CSIT5900
Lecture 2: Reactive Agents

Department of Computer Science and Engineering
Hong Kong University of Science and Technology

Overview

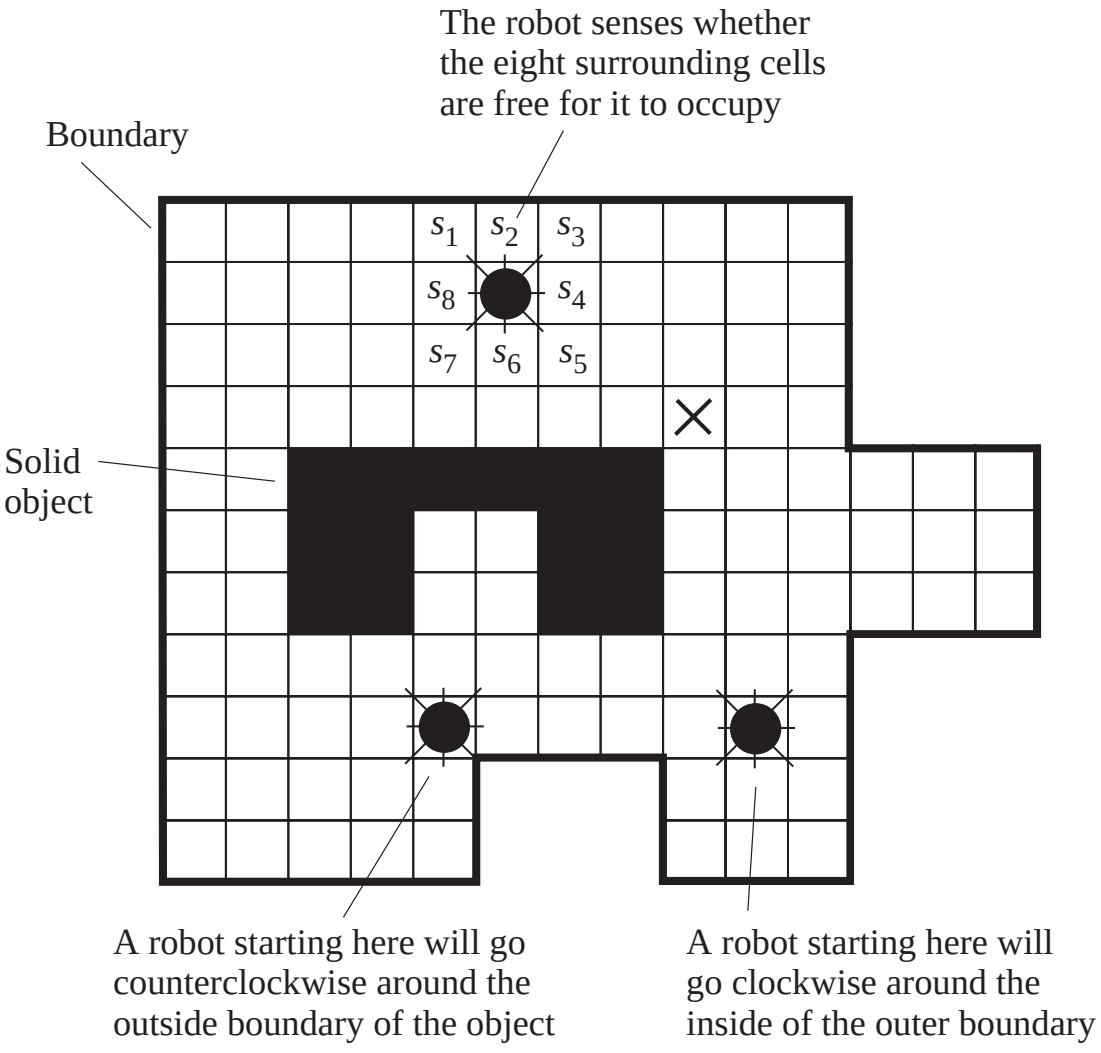
- Agent is a term referring to entities that can perform certain tasks in some environments autonomously.
- An agent needs to have perception, can perform some actions, has some purpose (goal).
- We first study stimulus-response agents, then consider adding states to these agents.
- We consider how to control these agents using rules, neural networks, and genetic algorithms, and whether the programs are desiged or learned (evolved).

