

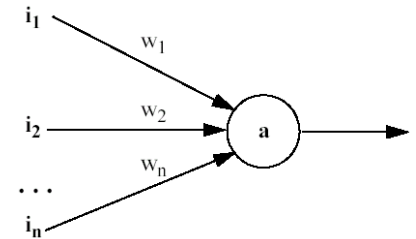
⋮

# 6. Intelligent Systems

- Neural Networks
- Genetic Algorithms
- Genetic Programming

⋮

# 6.1 Neural Networks

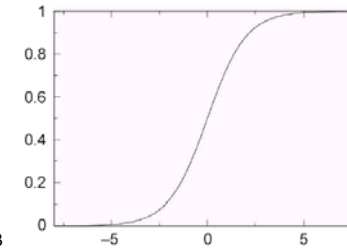


**Activation**

$$a(I, W) = \sum_{k=1}^n i_k \cdot w_k$$

**Output**

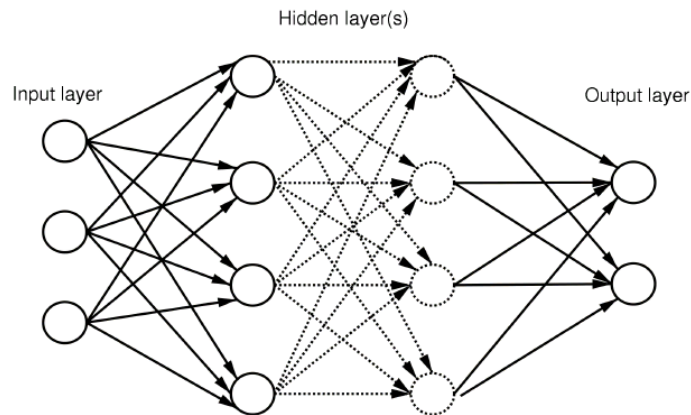
$$o(I, W) = \frac{1}{1 + e^{-p \cdot a(I, W)}}$$



Sigmoid activation function

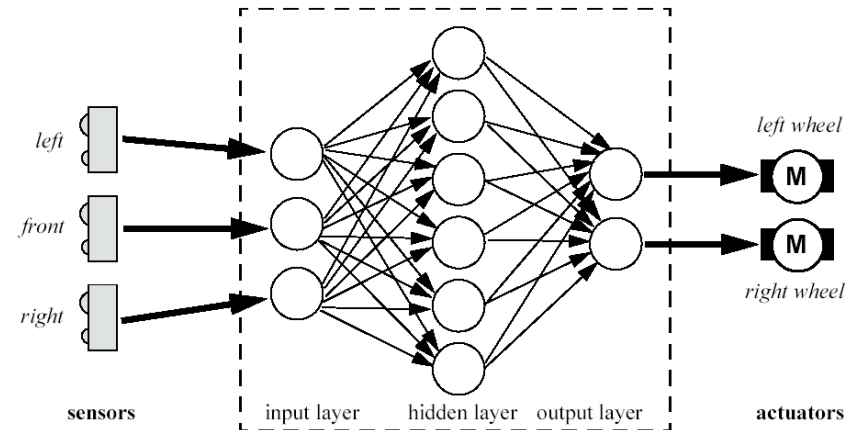
⋮

# Neural Networks



⋮

# Neural Networks



⋮

# Feedforward

```

void feedforward(float N_in[NIN],float N_hid[NHID],float N_out[NOUT])
{ int i,j;
  N_in[NIN-1] = 1.0; // set bias input neuron
  for (i=0; i<NHID-1; i++) // calculate activation of hidden neurons
  { N_hid[i] = 0.0;
    for (j=0; j<NIN; j++)
      N_hid[i] += N_in[j] * w_in[j][i];
    N_hid[i] = sigmoid(N_hid[i]);
  }
  N_hid[NHID-1] = 1.0; // set bias hidden neuron
  for (i=0; i<NOUT; i++) // calculate output neurons
  { N_out[i] = 0.0;
    for (j=0; j<NHID; j++)
      N_out[i] += N_hid[j] * w_out[j][i];
    N_out[i] = sigmoid(N_out[i]);
  }
}

