{\mu_{\text{young}}(x)} = 1 - \mu_{\text{young}}(x)$$

$$\text{Young but not too young} = \mu_{\text{young}}(x) \cap \overline{\mu_{\text{young}}(x)}$$

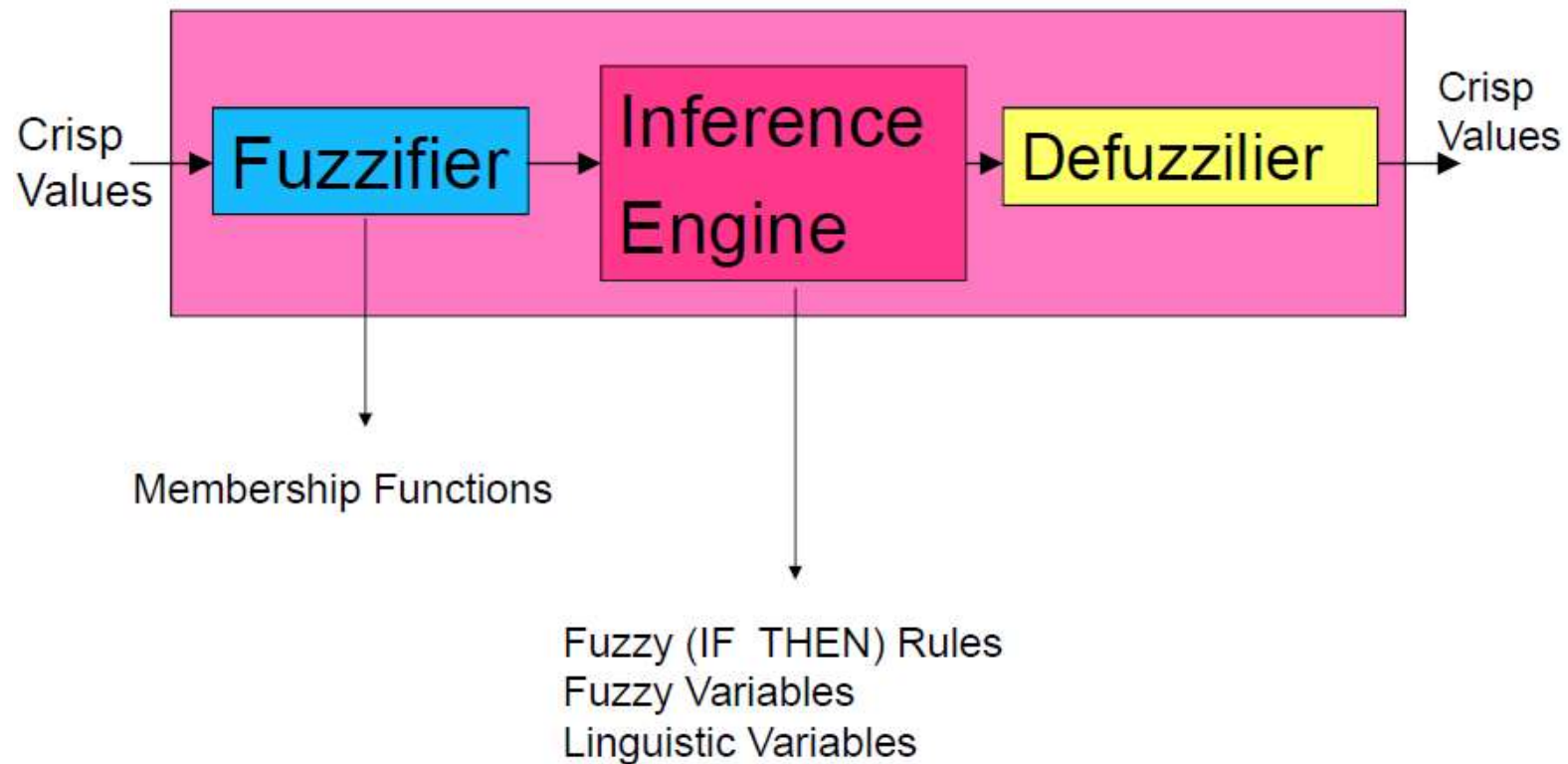
Types

- **Ebrahim Mamdani Fuzzy Models**
- **Sugeno Fuzzy Models**
- **Tsukamoto Fuzzy Models**
- The differences between these three FISs lie in the consequents of their fuzzy rules, and thus their aggregation and defuzzification procedures differ accordingly.

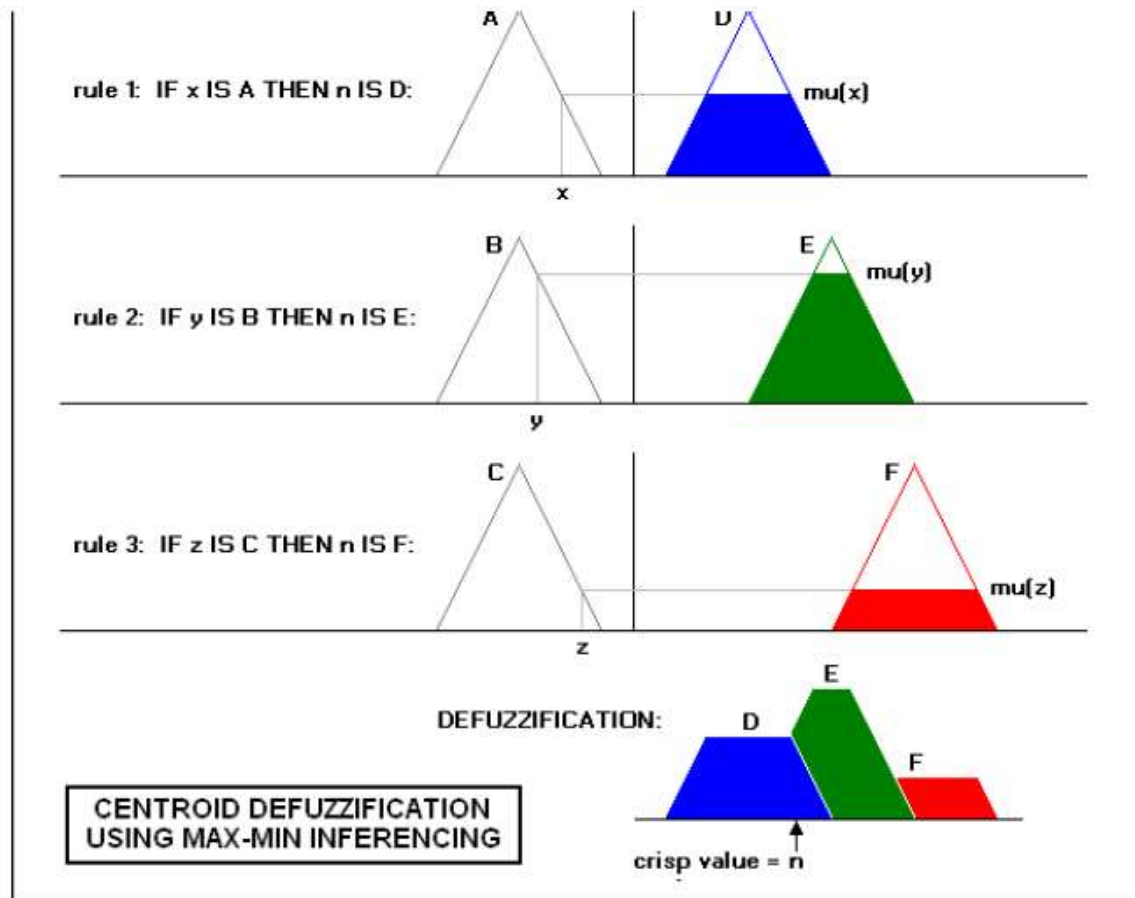
Mamdani Fuzzy Model

- The most commonly used fuzzy inference technique is the so-called **Mamdani** method.
- In 1975, Professor Ebrahim Mamdani of London University built one of the first fuzzy systems to control a steam engine and boiler combination. He applied a set of fuzzy rules supplied by experienced human operators.
- The Mamdani-style fuzzy inference process is performed in four steps:
 1. Fuzzification of the input variables
 2. Rule evaluation (inference)
 3. Aggregation of the rule outputs (composition)
 4. Defuzzification

