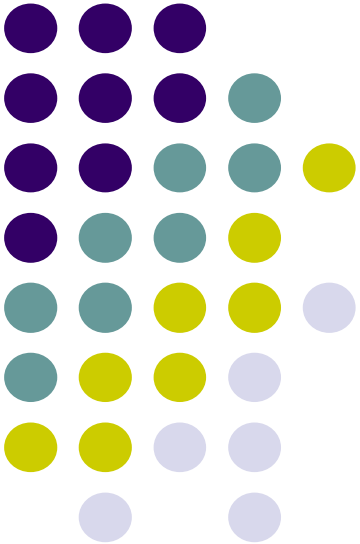
Lecture 10: Why GAs work, More on Gas ,The Microbial GA, Embodied Robotics

Computación Evolutiva

Gabriela Ochoa

http://www.ldc.usb.ve/~gabro/



Content

*We have covered the main elements of Evolution and GAs (***Heredity + Variation + Selection)**

Today, some GA background theory, review of Steady State GAs and Tournament Selection, The Microbial GA, Embodied Evolution and application in robotics

This lecture is based on a lecture from ALIFE course (by Inman Harvey) MSC Evolutionary and Adaptive Systems, The University of Sussex

Why Should GAs work ?

John Holland (1975) 'Adaptation in Natural and Artificial Systems' -- and most of the textbooks -- explain this with the **Schema Theorem**, and ideas of **building blocks.**

Roughly speaking, building blocks are segments of the genotype which encode for functional components of the 'phenotype', or potential solution to the problem.

These building blocks can, in principle, be evaluated independently of all the rest, as varying between 'good' and 'bad'.

Cartoon Version

Cartoon version of genotypes:

*** long legs **********|**********short arms**********

short legs*******|********** long arms **********

^

Recombination (when crossover happens to land appropriately) allows different parents like these **in one generation** to produce a child with long legs **and** long arms

Schemata

Schemata (plural of schema) are a formalisation of this idea of a building block.

Consider binary genotypes of length 16. Let # be a **'wild-card'** or **'dont-care'** character.

Then ######00#010######

is a schema of **order** 5 (5 specified alleles) and of **defining length** 6 (length of segment which includes specified alleles).

‘Processing Schemata’

Considering this schema
######00#010######
then
0000000001010000
is just **one** of many genotypes corresponding to this schema -- and actually this genotype also corresponds simultaneously to **many** other schemata.

Implicitly, the GA **'evaluates'** and **'processes'** loads of schemata **in parallel,** every generation.

The Schema Theorem claims …

... that **schemata** of **short defining lengths** (coding for building blocks such as 'cartoon legs') will,

✓**IF** they are of above-average fitness, (..that is, evaluated whatever the other loci outside the schema are)

✓get **exponentially** increasing numbers of trials in successive generations.

i.e., **despite** recombination and mutation being **'disruptive'** (too not too disruptive of short schemata)
'good building blocks' will multiply and **take over ---**
and '**mix and match**' with other 'good building blocks'.

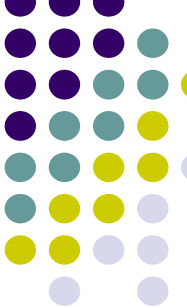
Implications of the Schema Theorem ??

The Schema Theorem is **formally** proved subject to certain conditions.

This Theorem is widely **interpreted** as implying that **RECOMBINATION** is the **'powerhouse'** of GAs,

-- whereas mutation is just a 'background operator‘ (whose only role is to add variety in loci where, throughout the whole population, no variety is left).

Doubts about the Schema Theorem



The Schema Theorem is formally **correct.**

But nowadays many peoplebelieve it has been **missinterpreted.**

The 'subject to certain conditions' bit means that this exponential increase is only guaranteed over 1 generation

-- thereafter the conditions **change!**

Recombination versus Mutation ?

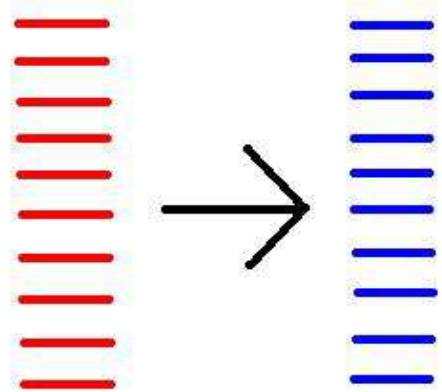
So be aware that despite this common view in the textbooks, some people think that in some sense **MUTATION** is the **powerhouse** of GAs, with recombination as a background (tho often useful) genetic operator.