```

⋮

⋮

# Backpropagation Learning

1. Initialize network with random weights
2. For all training cases:
  - a. Present training inputs to network and calculate output
  - b. For all layers (starting with output layer, back to input layer):
    - i. Compare network output with correct output (error function)
    - ii. Adapt weights in current layer

⋮

⋮

# Backpropagation Learning

$$E_{out\ i} = d_{out\ i} - out_i$$

$$E_{total} = \sum_{i=0}^{num(n_{out})} E_{out\ i}^2$$

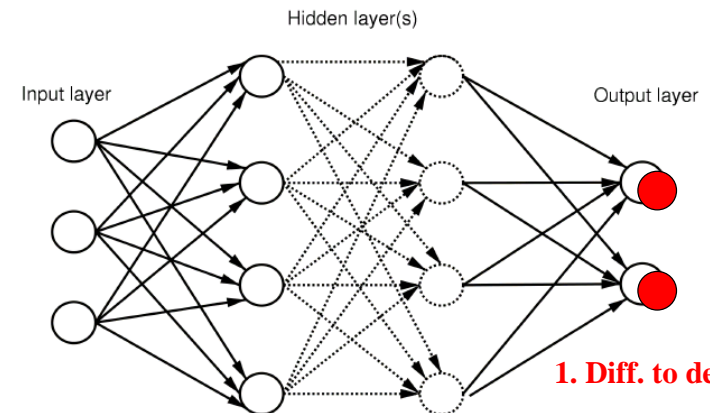
$$E_{hid\ i} = \sum_{k=1}^{num(n_{out})} E_{out\ k} \cdot w_{out\ i,k}$$