Mamdani Fuzzy Model



Mamdani Fuzzy Model

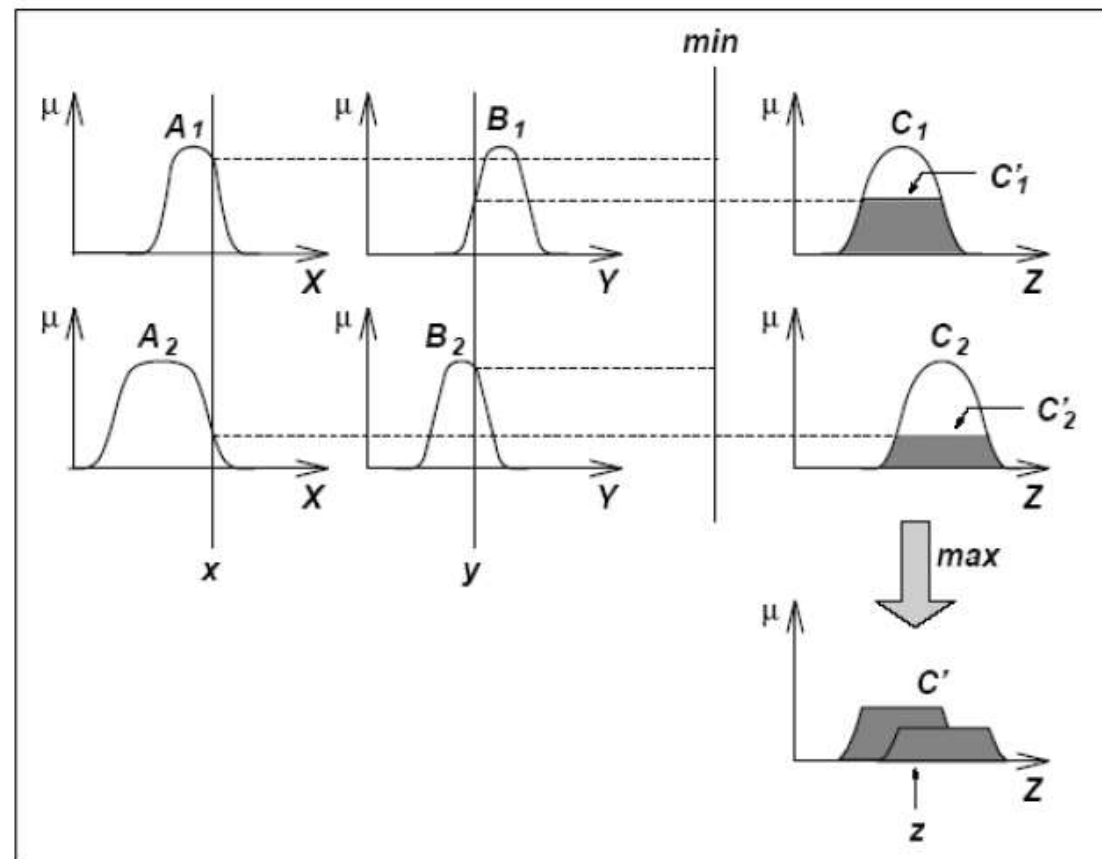


Mamdani composition of three SISO fuzzy outputs

http://en.wikipedia.org/wiki/Fuzzy_control_system

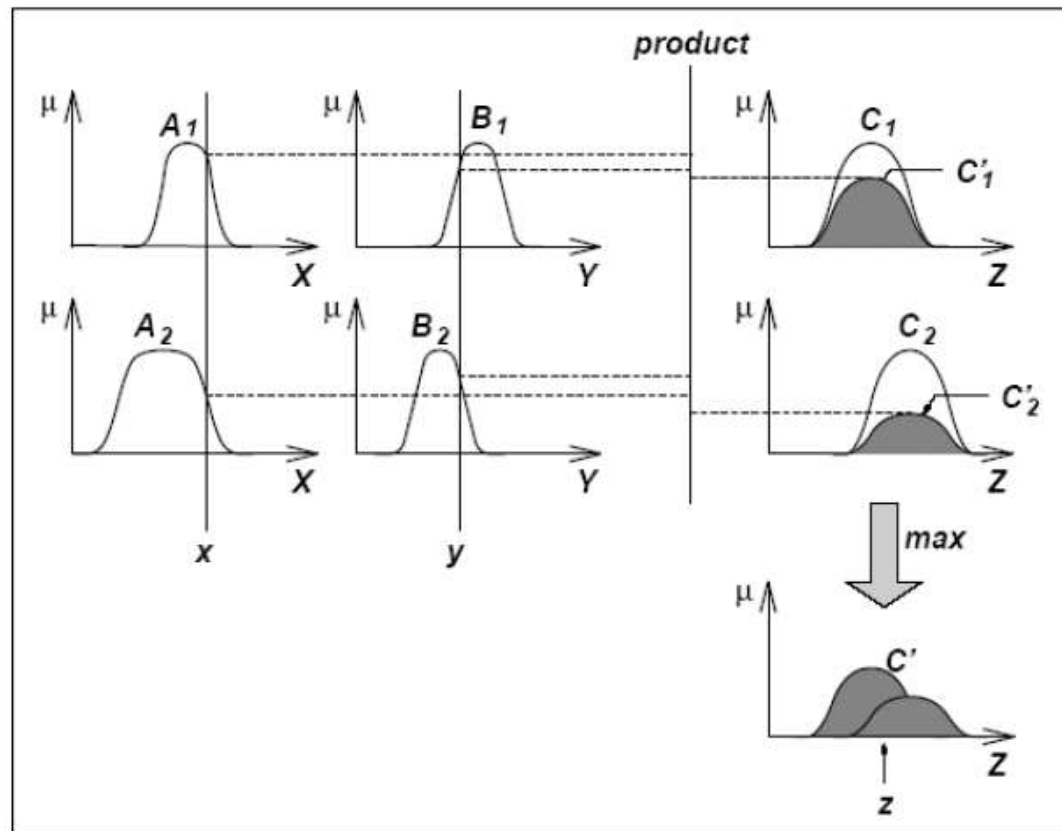
Mamdani Fuzzy Model

The mamdani FIS using **min** and **max** for **T-norm** and **S-norm** and subject to two crisp inputs x and y

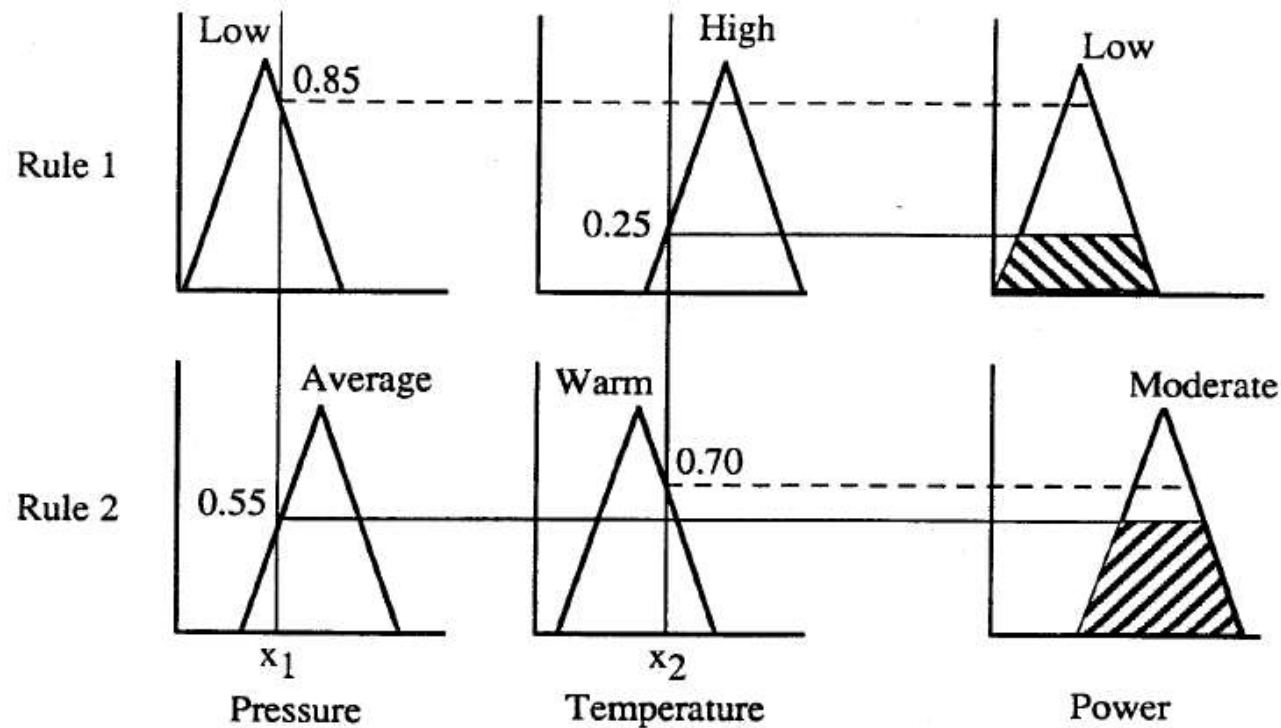


Mamdani Fuzzy Model

The mamdani FIS using **product** and **max** for **T-norm** and **S-norm** and subject to two crisp inputs x and y



Mamdani Fuzzy Model

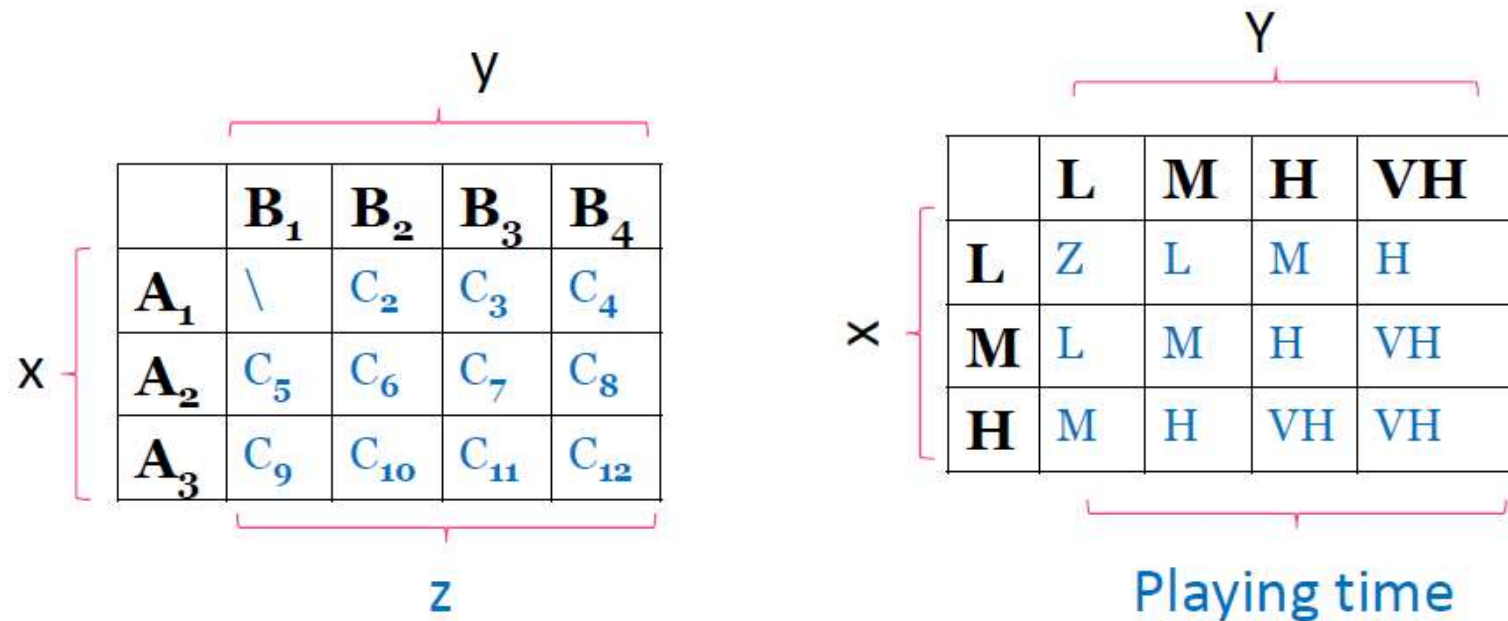


Rule 1: If pressure is low and temperature is high then power is low
Rule 2: If pressure is average and temperature is warm
then power is moderate

Mamdani Fuzzy Model

Two-input, one-output example:

If x is A_i and y is B_k then z is $C_{m(i,k)}$



Mamdani Fuzzy Model

- In many applications we have to use crisp values as inputs for controlling of machines and systems.
- So, we have to use a defuzzifier to convert a fuzzy set to a crisp value.

Mamdani Fuzzy Model

- Defuzzification refers to the way a crisp value is extracted from a fuzzy set as a representative value.
- Defuzzification Methods:
 - Centroid of Area
 - Bisector of Area
 - Mean of Max
 - Smallest of Max
 - Largest of Max

Mamdani Fuzzy Model

$$z_{\text{COA}} = \frac{\int_Z \mu_A(z) z \, dz}{\int_Z \mu_A(z) \, dz},$$

- where μ_A is aggregated output MF.
- This is the most widely adopted defuzzification strategy, which is reminiscent of the calculation of expected values of probability distributions.

Mamdani Fuzzy Model

- z_{BOA} satisfies

$$\int_{\alpha}^{z_{\text{BOA}}} \mu_A(z) dz = \int_{z_{\text{BOA}}}^{\beta} \mu_A(z) dz,$$

$$\alpha = \min\{z | z \in Z\} \quad \beta = \max\{z | z \in Z\}$$

- That is, the vertical line $z = z_{\text{BOA}}$ partitions the region between $z = \alpha$, $z = \beta$, $y = 0$ and $y = \mu_A(z)$ into two regions with the same area.