"The Schema Theorem is true, but not very significant"

Nevertheless, the common view of the importance of recombination lies behind the exclusive emphasis (often without any mutation) on recombination in
GP = Genetic Programming.

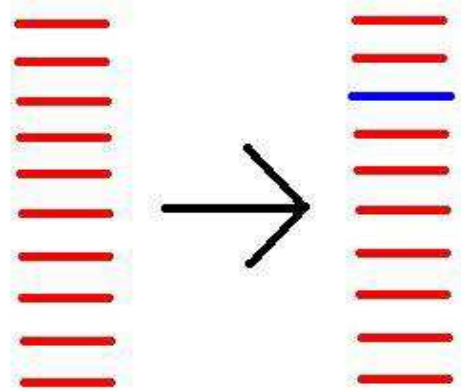
Generational vs. Steady State GA

You need not have a **generational** GA (where the whole population is swept aside every generation, and replaced by a fresh lot of offspring).



You can have a **STEADY STATE** GA.

Here just **ONE** member of the population is replaced at each time step, by the offspring of some others.



Steady State GA

Eg with a popn of 100:

- ✓Choose a mum by some selection mechanism
  biased towards the fitter.
- ✓Choose a dad by same method.
- ✓Generate a child by recombination + mutation
- ✓Add the child to the population
- ✓Keep the numbers down to 100 by choosing
  someone else to die
  (eg at random, or biased towards the less fit)

Roughly speaking, 100 times round this loop is equivalent to one generation of a generational GA

Tournament Selection

Here is a very simple way to implement the equivalent of linear rank selection in a Steady State GA

Pick 2 at random

3.5 compare fitnesses

2.7

Fittest of tournament is mum – choose dad the same way

Generate offspring from mum and dad – the new offspring replaces someone chosen at random.
Note: everyone else remains, including mum and dad.
Repeat until happy!

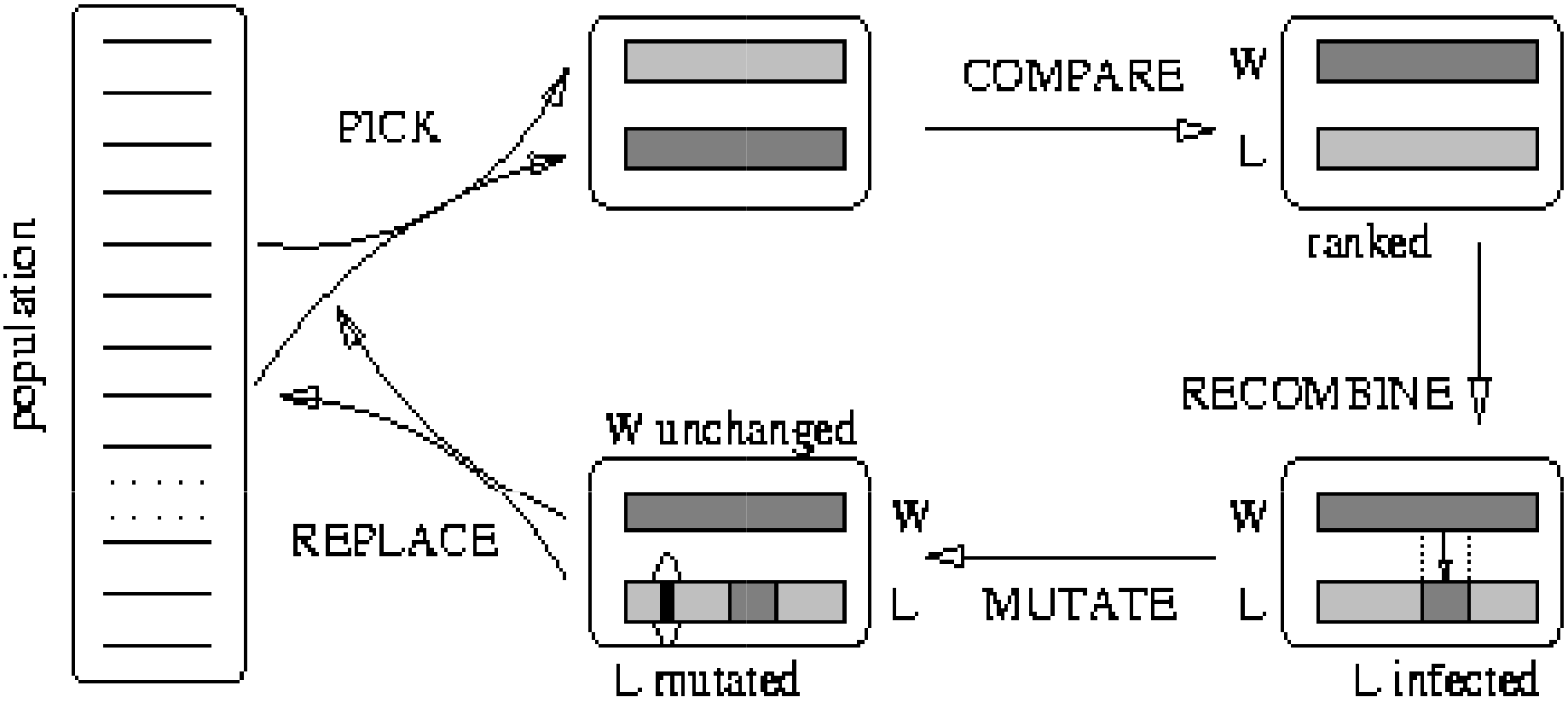
You needn’t even have death !