$$diff_{hid\ i} = E_{hid\ i} \cdot (1 - o(n_{hid\ i})) \cdot o(n_{hid\ i})$$

⋮

⋮

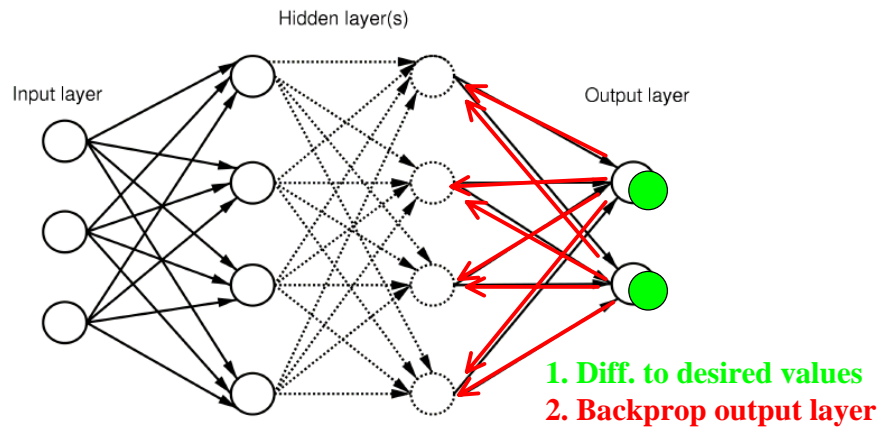
# Backpropagation Learning



⋮

⋮

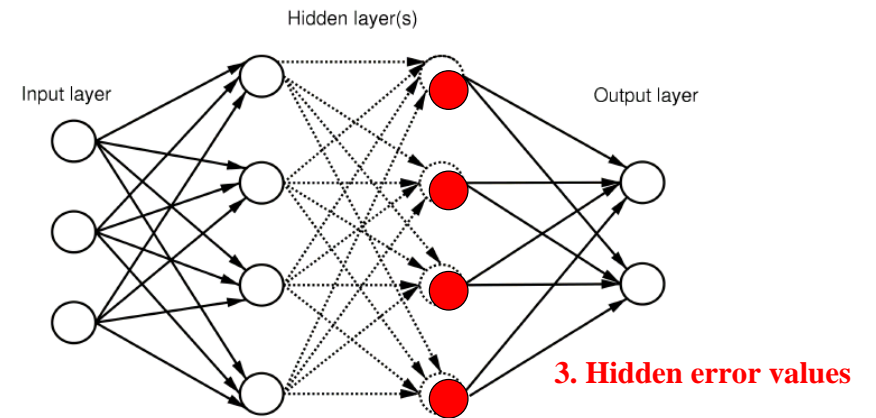
# Backpropagation Learning



⋮

⋮

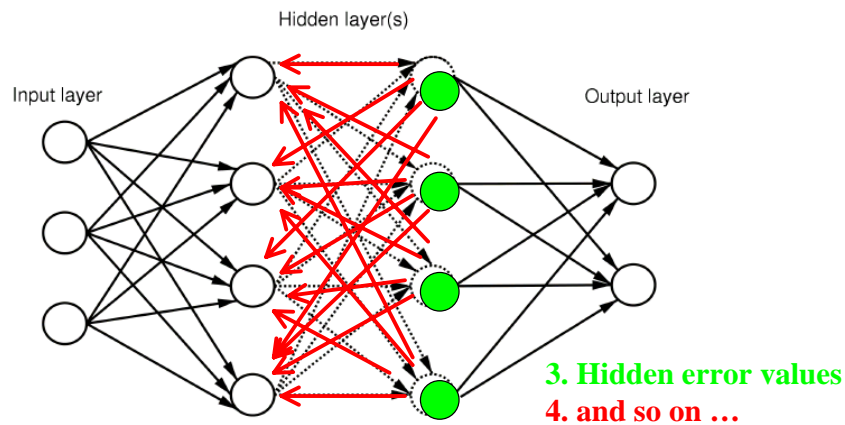
# Backpropagation Learning



⋮

⋮

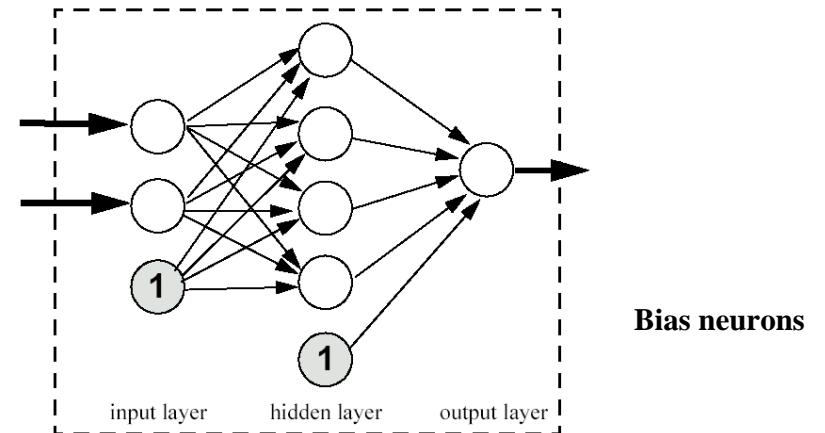
# Backpropagation Learning



⋮

⋮

# Backpropagation Learning



⋮

# Backpropagation Learning

```
float backprop(float train_in[NIN], float train_out[NOUT])
/* returns current square error value */
{ int i,j;
  float err_total;
  float N_out[NOUT],err_out[NOUT];
  float diff_out[NOUT];
  float N_hid[NHID], err_hid[NHID], diff_hid[NHID];

  //run network, calculate difference to desired output
  feedforward(train_in, N_hid, N_out);
  err_total = 0.0;
  for (i=0; i<NOUT; i++)
  { err_out[i] = train_out[i]-N_out[i];
    diff_out[i]= err_out[i] * (1.0-N_out[i]) * N_out[i];
    err_total += err_out[i]*err_out[i];
  }
}
```

# Backpropagation Learning

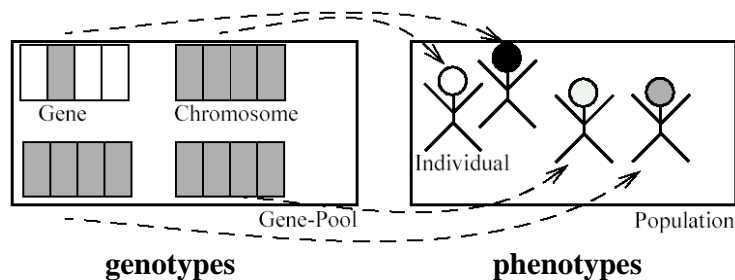
```
...
// update w_out and calculate hidden difference values
for (i=0; i<NHID; i++)
{ err_hid[i] = 0.0;
  for (j=0; j<NOUT; j++)
  { err_hid[i] += err_out[j] * w_out[i][j];
    w_out[i][j] += diff_out[j] * N_hid[i];
  }
  diff_hid[i] = err_hid[i] * (1.0-N_hid[i]) * N_hid[i];
}
// update w_in
for (i=0; i<NIN; i++)
  for (j=0; j<NHID; j++)
    w_in[i][j] += diff_hid[j] * train_in[i];

return err_total;
}
```