Mamdani Fuzzy Model

- z_{MOM} is the mean of maximizing z at which the MF reaches maximum μ^* . In Symbols,

$$z_{\text{MOM}} = \frac{\int_{Z'} z \, dz}{\int_{Z'} dz},$$

$$Z' = \{z | \mu_A(z) \in \mu^*\}$$

- In particular, if $\mu_A(z)$ has a single maximum at $z = z^*$, then the $z_{\text{MOM}} = z^*$.
- Moreover, if $\mu_A(z)$ reaches its maximum whenever

$$z \in [z_{\text{left}}, z_{\text{right}}]$$

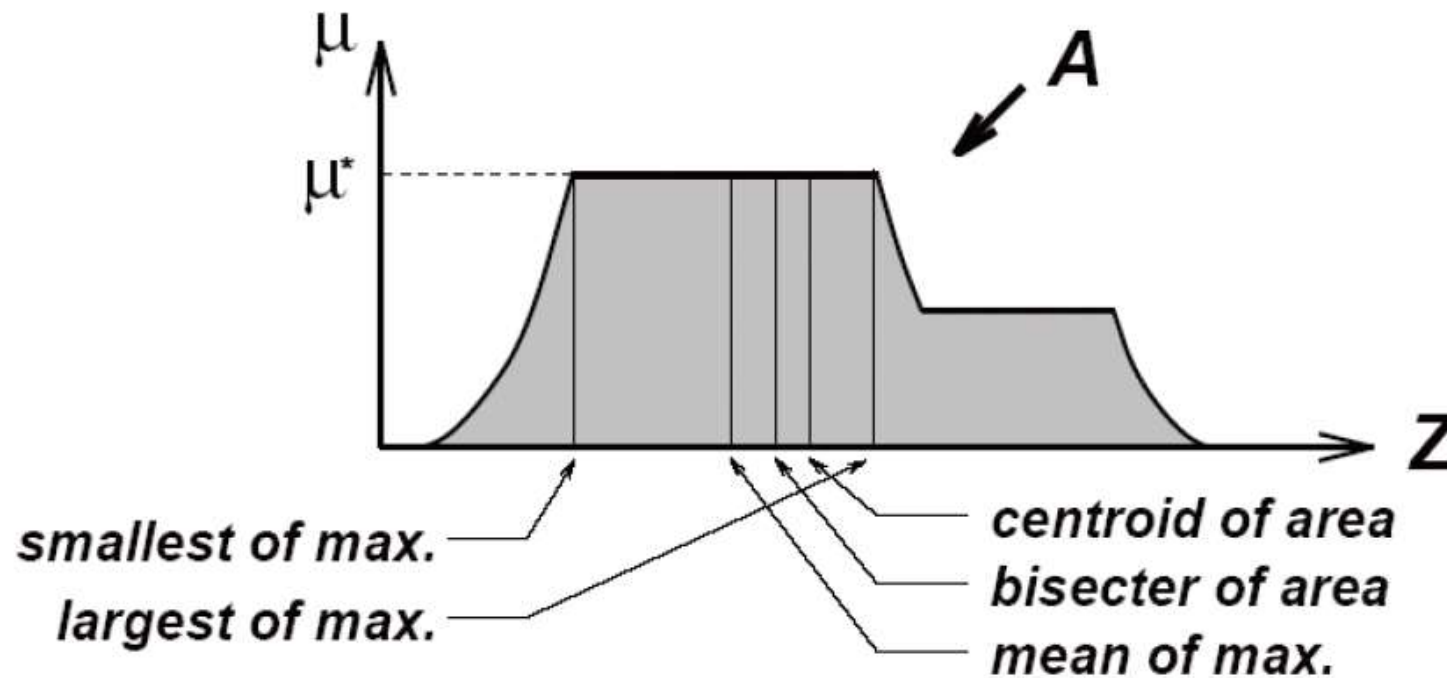
then

$$z_{\text{MOM}} = (z_{\text{left}} + z_{\text{right}})/2$$

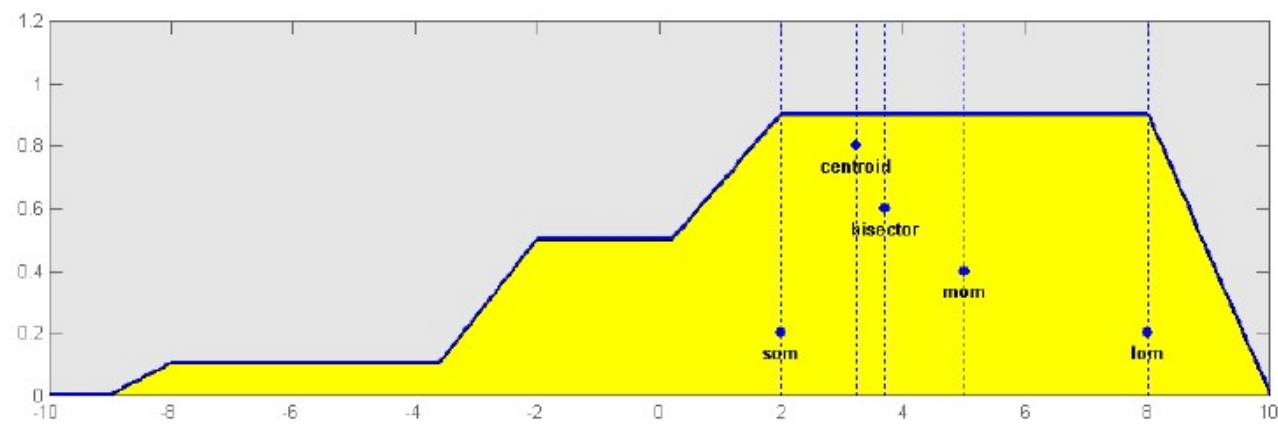
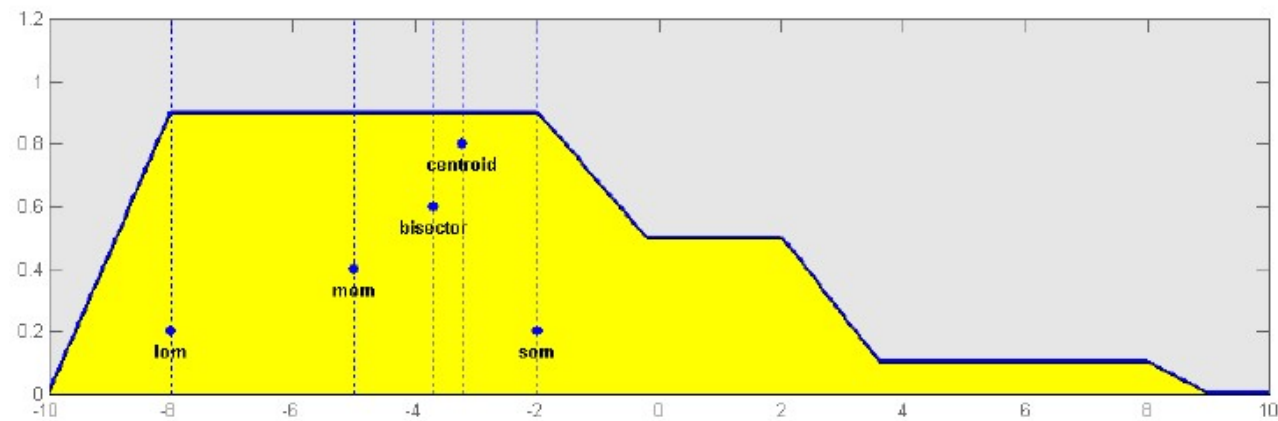
Mamdani Fuzzy Model

- z_{SOM} is the minimum (in terms of magnitude) of the maximizing z .
- z_{LOM} is the maximum (in terms of magnitude) of the maximizing z .
- Because of their obvious bias, z_{SOM} and z_{LOM} are not used as often as the other three defuzzification methods.