Stimulus-Response Agents

Stimulus-Response (S-R) agents are machines without internal states that simply react to immediate stimuli in their environments.

A Boundary-Following Robot

A robot in a two-dimensional grid world:



Sensors and Actions

Sensors: eight of them s_1 - s_8. $s_i = 1$ if the corresponding cell is occupied, it equals to 0 otherwise.
Actions: the robot can move to an adjacent free cell. There are four of them:

1. *north* - moves the robot one cell up.
2. *east* - moves the robot one cell to the right.
3. *south* - moves the robot one cell down.
4. *west* - moves the robot one cell to the left.

All of them have their indicated effects unless the robot attempts to move to a cell that is not free; in that case, it have no effect.

Controling a robot

We now have a model of the problem: an environment modeled by a two-dimensional grid world, an agent with sensors to check if any of the nearby cells is occupied, a collection of actions for moving around the world, and the task of following the boundary of the first obstacle that it comes into.
We now need an algorithm to control the robot!

Learning Action Function - Supervised Learning with TLUs

Supervised Learning

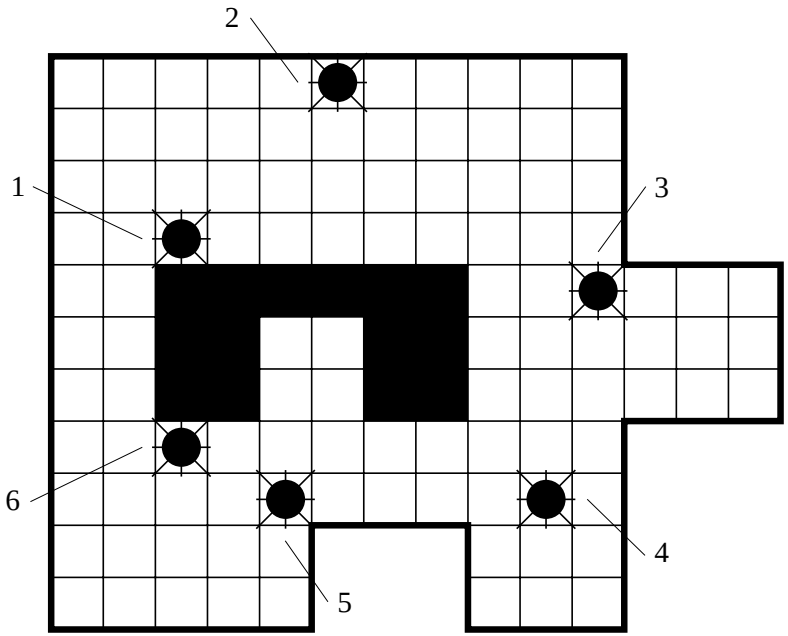
Given a training set consisting of

- a set Σ of n-dimensional vectors (these vectors could be the vectors of the robot's sensory inputs, or they might be the feature vectors computed by the perceptual processing component);
- for each vector in Σ, an associated action called *label* of the vector (this action could be the one that the learner observed performed by the teacher, or simply the desired action given by the designer of a robot);

the task of learning is to compute a function that responds "acceptably" to the members of the training set: this usually means that the function agrees with as many members of the training set as possible.
What functions to learn is crucial. Here we consider a class of simple linear weighted functions called TLUs.