microbes can evolve by **horizontal** transmission
of genes (within the same generation)
rather than (or as well as) **vertical** transmission
(down the generations, from parents to offspring).

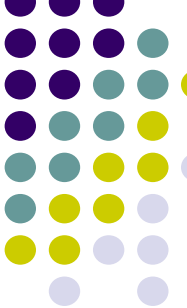
ie recombination happens **within** generations

Microbial sex =
'hey, wanna swap some of my genes for yours?'
rather than
'lets make babies'

Microbial Genetic Algorithm – the picture



Microbial Genetic Algorithm – the algorithm



- ✓Pick two genotypes at random
- ✓Compare scores -> Winner and Loser
- ✓Go along genotype, at each locus
  - with some prob copy from Winner to Loser (overwrite)
  - with some prob mutate that locus of the Loser

So **ONLY** the Loser gets changed
(gives a version of Elitism for free!)

This allows what is technically a one-liner GA (bar the evaluate(), which is problem-specific) -- quite a long line !

Microbial Genetic Algorithm – the one-liner

```
/* tournament loop */
for (t=0;t<END;t++)
        /*  loop along genotype of winner of tournament,
         selected in initial loop conditions */
        for (W=(evaluate(a=POP*drand48())>
                evaluate(b=POP*drand48()) ? a : b),
                        L=(W==a ? b : a), i=0; i<LEN; i++)
                /* throw dice to decide: cross or mutate */
                if ((r=drand48())<REC+MUT)
                /* update genotype of loser */
                gene[L][i]=(r<REC ? gene[W][i] : gene[L][i]^1);
```

… or slightly longer

```
int gene[POP][LEN];

Initialise genes at random;  define problem-specific evaluate(n)

/* tournament loop */
for (t=0;t<END;t++) {
        /*  pick 2 at random, find Winner and Loser */
        a=POP*drand48();
        do {b=POP*drand48()}
                while (a==b);  /*make sure a and b different */
        if (evaluate(a) > evaluate(b)) {W=a; L=b;}
        else {W=b; L=a;}
```