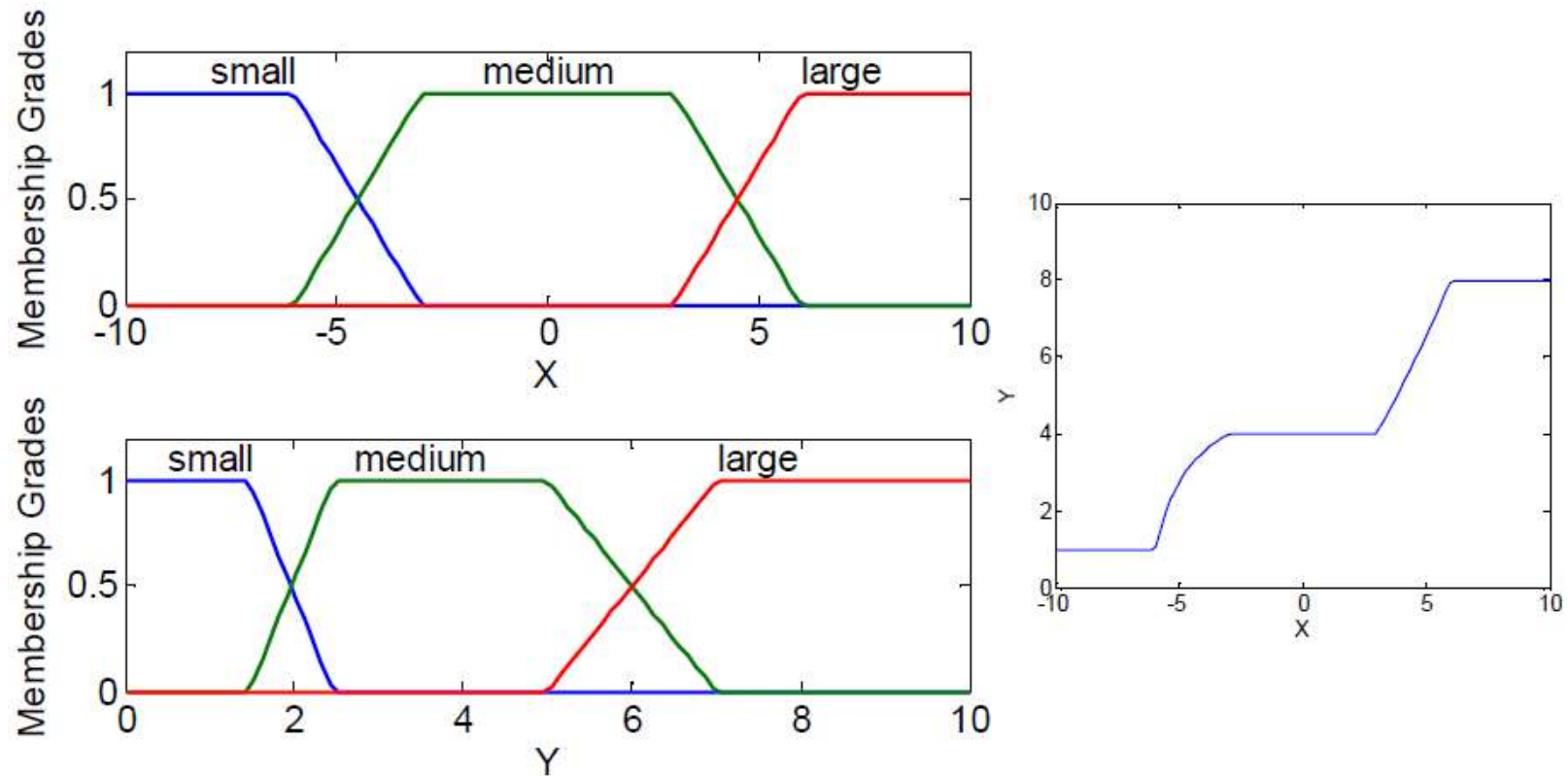
Mamdani Fuzzy Model



Mamdani Fuzzy Model

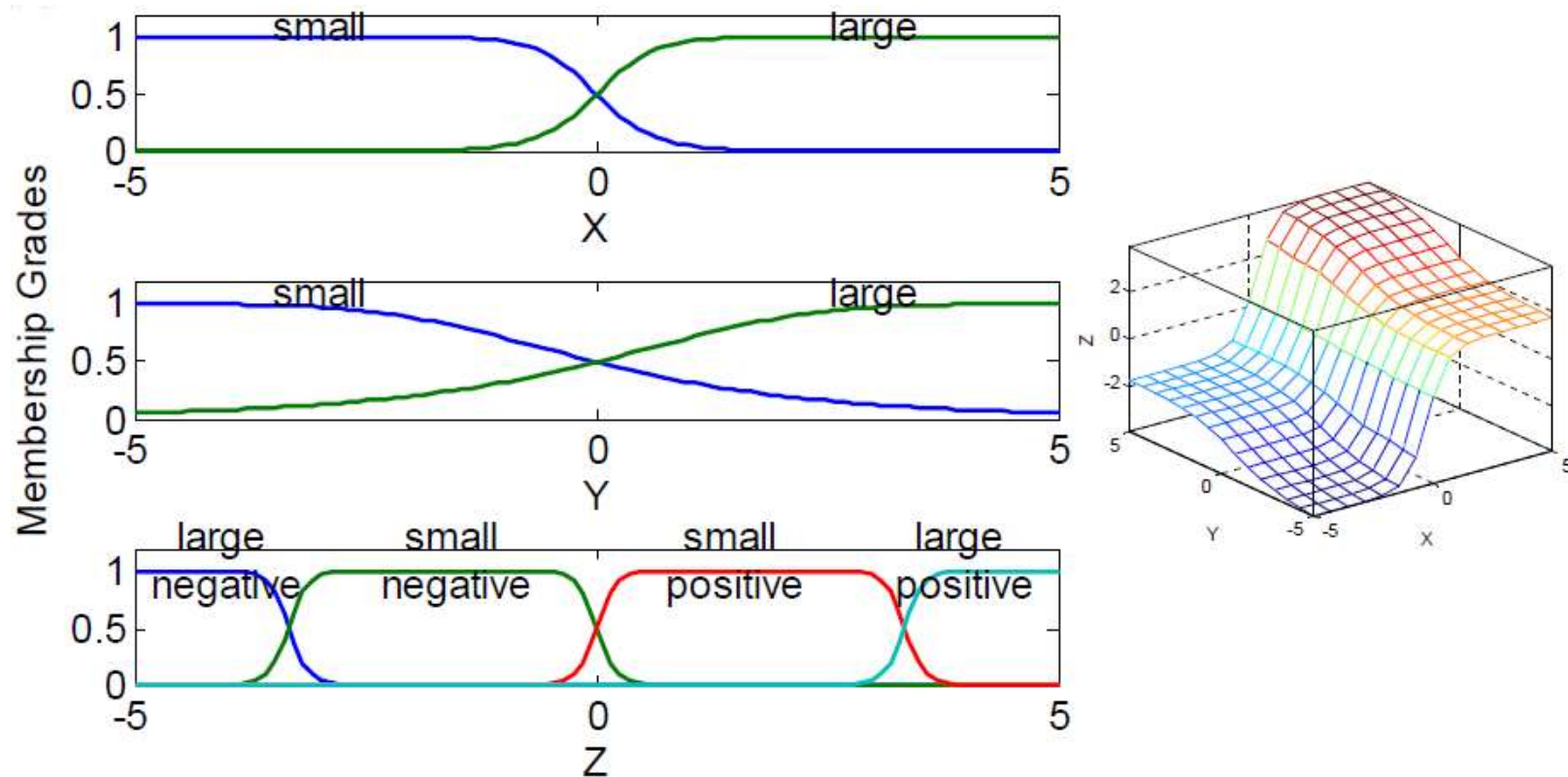


Mamdani Fuzzy Model



- a) MFs of the input and output
- b) Overall input-output curve

Mamdani Fuzzy Model



- a) MFs of the inputs and output
- b) Overall input-output curve

Mamdani Fuzzy Model

We examine a simple two-input one-output problem that includes three rules:

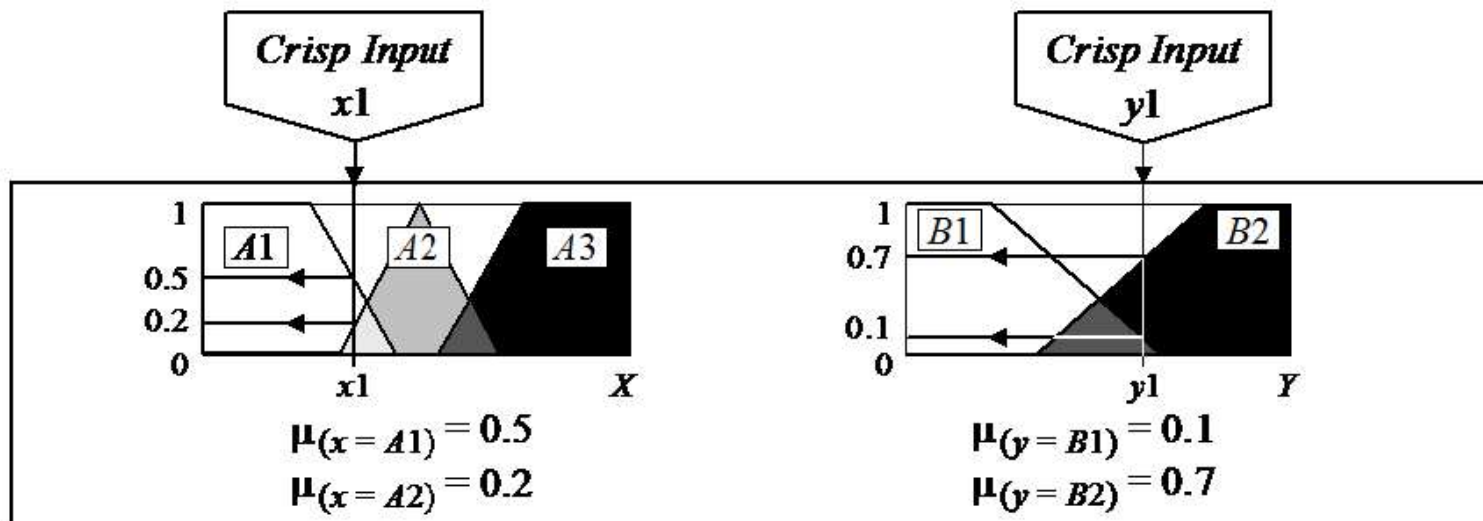
<u>Rule: 1</u>	IF x is A3	OR	y is B1	THEN	z is C1
<u>Rule: 2</u>	IF x is A2	AND	y is B2	THEN	z is C2
<u>Rule: 3</u>	IF x is A1			THEN	z is C3

Real-life example for these kinds of rules:

<u>Rule: 1</u>	IF project_funding is adequate	OR	project_staffing is small	THEN	risk is low
<u>Rule: 2</u>	IF project_funding is marginal	AND	project_staffing is large	THEN	risk is normal
<u>Rule: 3</u>	IF project_funding is inadequate			THEN	risk is high

Mamdani Fuzzy Model

- The first step is to take the crisp inputs, $x1$ and $y1$ (*project funding* and *project staffing*), and determine the degree to which these inputs belong to each of the appropriate fuzzy sets.



Mamdani Fuzzy Model

- The second step is to take the fuzzified inputs, $\mu_{(x=A1)} = 0.5$, $\mu_{(x=A2)} = 0.2$, $\mu_{(y=B1)} = 0.1$ and $\mu_{(y=B2)} = 0.7$, and apply them to the antecedents of the fuzzy rules.
- If a given fuzzy rule has multiple antecedents, the fuzzy operator (AND or OR) is used to obtain a single number that represents the result of the antecedent evaluation.

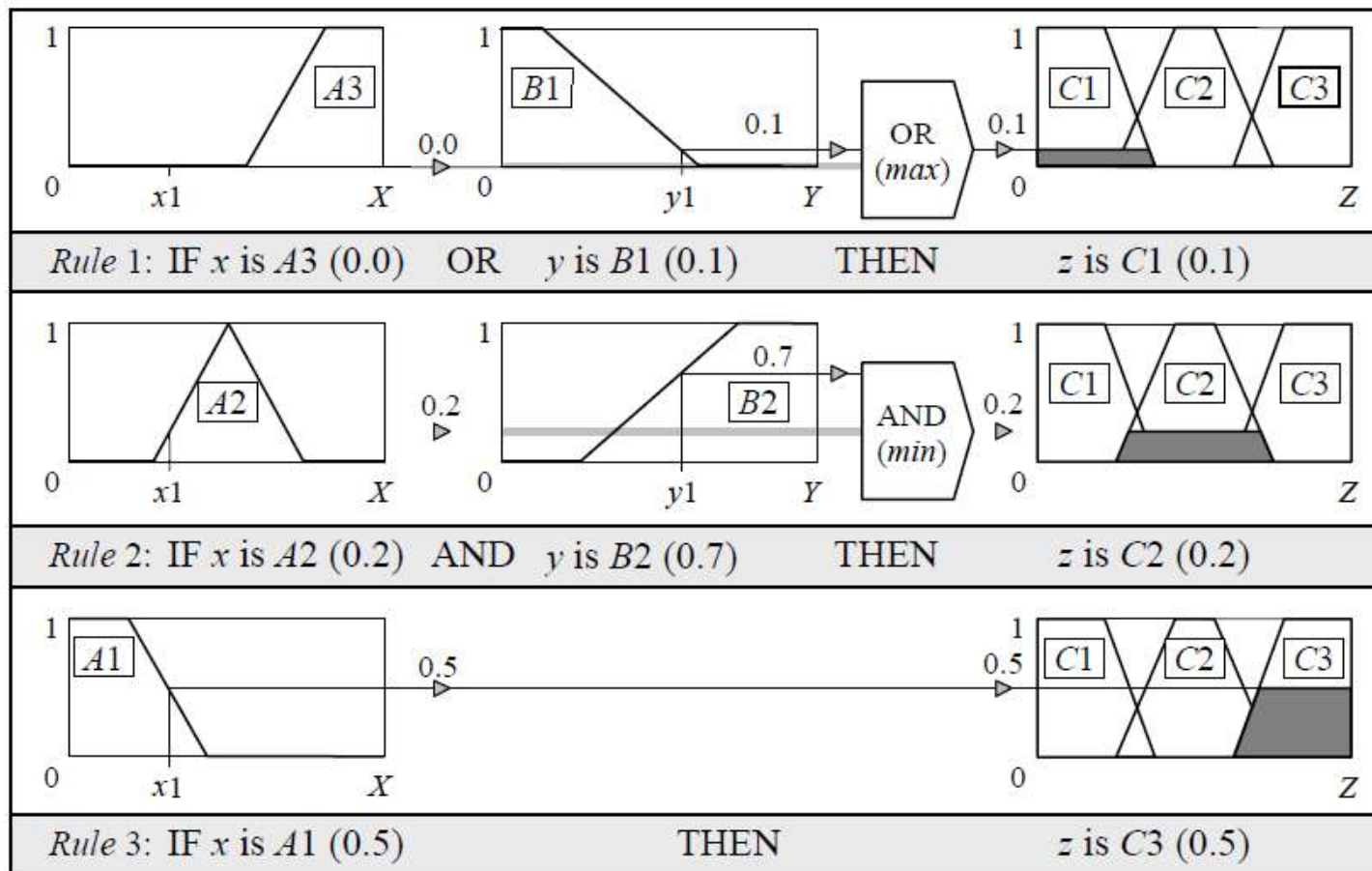
RECALL: To evaluate the disjunction of the rule antecedents, we use the **OR** fuzzy operation. Typically, fuzzy expert systems make use of the classical fuzzy operation union:

$$\mu_{A \cup B}(x) = \max [\mu_A(x), \mu_B(x)]$$

Similarly, in order to evaluate the conjunction of the rule antecedents, we apply the **AND** fuzzy operation intersection:

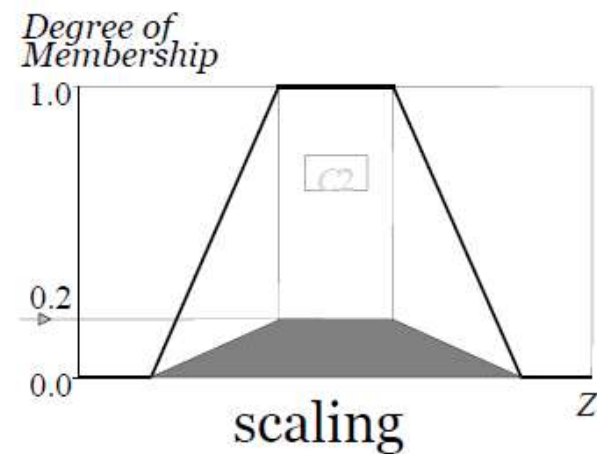
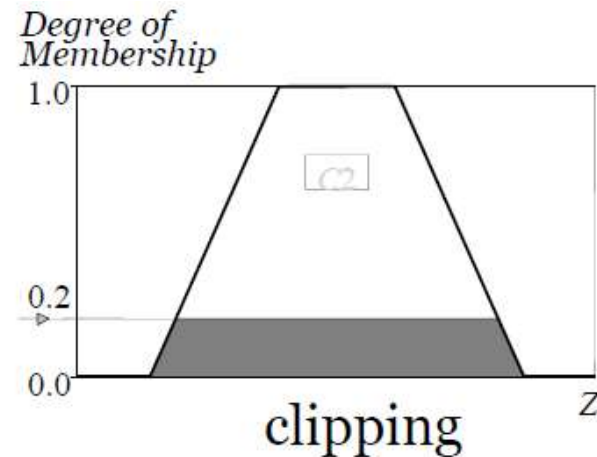
$$\mu_{A \cap B}(x) = \min [\mu_A(x), \mu_B(x)]$$

Mamdani Fuzzy Model



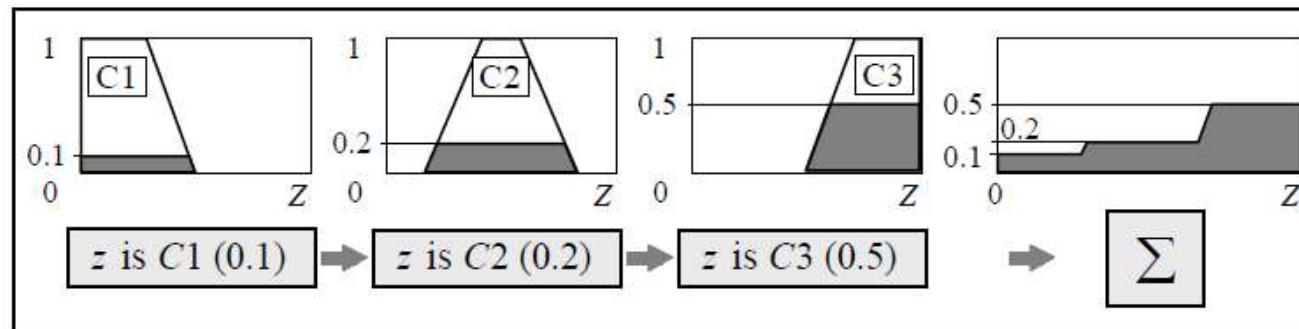
Mamdani Fuzzy Model

- Now the result of the antecedent evaluation can be applied to the membership function of the consequent.
- The most common method is to cut the consequent membership function at the level of the antecedent truth. This method is called **clipping** (alpha-cut).
 - Since the top of the membership function is sliced, the clipped fuzzy set loses some information.
 - However, clipping is still often preferred because it involves less complex and faster mathematics, and generates an aggregated output surface that is easier to defuzzify.
- While clipping is a frequently used method, **scaling** offers a better approach for preserving the original shape of the fuzzy set.
 - The original membership function of the rule consequent is adjusted by multiplying all its membership degrees by the truth value of the rule antecedent.
 - This method, which generally loses less information, can be very useful in fuzzy expert systems.



Mamdani Fuzzy Model

- Aggregation is the process of unification of the outputs of all rules.
- We take the membership functions of all rule consequents previously clipped or scaled and combine them into a single fuzzy set.
- The input of the aggregation process is the list of clipped or scaled consequent membership functions, and the output is one fuzzy set for each output variable.



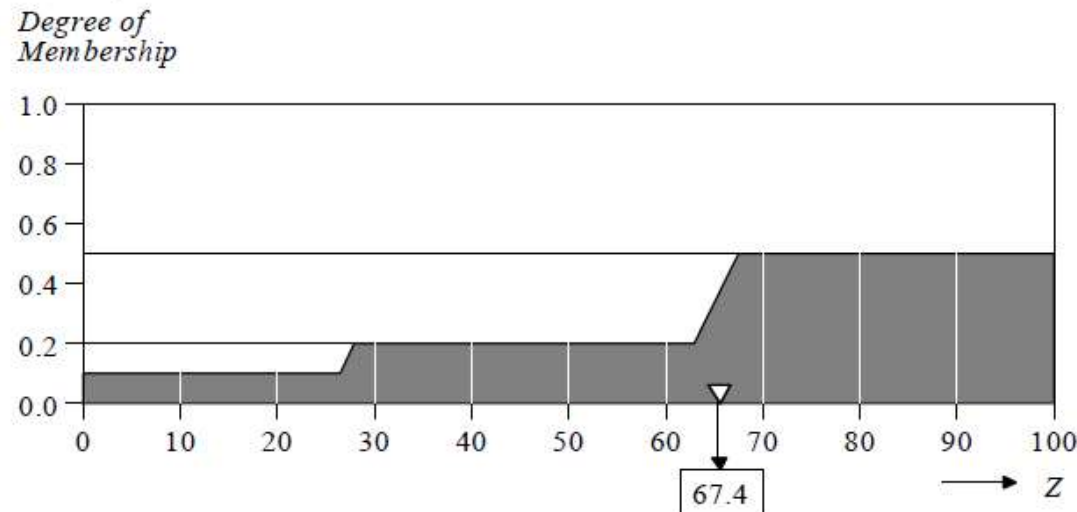
Mamdani Fuzzy Model

- The last step in the fuzzy inference process is defuzzification.
- Fuzziness helps us to evaluate the rules, but the final output of a fuzzy system has to be a crisp number.
- The input for the defuzzification process is the aggregate output fuzzy set and the output is a single number.
- There are several defuzzification methods, but probably the most popular one is the **centroid technique**. It finds the point where a vertical line would slice the aggregate set into two equal masses. Mathematically this **centre of gravity (COG)** can be expressed as:

$$COG = \frac{\int_a^b \mu_A(x) x dx}{\int_a^b \mu_A(x) dx}$$

Mamdani Fuzzy Model

- Centroid defuzzification method finds a point representing the centre of gravity of the aggregated fuzzy set A , on the interval $[a, b]$.
- A reasonable estimate can be obtained by calculating it over a sample of points.



$$COG = \frac{(0+10+20) \times 0.1 + (30+40+50+60) \times 0.2 + (70+80+90+100) \times 0.5}{0.1+0.1+0.1+0.2+0.2+0.2+0.2+0.5+0.5+0.5+0.5} = 67.4$$

Sugeno Fuzzy Inference

- Mamdani-style inference, as we have just seen, requires us to find the centroid of a two-dimensional shape by integrating across a continuously varying function. In general, this process is not computationally efficient.
- Michio Sugeno suggested to use a single spike, a singleton, as the membership function of the rule consequent.
- A singleton, or more precisely a fuzzy singleton, is a fuzzy set with a membership function that is unity at a single particular point on the universe of discourse and zero everywhere else.

Sugeno Fuzzy Inference

- Also known as the TSK fuzzy model (proposed by Takagi, Sugeno, and Kang)
- For developing a systematic approach to generating fuzzy rules from a given input-output data set
- A typical fuzzy rule in a Sugeno fuzzy model:
if x is A and y is B then $z = f(x, y)$
- A and B : fuzzy sets
- $z = f(x, y)$: a crisp function (usually polynomial in the input variables x and y)

Sugeno Fuzzy Inference

- Sugeno-style fuzzy inference is very similar to the Mamdani method.
- Sugeno changed only a rule consequent: instead of a fuzzy set, he used a mathematical function of the input variable.
- The format of the **Sugeno-style fuzzy rule** is

IF x is A AND y is B THEN z is $f(x, y)$

where:

- x, y and z are linguistic variables;
 - A and B are fuzzy sets on universe of discourses X and Y , respectively;
 - $f(x, y)$ is a mathematical function.
- The most commonly used **zero-order Sugeno fuzzy model** applies fuzzy rules in the following form:

IF x is A AND y is B THEN z is k

- where k is a constant.
- In this case, the output of each fuzzy rule is constant and all consequent **membership functions** are represented by **singleton spikes**.

Sugeno Fuzzy Inference

- **First-order Sugeno fuzzy model:** $f(x, y)$ is a *first-order polynomial*
- **zero-order Sugeno fuzzy model:** f is a constant
 - a special case of the Mamdani fuzzy inference system, in which each rule's consequent is specified by a fuzzy singleton;
 - or a special case of the Tsukamoto fuzzy model (to be introduced next) in which each rule's consequent is specified by an MF of a step function center at the constant

Sugeno Fuzzy Inference

The output is a weighted average:

$$z = \frac{\sum \mu_{A_i, B_k}(x, y) f_{m(i, k)}(x, y)}{\sum \mu_{A_i, B_k}(x, y)}$$

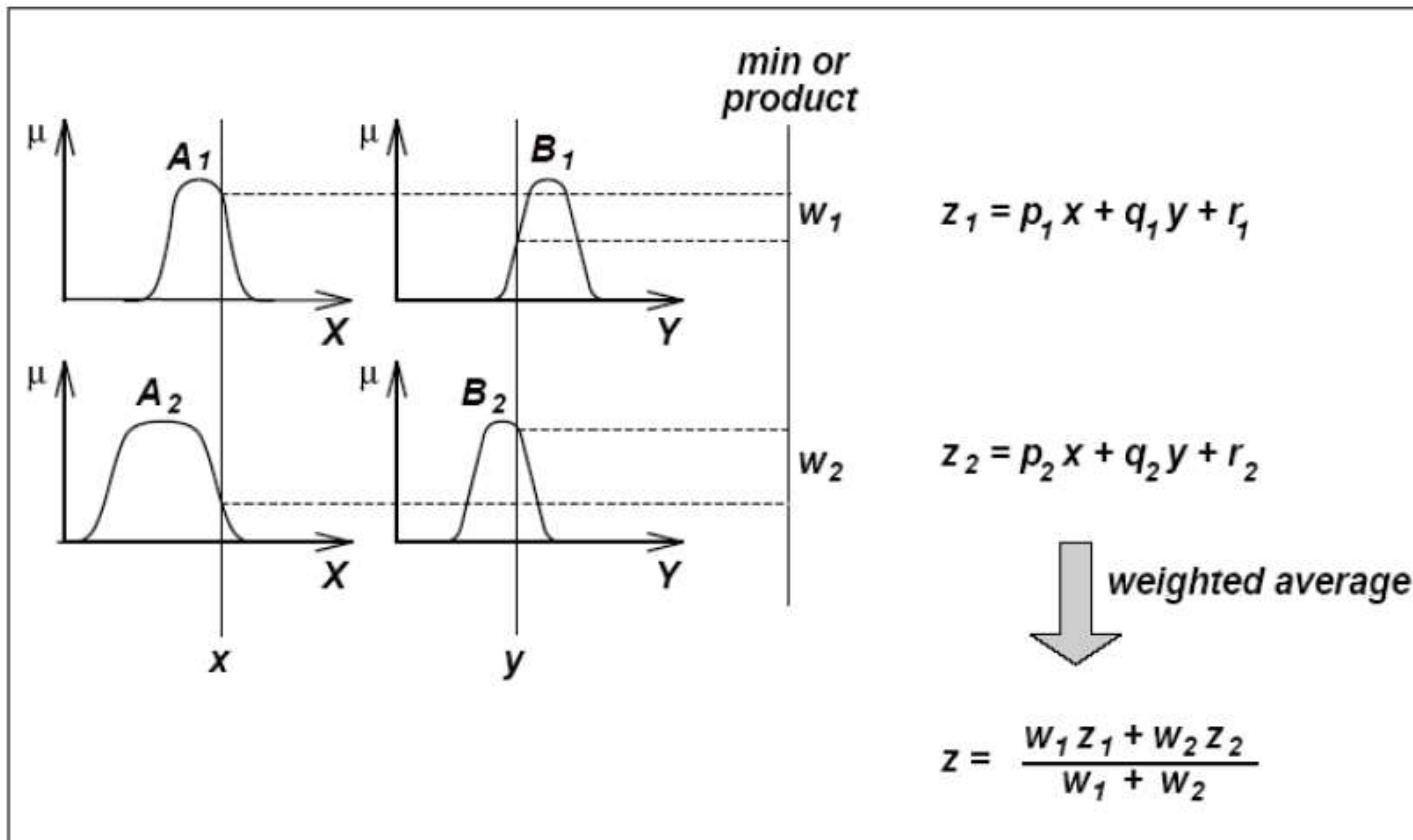
Double summation
over all i (x MFs) and
all k (y MFs)