# 6.2 Genetic Algorithms

Charles Darwin 1859:

*On the Origin of Species by Means of Natural Selection, or Preservation of Favoured Races in the Struggle for Life*



# Genetic Algorithms

1. Randomly initialize a population of chromosomes.
2. While the terminating criteria have not been satisfied:
  - a. Evaluate the fitness of each chromosome:
    - i. Construct the *phenotype* (e.g. simulated robot) corresponding to the encoded *genotype* (chromosome).
    - ii. Evaluate the phenotype (e.g. measure the simulated robot's walking abilities), in order to determine its fitness.
  - b. Remove chromosomes with low fitness.
  - c. Generate new chromosomes, using certain selection schemes and genetic operators.

⋮

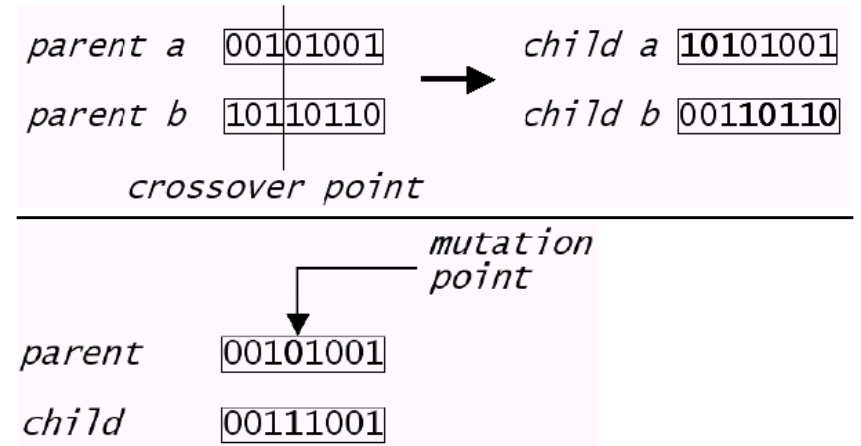
## Genetic Algorithms

- Fitness function
- Selection scheme
- Genetic operators
  - Crossover
  - Mutation

• • • • • • • •

⋮

## Genetic Operators



• • • • • • • •

⋮

## 6.3 Genetic Programming

- John Koza, 1992
- Evolve **program** instead of bitstring
- **Lisp** program structure is best suited
  - Genetic operators can do simple replacements of sub-trees
  - All generated programs can be treated as legal (no syntax errors)

• • • • • • • •

⋮

## Genetic Programming

1. Randomly generate a combinatorial set of computer programs.
2. Perform the following steps iteratively until a termination criterion is satisfied
  - a. Execute each program and assign a fitness value to each individual.
  - b. Create a new population with the following steps:
    - i. Reproduction: Copy the selected program unchanged to the new population.
    - ii. Crossover: Create a new program by recombining two selected programs at a random crossover point.
    - iii. Mutation: Create a new program by randomly changing a selected program.
3. The best sets of individuals are deemed the optimal solution upon termination

• • • • • • • •

⋮

# Lisp

- Every object is either an **atom** or a **list**
- Examples for atoms:  
7, 123, obj\_size
- Examples for lists:  
(1 2 3), (+ obj\_size 1), (+ (\* 8 5) 2)