Example - When to Move East

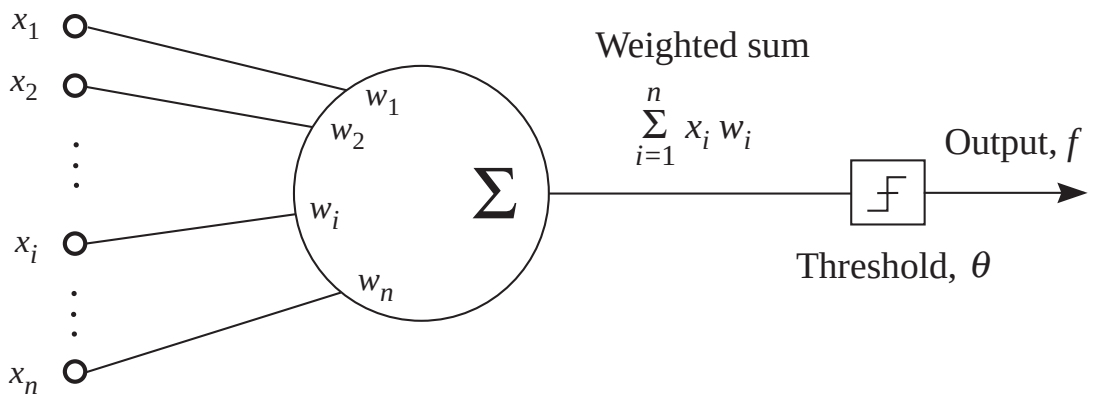
A training set for learning when to move east:



Input number	Sensory vector	$x_1\overline{x_2}$ (move east)
1	00001100	0
2	11100000	1
3	00100000	1
4	00000000	0
5	00001000	0
6	01100000	1

TLUs

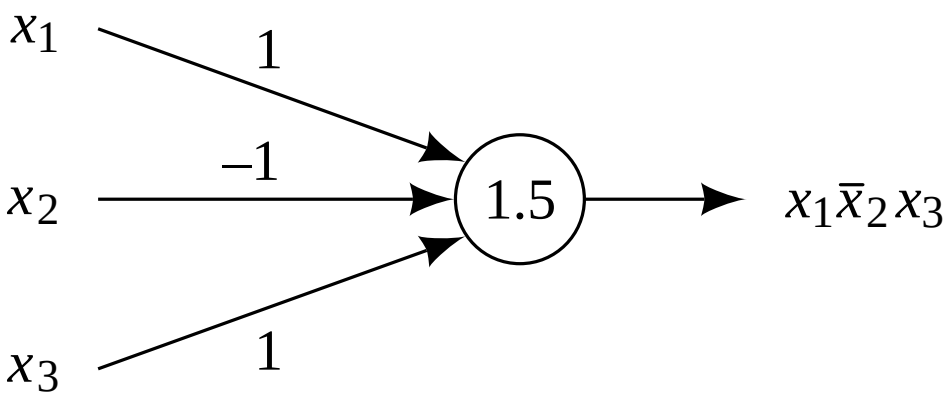
- Boolean functions, thus those production systems whose conditions are Boolean functions can be implemented easily as programs. They can also be implemented directly as circuits.
- A type of circuits of particular interest in AI is *threshold logic unit (TLU)*, also called *perceptron*:



$$f = 1 \text{ if } \sum_{i=1}^{n} x_i w_i \geq \theta$$
$$= 0 \text{ otherwise}$$

TLUs

Not all Boolean functions can be implemented as TLUs - those can are called *linearly separable functions*. Here is an example:



Neurons

TLU is a very simple model of neurons.
`http://en.wikipedia.org/wiki/Neuron`:

- A neuron is an electrically excitable cell that processes and transmits information through electrical and chemical signals via synapses.
- Neurons when connect to each other form neural networks, and are the core components of the nervous system.
- The number of neurons in the brain varies dramatically from species to species.
- One estimate (published in 1988) puts the human brain at about 100 billion (10^{11}) neurons and 100 trillion (10^{14}) synapses.
- The fruit fly Drosophila melanogaster, a common subject in biological experiments, has around 100,000 neurons and exhibits many complex behaviors.

Learning TLUs

Recall that a TLU is determined by:

- number of inputs;
- for each input an associated number called its weight;
- a number called the threshold.

We know the number of inputs from the training set; we can assume that the threshold is always o by introducing a new input with its weight set to be the negative of the original threshold, and its input always set to 1.
So what we need to learn is the vector of weights.