$$= \frac{\sum w_i f_i(x, y)}{\sum w_i}$$

Summation over all i
(fuzzy rules)

where w_i is the firing strength of the i -th output

Sugeno Fuzzy Inference

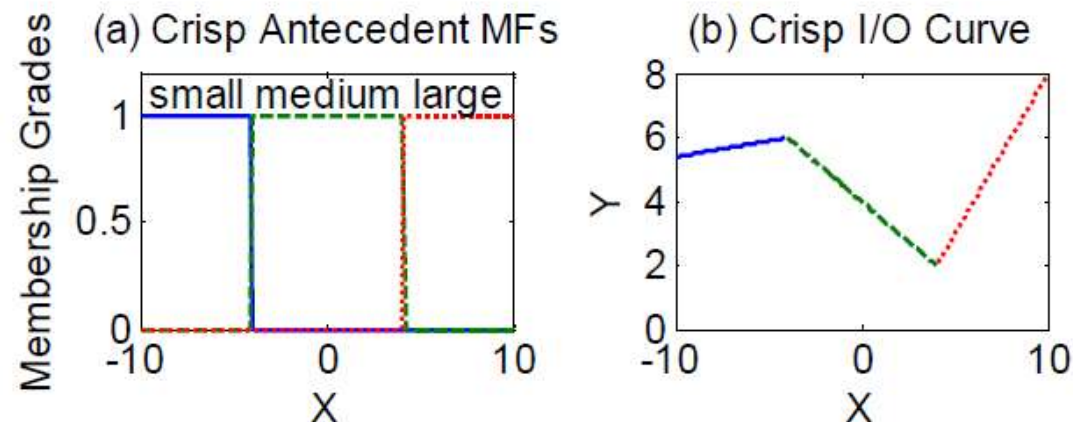


Sugeno Fuzzy Inference

- An example of a single-input Sugeno fuzzy model:
 - If X is small then $Y = 0.1X + 6.4$.
 - If X is medium then $Y = -0.5X + 4$.
 - If X is large then $Y = X - 2$.

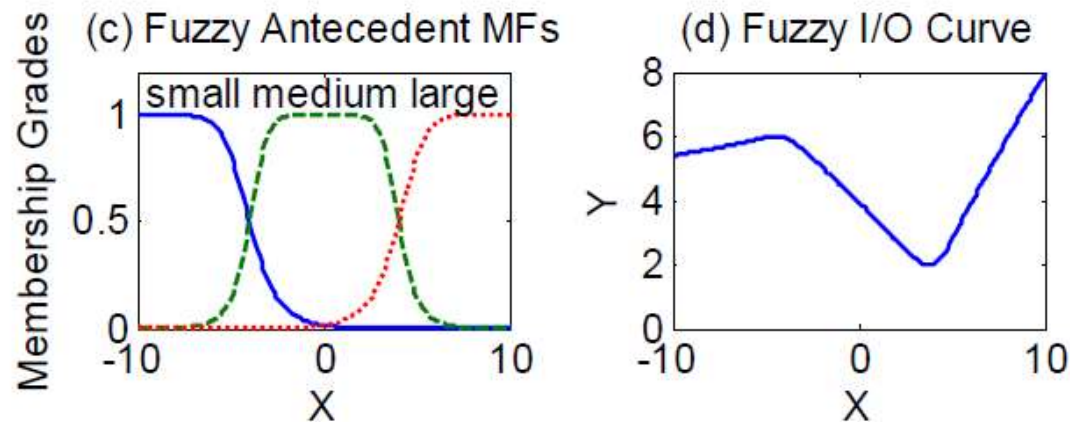
Sugeno Fuzzy Inference

- If "small," "medium," and "large" are nonfuzzy sets with membership functions shown in figure (a), then the overall input-output curve is piecewise linear, as shown in figure (b):



Sugeno Fuzzy Inference

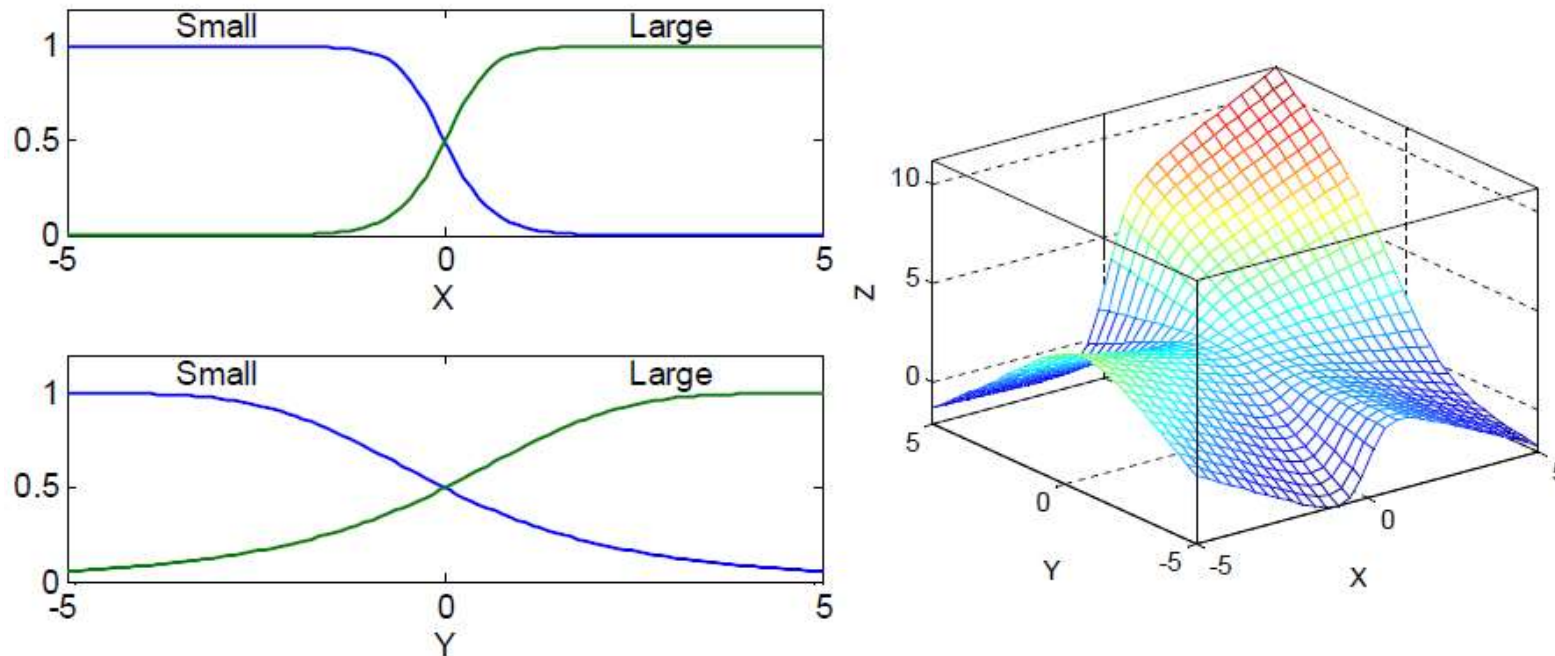
- If we have smooth membership functions [figure (c)] instead, the overall input-output curve [figure (d)] becomes a smoother one:



Sugeno Fuzzy Inference

- An example of a two-input single-output Sugeno fuzzy model with four rules:
 - If X is small and Y is small then $z = -x + y + 1$.
 - If X is small and Y is large then $z = -y + 3$.
 - If X is large and Y is small then $z = -x + 3$.
 - If X is large and Y is large then $z = x + y + 2$.