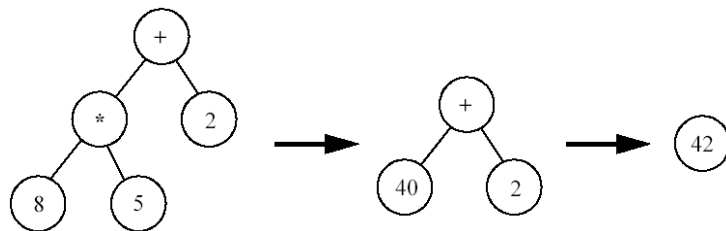
⋮

# Lisp

- Lisp does not distinguish between **data** and **programs**  
i.e. each object can be data or a program  
→ **S-expression**
- Examples for S-expressions:  
(+ 1 2) → 3  
(+ (\* 8 5) 2) → (+ 40 2) → 42

⋮

# Lisp



⋮

# Robot-Lisp

- Define subset of **Lisp** with robot functions
- Implement **generator** that produces Robot-Lisp programs from population with crossover and mutation
- Implement **interpreter** for Robot-Lisp programs, so these programs can be run (on simulator or real robot) and evaluated (determine fitness)

# Robot-Lisp

Name	Kind	Semantics
zero	atom, int, constant	0
low	atom, int, constant	20
high	atom, int, constant	40
(INC v)	list, int, function	<b>Increment</b> v+1
obj_size	atom, int, image sensor	search image for color object, return height in pixels (0..60)
obj_pos	atom, int, image sensor	search image for color object, return x-position in pixels (0..80) or return -1 if not found
psd_left	atom, int, distance sensor	measure distance in mm to left (0..999)
psd_right	atom, int, distance sensor	measure distance in mm to right (0..999)
psd_front	atom, int, distance sensor	measure distance in mm to front (0..999)
psd_back	atom, int, distance sensor	measure distance in mm to back (0..999)

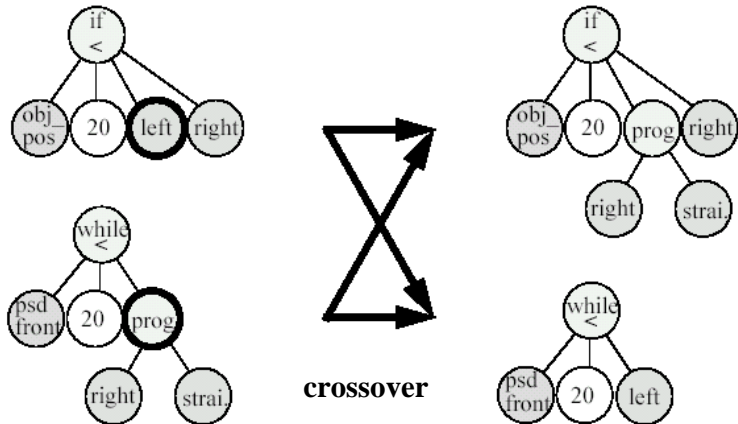
value

# Robot-Lisp

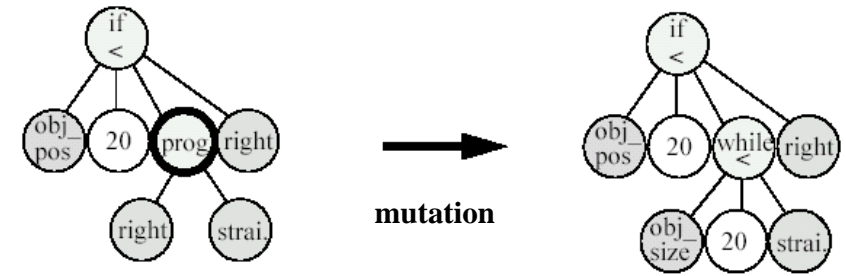
turn_left	atom, statem., act.	rotate robot 10° to the left
turn_right	atom, statem., act.	rotate robot 10° to the right
drive_straight	atom, statem., act.	drive robot 10cm forward
drive_back	atom, statem., act.	drive robot 10cm backward
(IF_LESS v1 v2 s1 s2)	list, statement, program construct	<b>Selection</b> if (v <sub>1</sub> <v <sub>2</sub> ) s <sub>1</sub> ; else s <sub>2</sub> ;
(WHILE_LESS v1 v2 s)	list, statement, program construct	<b>Iteration</b> while (v <sub>1</sub> <v <sub>2</sub> ) s;
(PROGN2 s1 s2)	list, statement, program construct	<b>Sequence</b> s <sub>1</sub> ; s <sub>2</sub> ;