The Error-Correction Procedure

Given a vector $\mathbf{X}$ from the training set (augmented by the $n+1$th special input 1), let d be the label (desired output) of this input, and f the actual output of the old TLU, the weight change rule is:

$$\mathbf{W} \leftarrow \mathbf{W} + c(d - f)\mathbf{X},$$

where c, a number, is called the learning rate.
To use this rule to learn a TLU: start with a random initial weight vector, choose a learning rate c, and then iterate through the set of training set repeatedly until it becomes stable.
If the traning set can be captured by a linearly separable Boolean function, then this procedure will terminate. The exact number of steps needed depends on:

- Initial weight values
- Learning rate in weight updating rule
- Order of presentation of training examples

Activation Functions

- A TLU (perceptron) is a simple model of a neuron. According to it, a neuron is either on or off.
- There are many other models or activation functions, and almost all of them are defined using the weighted sum, also called *score*, of the inputs $\mathbf{w} \cdot \mathbf{x}$.
- Linear activation: $output(\mathbf{x}) = a + \mathbf{w} \cdot \mathbf{x}$.
- ReLU activation: $output(\mathbf{x}) = max(0, a + \mathbf{w} \cdot \mathbf{x})$.
- Logistic activation: $output(\mathbf{x}) = (1 + exp(-a - \mathbf{w} \cdot \mathbf{x}))^{-1}$.
- See `https://en.wikipedia.org/wiki/Activation_function` for more.

Artificial Neural Networks

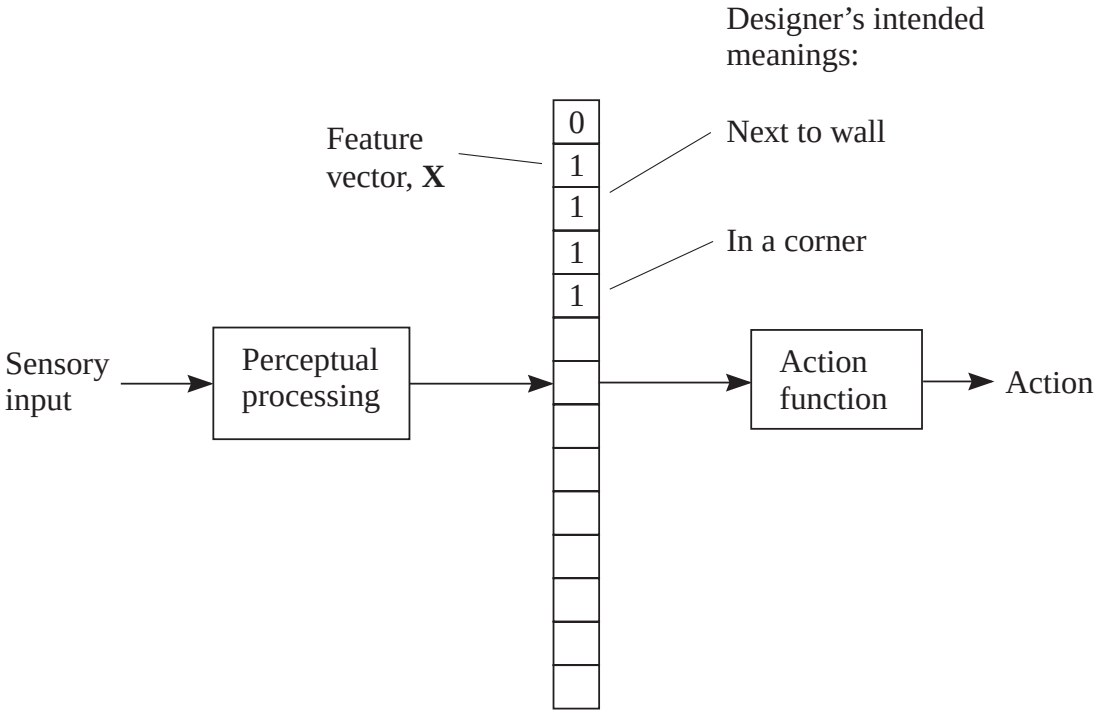
- An (artificial) neural network is a directed graph whose nodes are models of neurons.
- The sources (no incoming arcs) are inputs and the targets (no outgoing arcs) are outputs. The internal nodes represent "hidden" features.
- A general description of machine learning algorithms for training a neural network is as follows