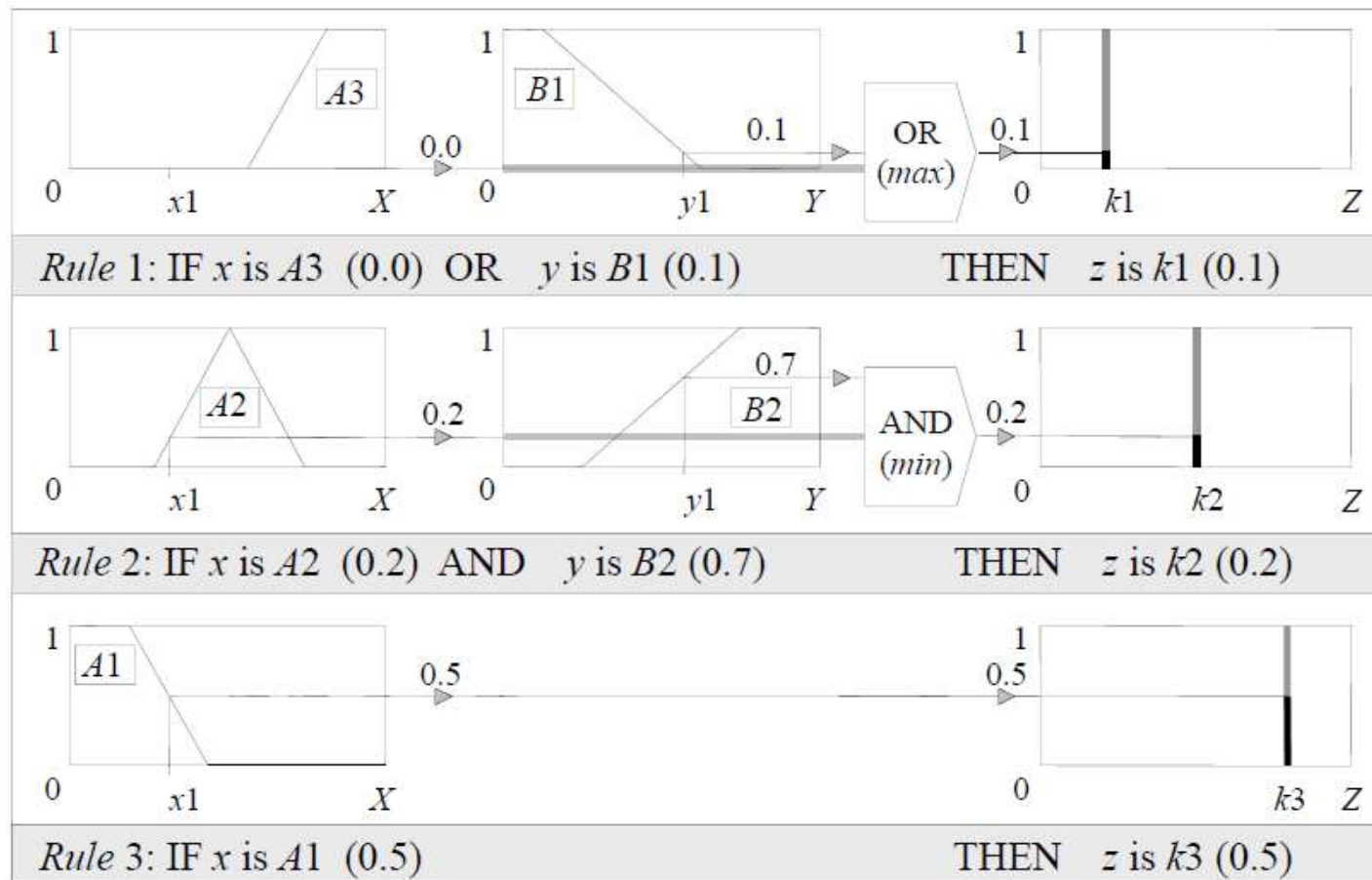
Sugeno Fuzzy Inference



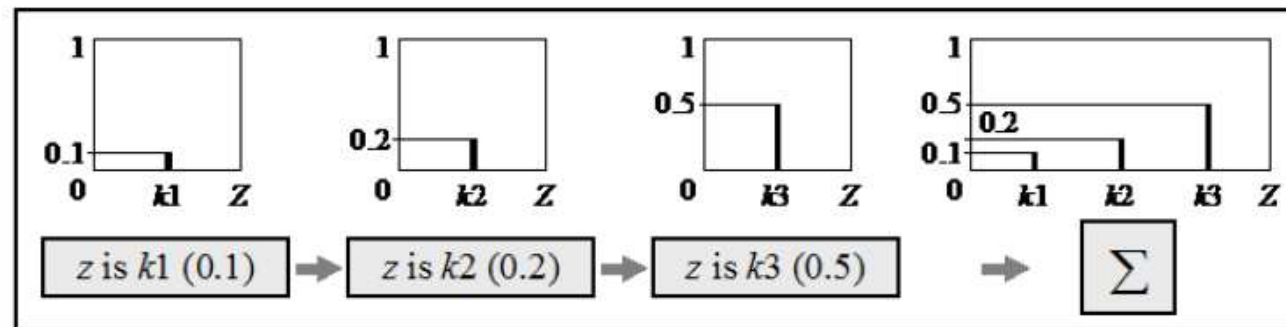
- a) MFs of the inputs and output
- b) Overall input-output curve

- The surface is composed of four planes, each of which is specified by the output equation of a fuzzy rule.

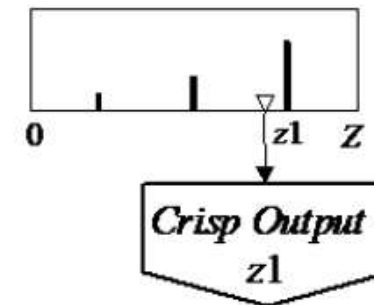
Sugeno Fuzzy Inference



Sugeno Fuzzy Inference



COG becomes Weighted Average (WA)



$$WA = \frac{\mu(k1) \times k1 + \mu(k2) \times k2 + \mu(k3) \times k3}{\mu(k1) + \mu(k2) + \mu(k3)} = \frac{0.1 \times 20 + 0.2 \times 50 + 0.5 \times 80}{0.1 + 0.2 + 0.5} = 65$$

Sugeno Fuzzy Inference

- Unlike the Mamdani fuzzy model, the Sugeno fuzzy model cannot follow the compositional rule of inference strictly in its fuzzy reasoning mechanism
- Without the time-consuming and mathematically intractable defuzzification operation, the Sugeno fuzzy model is by far the most popular candidate for sample data-based fuzzy modeling (we will see an application in ANFIS)

Sugeno Fuzzy Inference

- Mamdani method is widely accepted for capturing expert knowledge. It allows us to describe the expertise in more intuitive, more human-like manner. However, Mamdani-type fuzzy inference entails a substantial computational burden.
- On the other hand, Sugeno method is computationally effective and works well with optimisation and adaptive techniques, which makes it very attractive in adaptive problems, particularly for dynamic nonlinear systems.

Building a Fuzzy System

- A service centre keeps spare parts and repairs parts.
- A customer brings a failed item and receives a spare of the same type.
- Failed parts are repaired by **servers**, placed on the shelf, and thus become spares.
- The objective here is to advise a manager of the service centre on certain decision policies to keep the customers satisfied.
- Advise on the initial number of spares to keep **delay** reasonable

From: <http://www2.cs.siu.edu/~rahimi>

Building a Fuzzy System

There are four main linguistic variables: average waiting time (mean delay) m , repair utilisation factor of the service centre ρ , number of servers s , and initial number of spare parts n .

$$\rho = \frac{\text{CustomerArrivalRate}}{\text{CustomerDepartureRate}}$$

The system must advise management on the number of spares to keep as well as the number of servers. Increasing either will increase cost and decrease waiting time in some proportion.

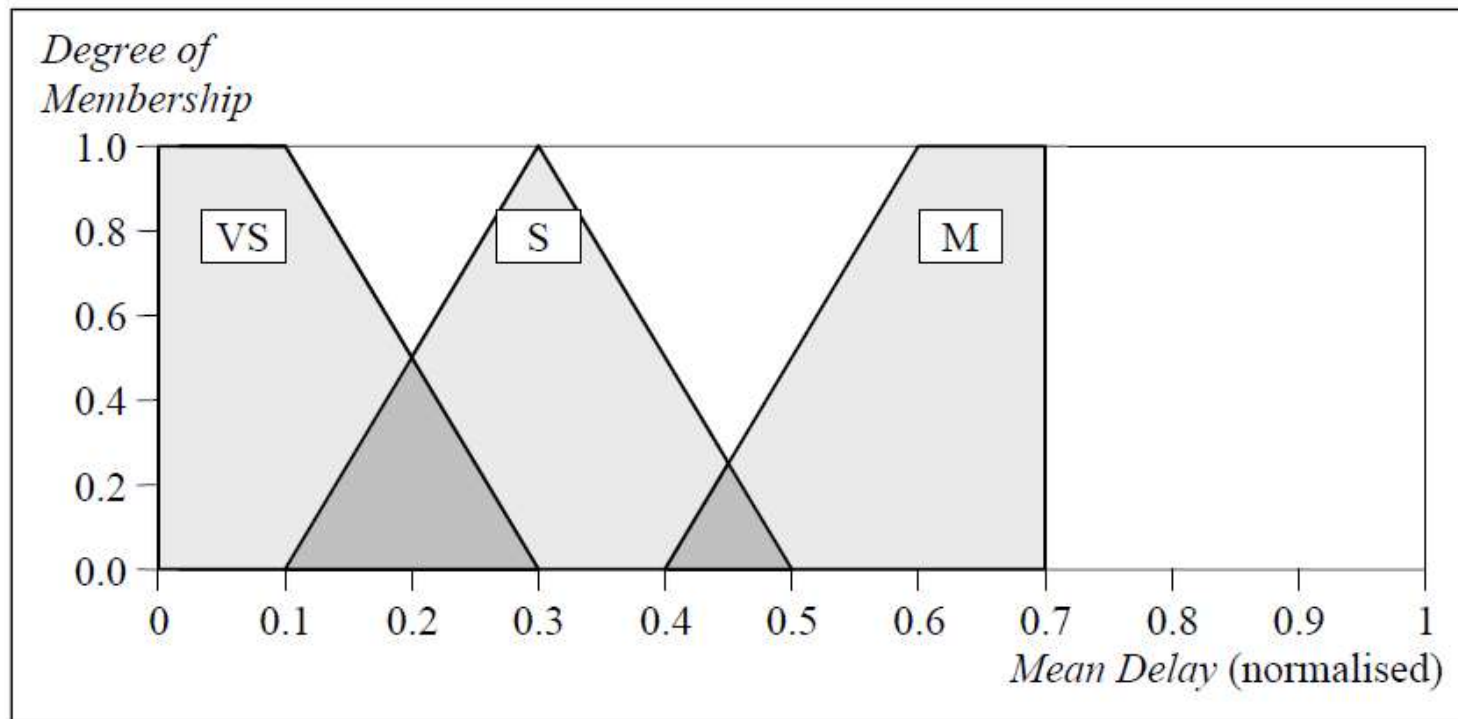
Building a Fuzzy System

Linguistic Variable: <i>Mean Delay, m</i>		
Linguistic Value	Notation	Numerical Range (normalised)
Very Short	VS	[0, 0.3]
Short	S	[0.1, 0.5]
Medium	M	[0.4, 0.7]
Linguistic Variable: <i>Number of Servers, s</i>		
Linguistic Value	Notation	Numerical Range (normalised)
Small	S	[0, 0.35]
Medium	M	[0.30, 0.70]
Large	L	[0.60, 1]
Linguistic Variable: <i>Repair Utilisation Factor, ρ</i>		
Linguistic Value	Notation	Numerical Range
Low	L	[0, 0.6]
Medium	M	[0.4, 0.8]
High	H	[0.6, 1]
Linguistic Variable: <i>Number of Spares, n</i>		
Linguistic Value	Notation	Numerical Range (normalised)
Very Small	VS	[0, 0.30]
Small	S	[0, 0.40]
Rather Small	RS	[0.25, 0.45]
Medium	M	[0.30, 0.70]
Rather Large	RL	[0.55, 0.75]
Large	L	[0.60, 1]
Very Large	VL	[0.70, 1]

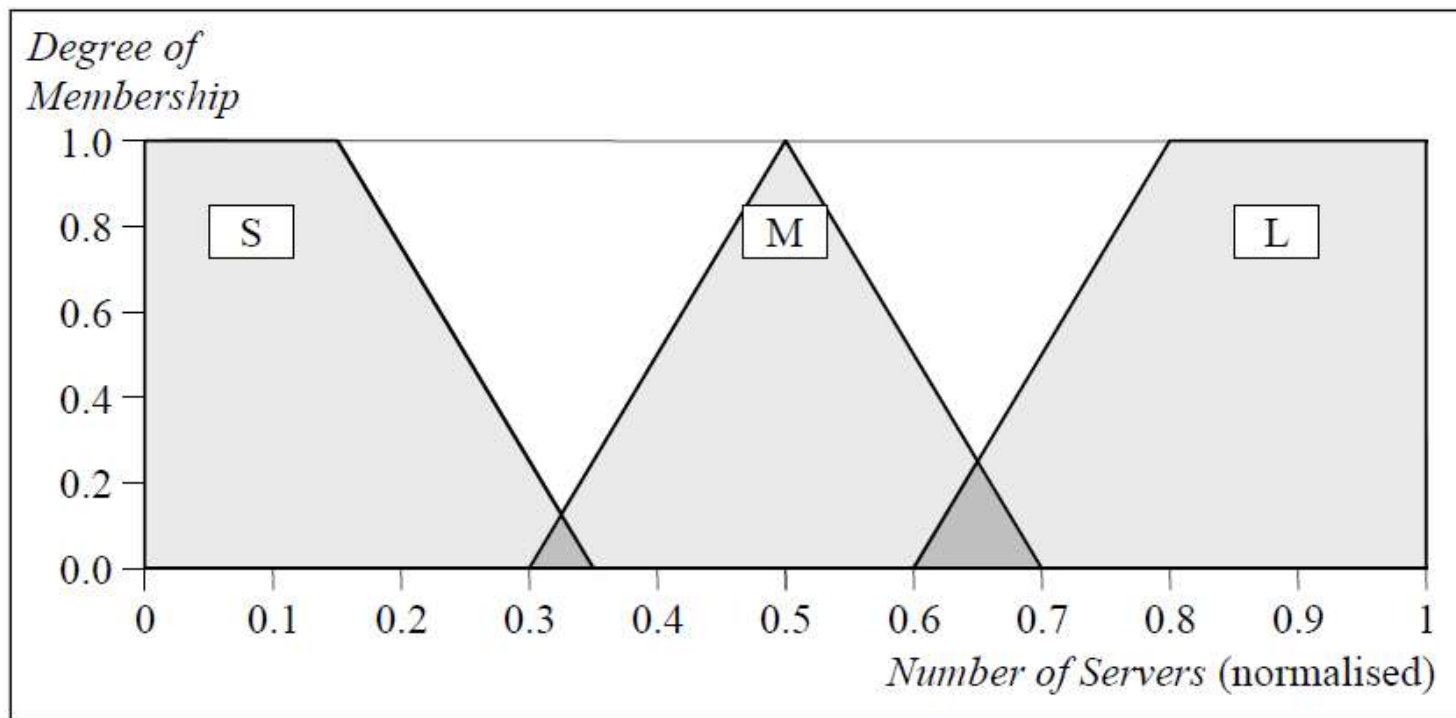
Building a Fuzzy System

Fuzzy sets can have a variety of shapes. However, a triangle or a trapezoid can often provide an adequate representation of the expert knowledge, and at the same time, significantly simplifies the process of computation.