statement

# Robot-Lisp Generator



# Robot-Lisp Generator



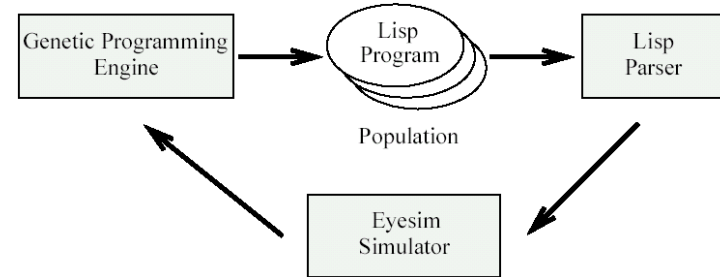
⋮

# Robot-Lisp Interpreter

```
int compute(Node n)
{ int ret, return_val1, return_val2;
  ...
  switch(n->symbol) {
  case PROGN2:
    compute(n->children[0]);
    compute(n->children[1]);
    break;
  case IF_LESS:
    return_val1 = compute(n->children[0]);
    return_val2 = compute(n->children[1]);
    if (return_val1 <= return_val2) compute(n->children[2]);
    else compute(n->children[3]);
    break;
  case turn_left:  turn_left(&vwhandle);
    break;
  case turn_right: turn_right(&vwhandle);
    break;
  }
```

⋮

# Evolution



⋮

# Evolution

Choose percentage of

- Keep best individuals
- Apply crossover
- Apply mutation

Selection Methods

- Fitness proportionate
- Tournament selection
- Linear rank selection
- Truncation selection

⋮

# Sample Problem: Ball Tracking

- Find ball in environment
- Drive towards it
- Stop when close to ball

# Ball Tracking

## Hand-coded solution in C/RoBIOS

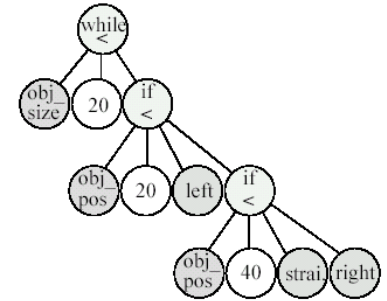
```
do
{ CAMGetColFrame(&c,0);
  ColSearch(c, BALLHUE, 10, &pos, &val); /* search image */

  if (val < 20) /* otherwise FINISHED */
  { if (pos == -1 || pos < 20) VWDriveTurn(vw, 0.10, 0.4);/* left */
    else if (pos > 60) VWDriveTurn(vw, -0.10, 0.4);/* right*/
    else VWDriveStraight(vw, 0.05, 0.1);
    VWDriveWait(vw); /* finish motion */
  }
} while (val < 20);
```

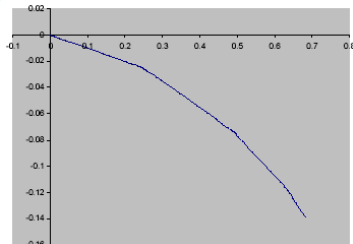
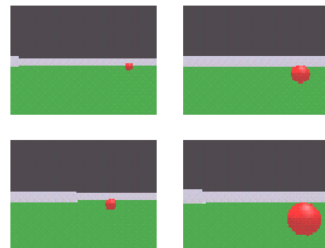
# Ball Tracking

## Hand-coded solution in Robot-Lisp

```
( WHILE_LESS obj_size low
  ( IF_LESS obj_pos low rotate_left
    ( IF_LESS obj_pos high drive_straight
      rotate_right )))
```



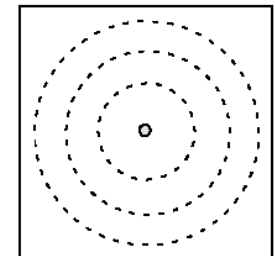
# Ball Tracking - Sample Solution



# Ball Tracking

- Run every individuum 4 times for evolution (add scores)
- Generate random starting positions and orientations, but at identical distances
- Fitness function:  
initial minus final ball distance

$$f = \sum_{i=1}^4 (dist_{i,0} - dist_{i,N})$$



⋮

# Evolution Results

## Generation 1, fitness 0.24

```
( IF_LESS obj-size obj-size turn-left move-forw )
```

## Generation 6, fitness 0.82

```
( WHILE_LESS low obj-pos move-forw )
```

## Generation 16, fitness 0.95

```
( WHILE_LESS low high ( IF_LESS high low turn-left ( PROGN2 ( IF_LESS low low move-back turn-left ) move-forw ) ) )
```