  Initialize $\mathbf{w}$ (e.g. $\mathbf{0}$);
  while C {
  $\mathbf{w} = \mathit{successor}(\mathbf{w})$;
  update C; }

  where C is the condition for keep updating the weights, e.g. it can be $i \leq N$ for some fixed N or could be a condition about the desired accuracy; $\mathit{successor}(\mathbf{w})$ returns a "better" $\mathbf{w}$ and is computed using training instances and the loss function.
- In ML, two popular ways of updating weights are so called gradient descend and stochastic gradient descend.

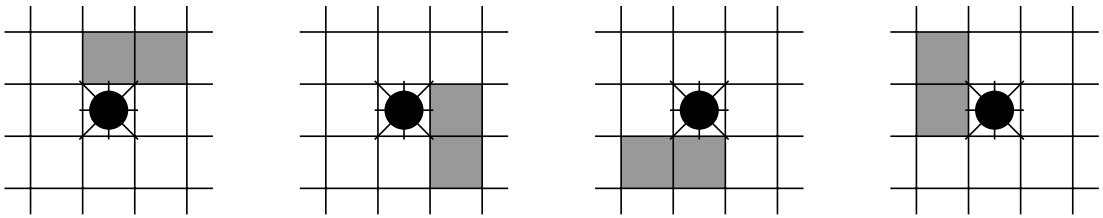
Basic Architecture

Designer's job: specify a function of the sensory inputs that selects actions appropriate for the task at the hand (boundry-following in our example). In general, it is often convenient to divide the function into two components: perceptual processing and action function:



Perception

- The robot has 8 sensory inputs $s_1,\ldots,s_8$.
- There are $2^8 = 256$ different combinations of these values.
- Although some of the combinations are not legal because of our restriction against tight spaces (spaces between objects and boundaries that are only one cell wide), there are still many legal ones.
- For the task at the hand, we only need four binary-valued features about the sensor: x_1, x_2, x_3, x_4:



In each diagram, the indicated feature has value 1 if and only if at least one of the shaded cells is *not* free.

Action

Given the four features, we can specify a function of these features that will select the right action for boundary-following as follows:

- if none of the four features equal to 1, then move *north* (it could be any direction).
- if $x_1 = 1$ and $x_2 = 0$, then move *east*;
- if $x_2 = 1$ and $x_3 = 0$, then move *south*;
- if $x_3 = 1$ and $x_4 = 0$, then move *west*;
- if $x_4 = 1$ and $x_1 = 0$, then move *north*.

We now need to represent and implement perception and action functions. To that end, we briefly review Boolean algebra below.

Boolean Algebra

- A Boolean function maps an n tuple of $(0, 1)$ values to $\{0, 1\}$;
- Boolean algebra is a convenient notation for representing Boolean functions using $\cdot$ (and, often omitted), $+$ (or), and $-$ (negation):

$$1 + 1 = 1,\ 1 + 0 = 1,\ 0 + 0 = 0,$$
$$1 \cdot 1 = 1,\ 1 \cdot 0 = 0,\ 0 \cdot 0 = 0,$$
$$\overline{1} = 0,\ \overline{0} = 1$$

  The two binary operators are commutative and associative.
- Examples: x_4 is $s_1 + s_8$; the condition for the robot moving *north* is $\overline{x_1 x_2 x_3 x_4} + x_4 \overline{x_1}$.

Production Systems

One convenient representation for an action function is *production systems*.