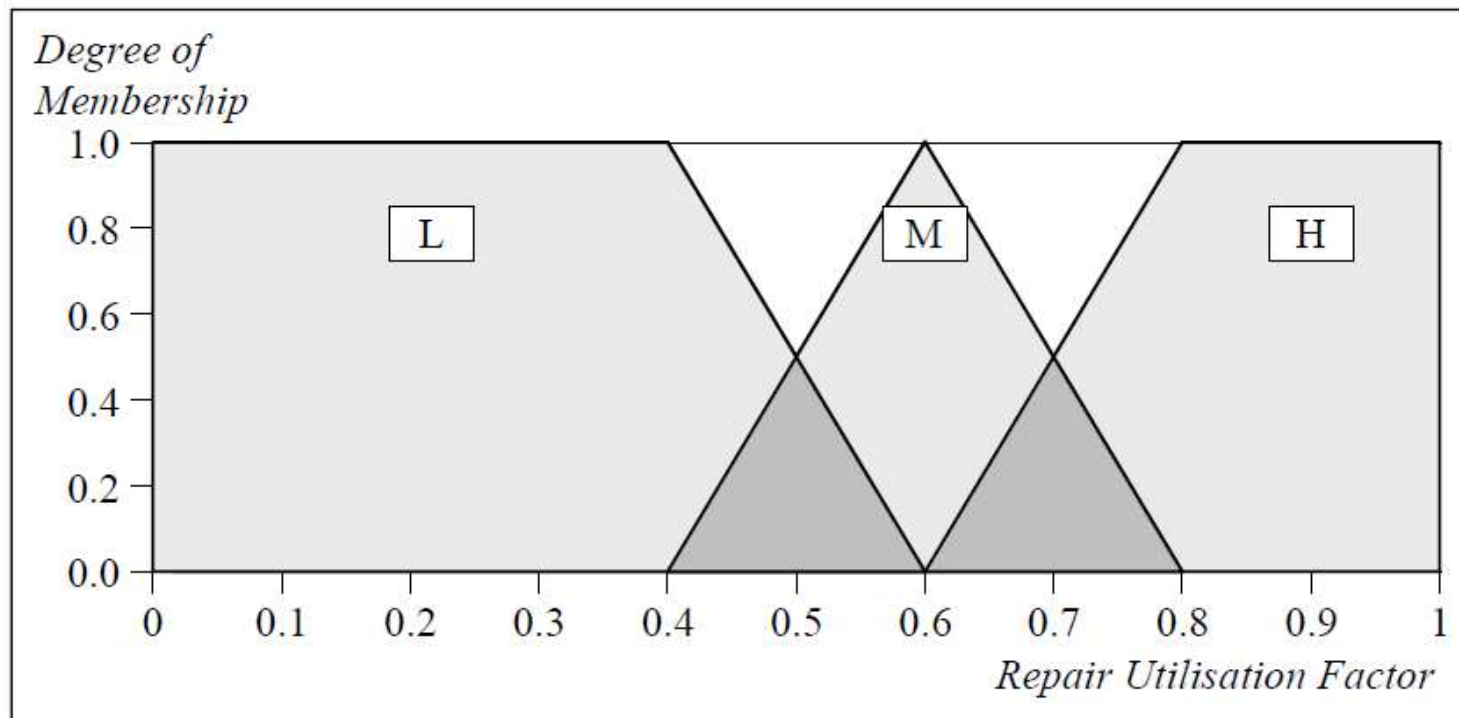
Building a Fuzzy System



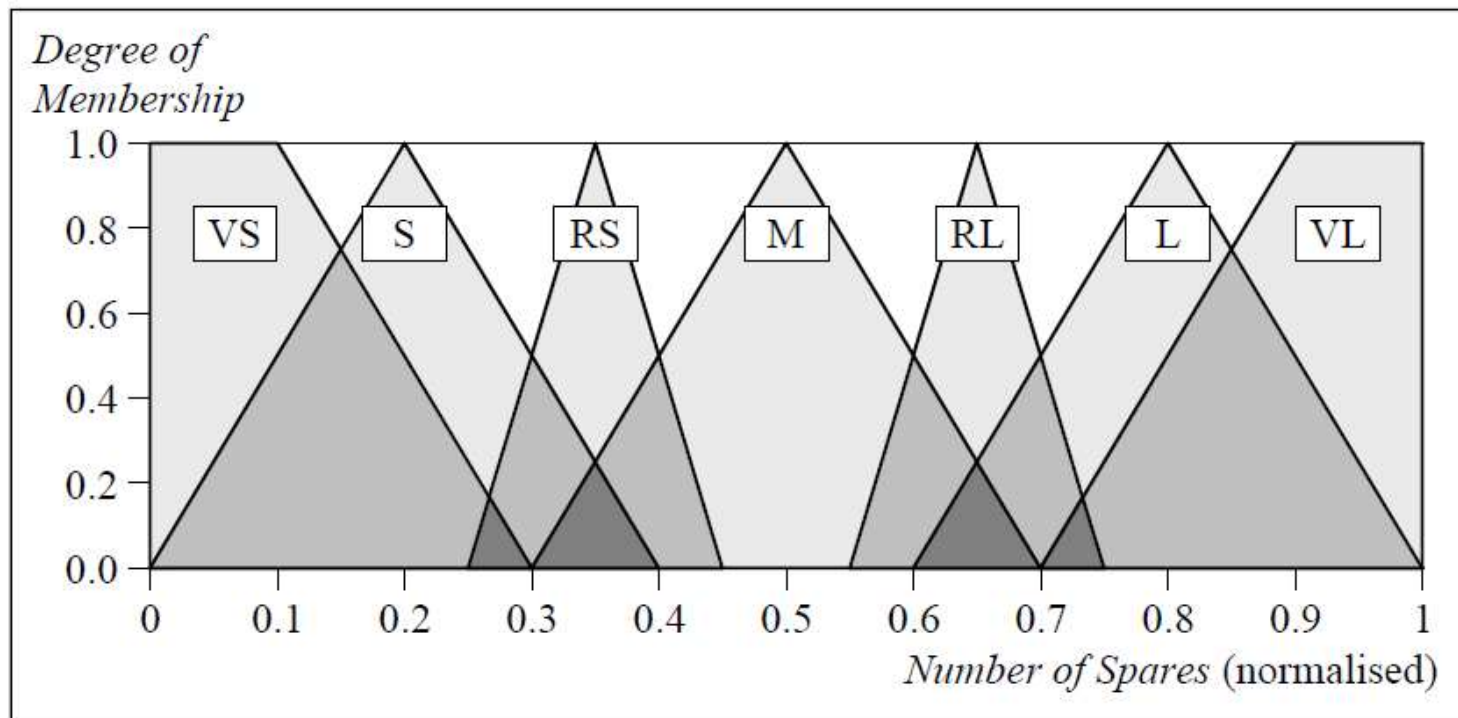
Building a Fuzzy System



Building a Fuzzy System



Building a Fuzzy System



Create Fuzzy Rules

To accomplish this task, we might ask the expert to describe how the problem can be solved using the fuzzy linguistic variables defined previously.

Required knowledge also can be collected from other sources such as books, computer databases, flow diagrams and observed human behaviour.

Create Fuzzy Rules

1. If (utilisation_factor is L) then (number_of_spares is S)
2. If (utilisation_factor is M) then (number_of_spares is M)
3. If (utilisation_factor is H) then (number_of_spares is L)
4. If (mean_delay is VS) and (number_of_servers is S) then (number_of_spares is VL)
5. If (mean_delay is S) and (number_of_servers is S) then (number_of_spares is L)
6. If (mean_delay is M) and (number_of_servers is S) then (number_of_spares is M)
7. If (mean_delay is VS) and (number_of_servers is M) then (number_of_spares is RL)
8. If (mean_delay is S) and (number_of_servers is M) then (number_of_spares is RS)
9. If (mean_delay is M) and (number_of_servers is M) then (number_of_spares is S)
10. If (mean_delay is VS) and (number_of_servers is L) then (number_of_spares is M)
11. If (mean_delay is S) and (number_of_servers is L) then (number_of_spares is S)
12. If (mean_delay is M) and (number_of_servers is L) then (number_of_spares is VS)

Create Fuzzy Rules

Rule	m	s	ρ	n	Rule	m	s	ρ	n	Rule	m	s	ρ	n
1	VS	S	L	VS	10	VS	S	M	S	19	VS	S	H	VL
2	S	S	L	VS	11	S	S	M	VS	20	S	S	H	L
3	M	S	L	VS	12	M	S	M	VS	21	M	S	H	M
4	VS	M	L	VS	13	VS	M	M	RS	22	VS	M	H	M
5	S	M	L	VS	14	S	M	M	S	23	S	M	H	M
6	M	M	L	VS	15	M	M	M	VS	24	M	M	H	S
7	VS	L	L	S	16	VS	L	M	M	25	VS	L	H	RL
8	S	L	L	S	17	S	L	M	RS	26	S	L	H	M
9	M	L	L	VS	18	M	L	M	S	27	M	L	H	RS

if mean_delay is VS
 and number_servers is S
 and utilization is Low

Evaluation and Tuning

- The last and the most laborious task is to evaluate and tune the system. We want to see whether our fuzzy system meets the requirements specified at the beginning.
- Several test situations depend on the mean delay, number of servers and repair utilisation factor.
- The MatLab's Fuzzy Logic Toolbox can generate surface to help us analyse the system's performance.
- However, the expert might not be satisfied with the system performance.
- To improve the system performance, we may use additional sets – *Rather Small* and *Rather Large* – on the universe of discourse *Number of Servers*, and then extend the rule base.

Evaluation and Tuning

1. Review model input and output variables, and if required redefine their ranges.
2. Review the fuzzy sets, and if required define additional sets on the universe of discourse.
3. Provide sufficient overlap between neighbouring sets. It is suggested that triangle-to-triangle and trapezoid-to-triangle fuzzy sets should overlap between 25% to 50% of their bases.

Evaluation and Tuning

4. Review the existing rules, and if required add new rules to the rule base.
5. Examine the rule-base for opportunities to write hedge rules to capture the pathological behaviour of the system.
6. Adjust the rule execution weights. Most fuzzy logic tools allow control of the importance of rules by changing a weight multiplier
7. Revise shapes of the fuzzy sets. In most cases, fuzzy systems are highly tolerant of a shape approximation.

Evaluation and Tuning

- certain common issues concerning all these three fuzzy inference systems
 - how to partition an input space
 - how to construct a fuzzy inference system for a particular application