To be continued …

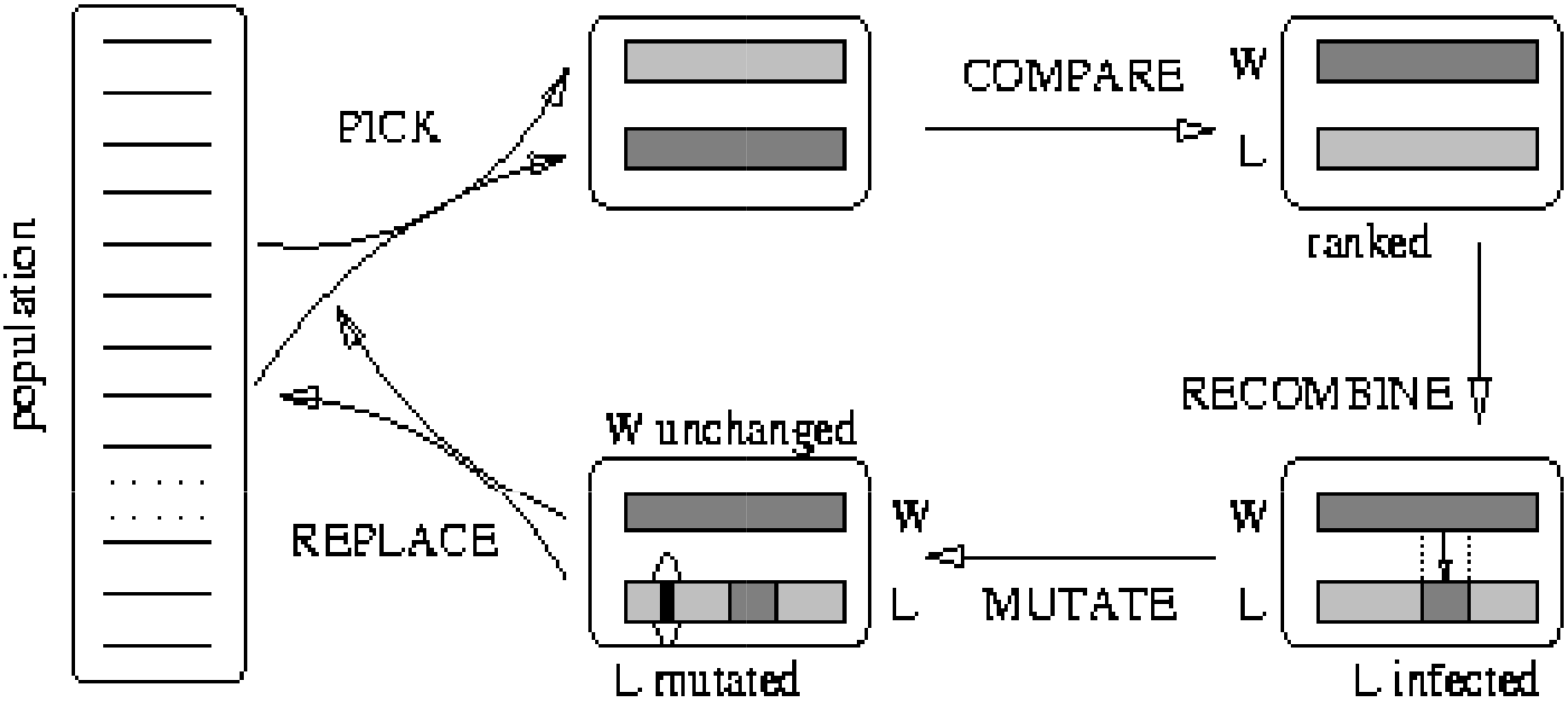
… continued

Continued …

```
        for (i=0;i<LEN;i++) {
            if (drand48()<REC) /* cross with probability REC */
                gene[L][i]=gene[W][i];
            if (drand48()<MUT) /* mutate with probability MUT */
                gene[L][i]=1-gene[L][i]; /* flip bit */
}       /* end tournament loop */
```

Possible values for **REC=0.5; ?** And **MUT=1.0/LEN;** **(If Binary) ?**

Microbial Genetic Algorithm – the picture



Is there a point ?

Microbial GA paper on my home page
http://www.cogs.susx.ac.uk/users/inmanh

It does actually work.

By no means guaranteed to be better than other GAs -- but does show **how really simple a GA can be**, and still work !

Apart from the one line, it needs declaration of gene[**POP**][**LEN**], initialisation of a random popn, and *evaluate(n)* that returns fitness of n^{th} member.

Embodied Evolution EE

Richard Watson, at Brandeis (papers available on web) has modified this to use with real robots in
'**Embodied Evolution**'.