- A production system is a *sequence* of *productions*, which are rules of the form:

$$c \rightarrow a$$

  meaning if condition c is true, then do a. Here a could be a primitive action, a call to a procedure, or a set of actions to be executed simultaneously.
- When there are more than one productions can be fired (their condition parts are true), then the first one is applied.
- The following is a production system representation of the action function for our boundary-following robot:

$$x_4\overline{x_1} \rightarrow \mathit{north}, \qquad x_3\overline{x_4} \rightarrow \mathit{west},$$
$$x_2\overline{x_3} \rightarrow \mathit{south}, \qquad x_1\overline{x_2} \rightarrow \mathit{east},$$
$$1 \rightarrow \mathit{north}.$$

- Here is a production system for getting the robot to a corner:

$$\begin{aligned} \textit{inCorner} &\rightarrow \textit{nil}, \\ 1 &\rightarrow \textit{bf}, \end{aligned}$$

where *inCorner* is a feature of the sensory input corresponding to detecting a corner, *nil* is the do-nothing action, and *bf* is the action that the above boundary-folliwng production system will produce.

Learning action functions with genetic programming.

Machine Evolution

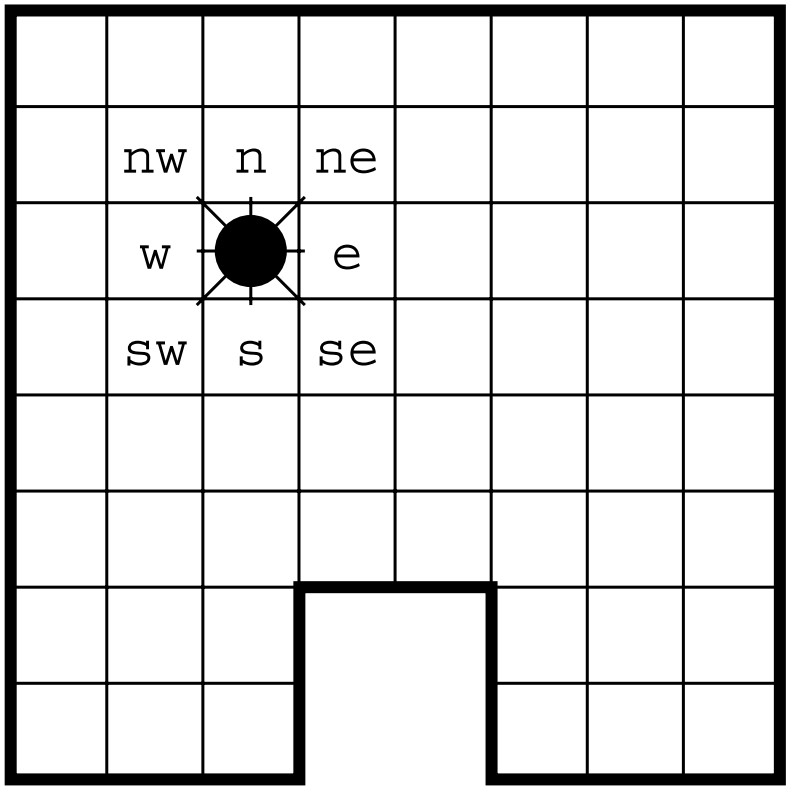
- Human becomes what we are now through millions of years evolution from apes.
- Presumably, we could simulate this evolution process to evolve smart machines as well.
- Evolution has two key components:
  - reproduction (how do parents produce descendants); and
  - survival of the fittest (how to select which of these descendants are going to reproduce more descendants).

Genetic Programming (GP)

- GP is about evolving programs to solve specific problems.
- The first issue is what the representation is for programs: the techniques for evolving C programs would be different from that for ML programs. The general idea is:
  - decide what the legal programs are;
  - define a fitness function;
  - select a set of legal programs as generation 0;
  - produce the next generation until a desired program is produced.
- Three common techniques for producing the next generations are:
  - copy (clone a good gene to the next generation);
  - crossover (mix-up two parent's genes);
  - mutation (mutate a gene)
- I will illustrate the idea using the boundary-following robot.