## Generation 22, fitness 1.46

```
( PROGN2 ( IF_LESS low low turn-left ( PROGN2 ( PROGN2 ( WHILE_LESS low obj-pos move-forw ) move-forw ) move-forw ) ) turn-left )
```

⋮

⋮

# Evolution Results

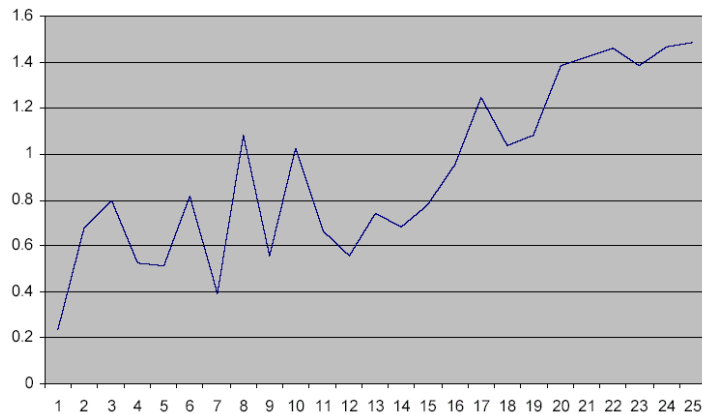
## Generation 25, fitness 1.49

```
( IF_LESS low obj-pos move-forw ( PROGN2 move-back ( WHILE_LESS low high ( PROGN2 turn-right ( PROGN2 ( IF_LESS low obj-pos move-forw ( PROGN2 turn-right ( IF_LESS obj-size obj-size ( PROGN2 turn-right move-back ) move-back ) ) ) move-forw ) ) ) ) )
```

⋮

⋮

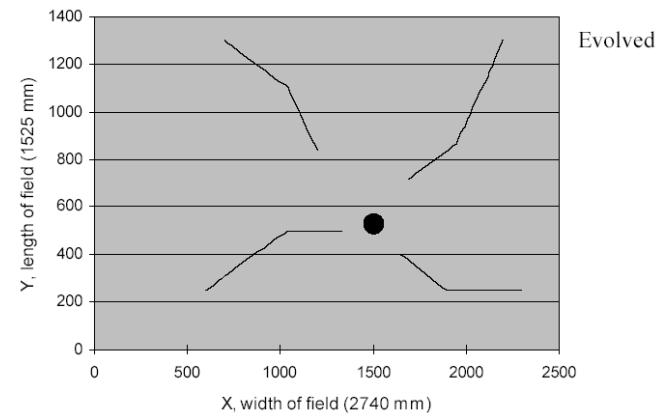
# Evolution Results



⋮

⋮

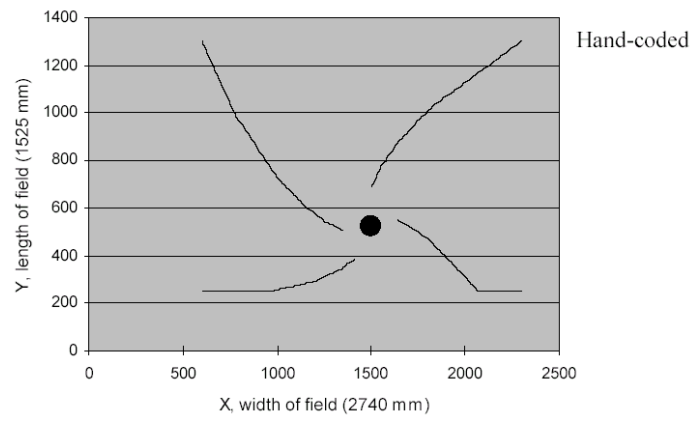
# Evolution Results



⋮

•  
•  
•

# Evolution Results



Bräunl 2003

41

• • • • • • • •