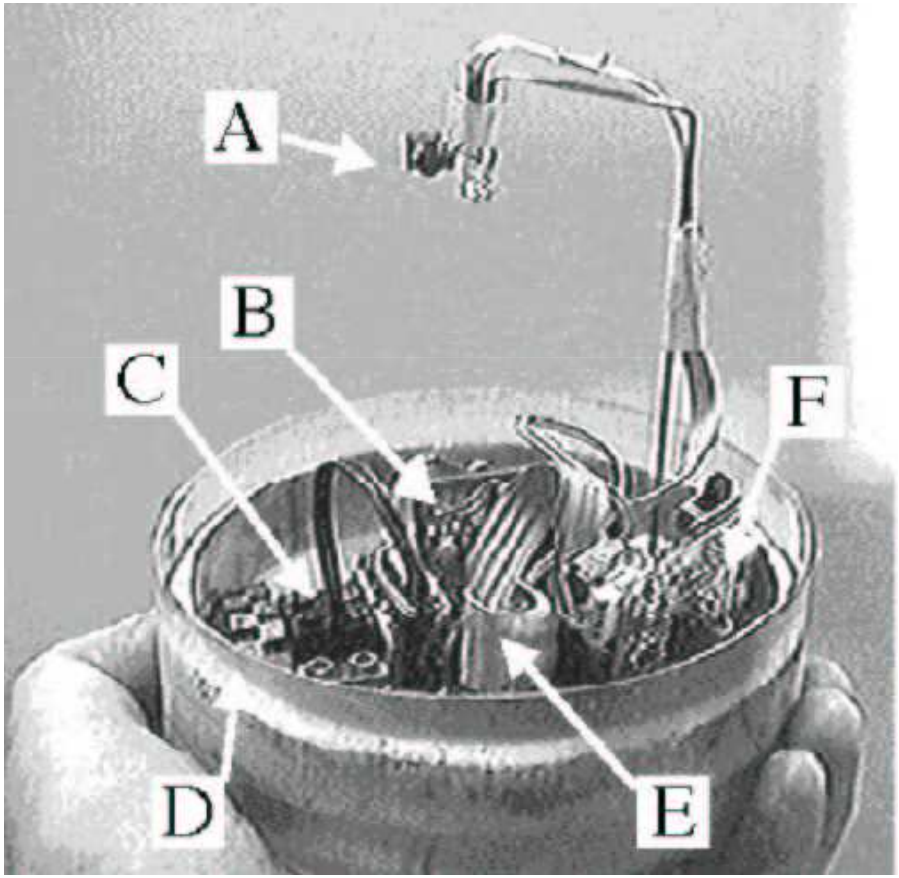
Robots go around 'broadcasting' their genes, and listening out to other broadcasts.

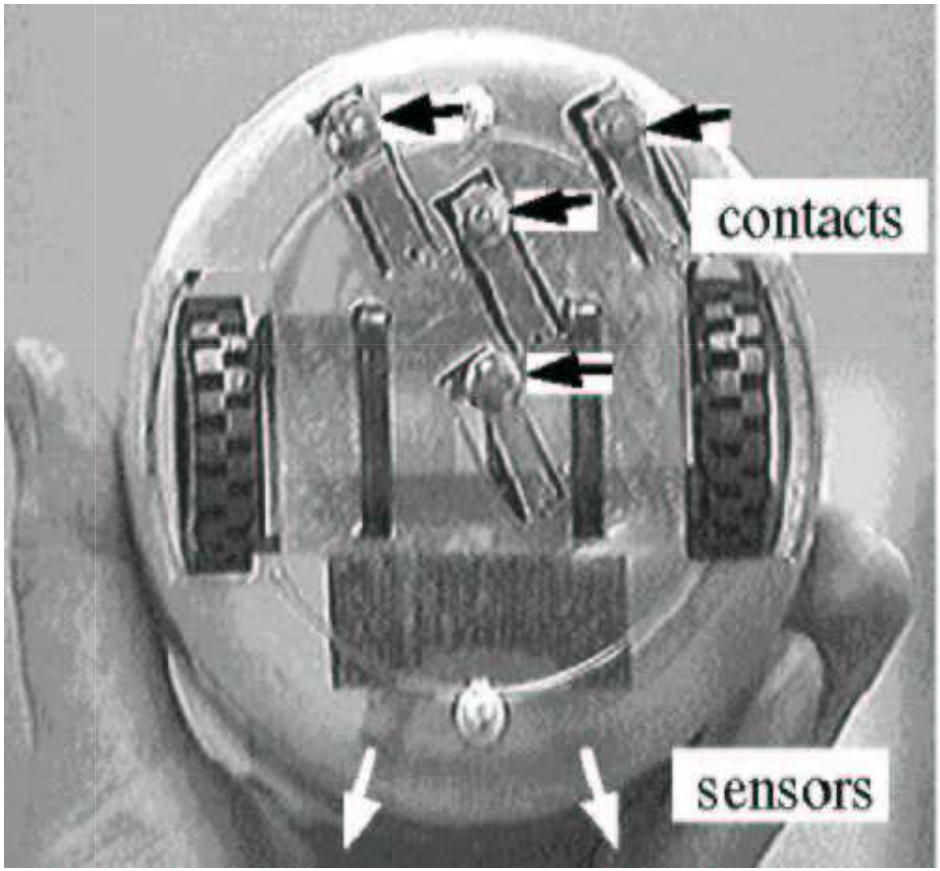
Fitter robots 'shout louder' (or more often)

Weaker robots are more likely to listen in, and use the genes they 'hear' to copy over their own.

Embodied Evolution (2)

- Evolutionary Robotics (ER), uses a virtual population (a set of controllers centrally stored)
- Fitness evaluation either simulation, or using real robots
- EE new methodology for evolutionary robotics (ER).
- EE uses a population of physical robots that autonomously reproduce with one another while situated in their task environment.
- Fully-distributed evolution algorithm embodied in physical robots.





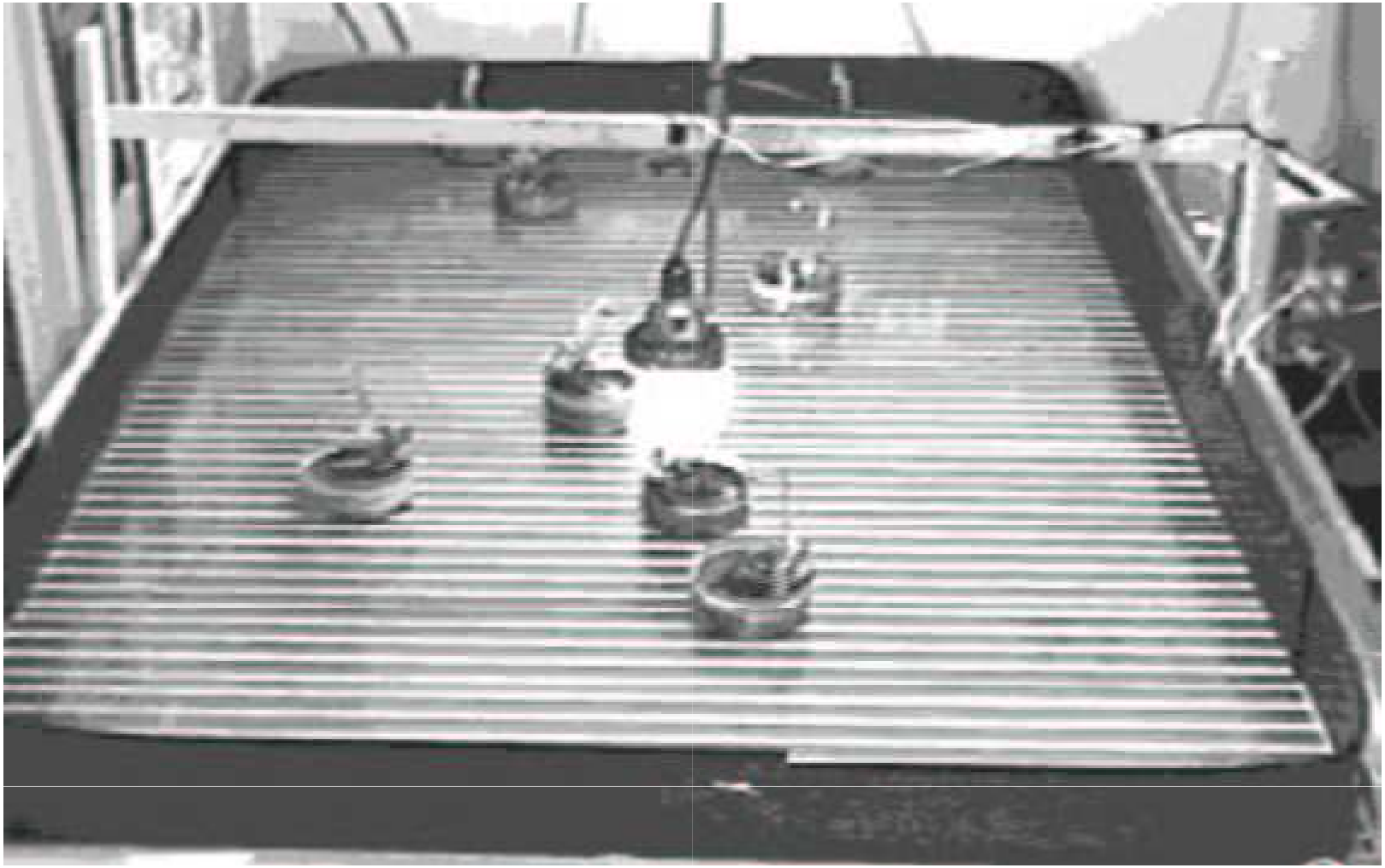
A) Infrared transmit/receive, B) PIC microcontroller, C) Lego motor

D) Tupperware body, E) Rechargeable cell, F) Recharge circuit

Light sensors, and 4 contact points that collect power from the floor

EE Experiment: Phototaxis task

- 8 robots
- Behaviour controlled by a simple ANN architecture
- Weights evolved to perform a phototaxis task
- Task environment: 1.30 x 2.0 M pen with a lamp in the middle, visible from all positions
- Robot task: reach the light from any starting point in the pen



Robot pen for the phototaxis experiment. 8 robots, light in the center, power floor

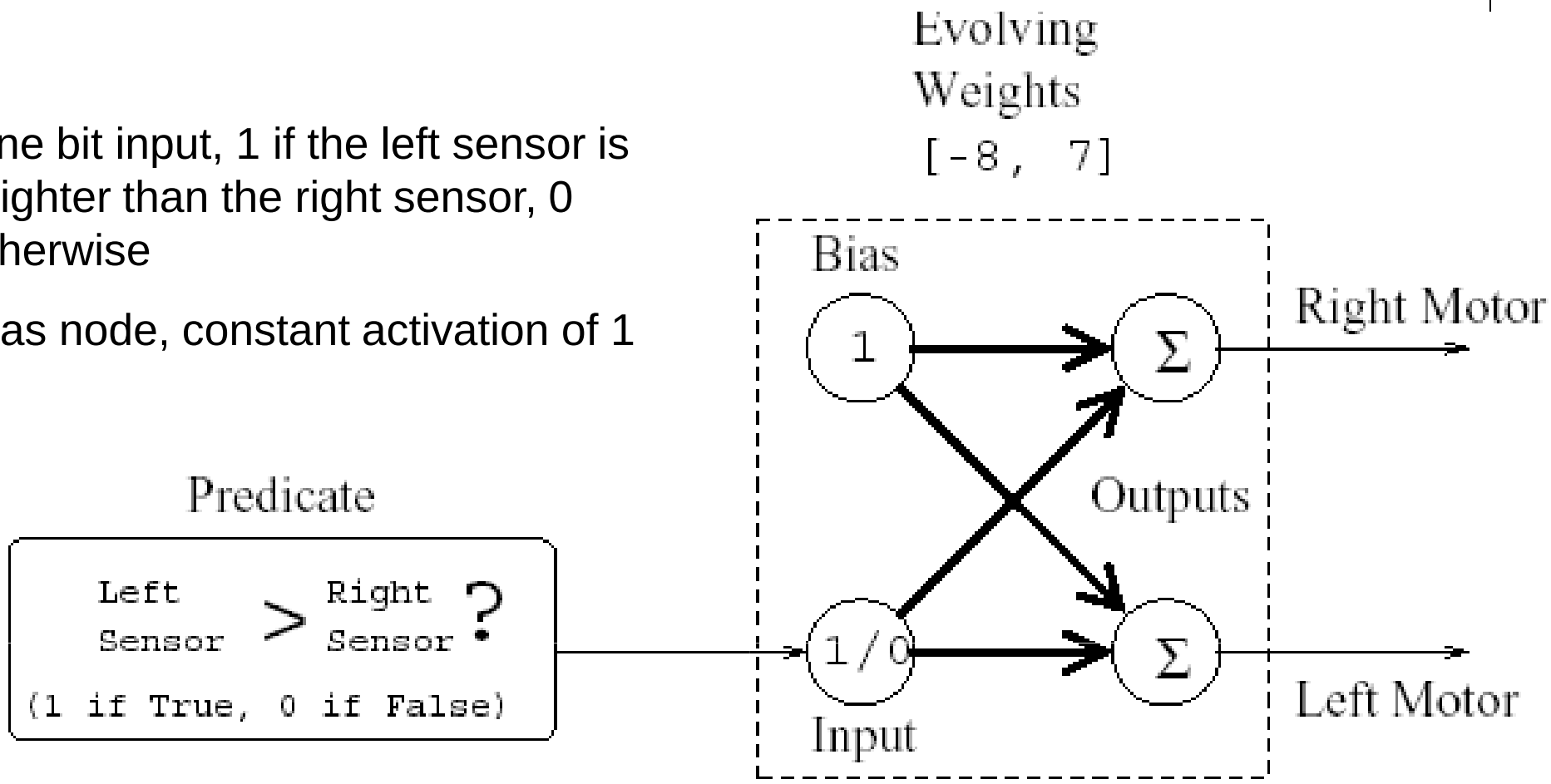
EE: Control Architecture

- Fully connected feed-forward neural network
- 4 nodes, 4 weights,
  - 2 outputs one for each motor (motor speed and direction)
  - 1 input node (binary-valued): which sensor is receiving more light, 1 bias node
- Values sent to output nodes: weighted sum of input nodes (no sigmoid function used)
- There is no individual learning, weights are evolved (robots get weights from other robots during reproduction

EE: Control architecture

One bit input, 1 if the left sensor is brighter than the right sensor, 0 otherwise

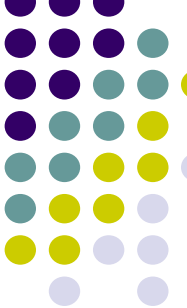
Bias node, constant activation of 1



EE: Evolution

- Robots reproduce when physically encounter
- Infrared communication channel broadcast of genes
- Energy levels reflect robot performance and regulate reproduction events. Energy level is updated as follows:
  - Whenever a robot reaches the light, its energy is increased
  - Whenever a robot sends a gene for reproduction, its energy is reduced by a small amount
- The robot rate of sending genes is proportional to its energy level

Choosing how to encode the Genotype

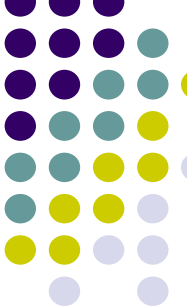


When faced with a new problem, if you are going to tackle it with a GA then one of the first decisions is:
How can I sensibly encode different phenotypes (possible solutions) as genotypes (artificial DNA, strings of symbols) ?

E.g., in the L-Systems examples, the symbols were those appropriate for an L-system rule (including [brackets])

Then it is necessary for genetic operators such as mutations to respect the encoding – cannot mutate just one side of a pair of brackets.

The purpose of the Genotype – Phenotype encoding



There are many possible ways **(P)** to solve your problem – many ways to build a plant, a neural network, a transmission tower …

You want to encode these different ways as different strings of symbols **(G)** so that **Heredity** and **Variation** work properly.

As far as possible, small changes in G (mutations) should make small changes in P. And inheriting bits of G from different parents should ideally result in inheriting bits of each parent's Phenotypic characteristics.

Fitness Function

When faced with a new problem, your first decision was:
How can I sensibly encode different phenotypes (possible solutions) as genotypes (artificial DNA, strings of symbols) ?

But then your second decision will have to be:

How can I sensibly give a score to each member of the population, how can I evaluate its *fitness* ?

This is a **problem-dependent** decision, no firm rules. Usually several different ways, some more sensible than others. This where you have to use **your** sense and discretion!