The Task

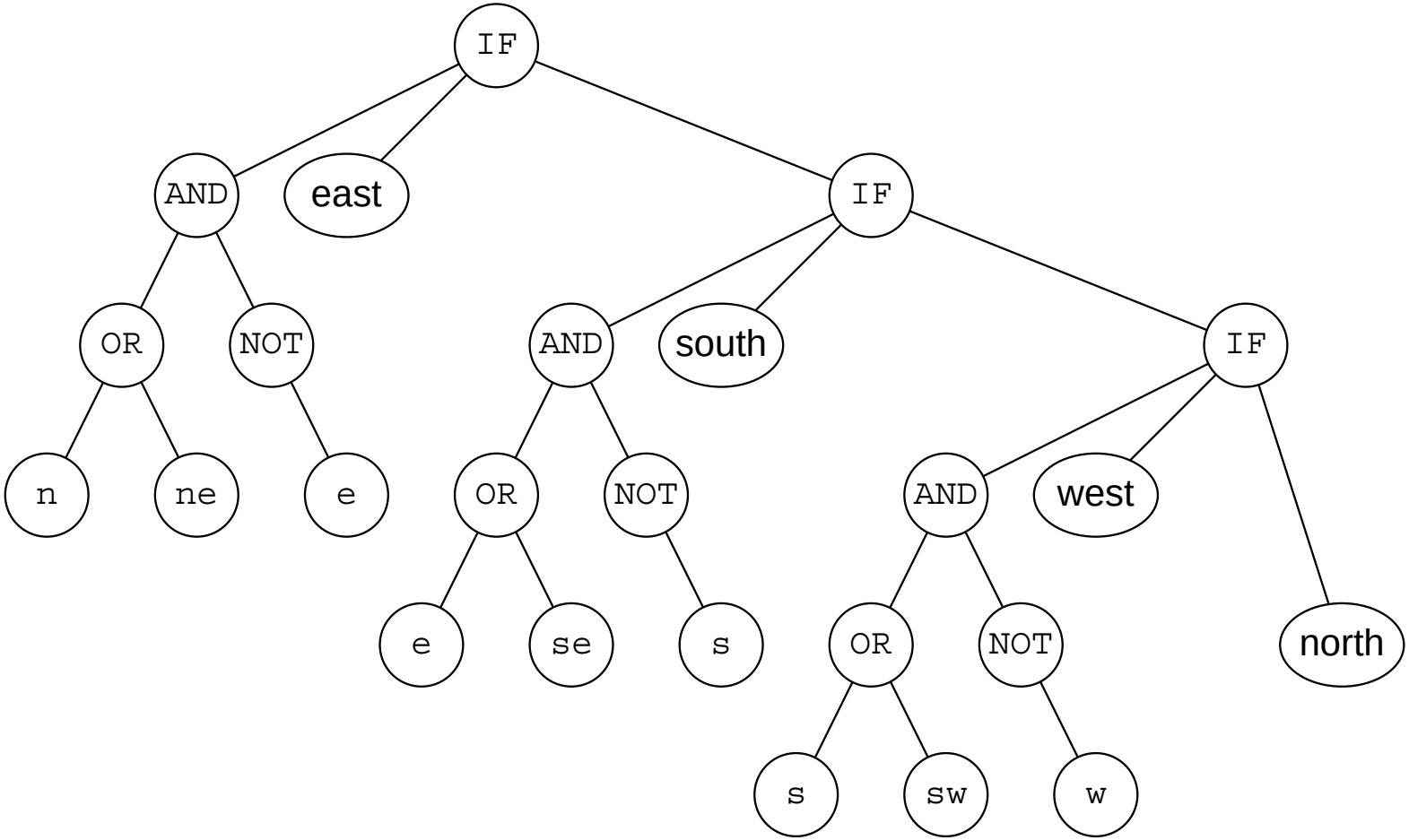
Wall-following in the following grid-world (we have to fix the environment):



Program Representation

We shall assume that the program that we are looking for can be constructed from the primitives ones (*east*, *west*, etc) by Boolean ops and conditionals. The example in the following slide represent a program that does the same thing as the following production system:

$$
\begin{aligned}
(n + ne)\overline{e} &\rightarrow east, \\
(e + se)\overline{s} &\rightarrow south, \\
(s + sw)\overline{w} &\rightarrow west, \\
1 &\rightarrow north.
\end{aligned}
$$




```
(IF (AND (OR (n) (ne)) (NOT (e)))
    (east)
    (IF (AND (OR (e) (se)) (NOT (s)))
        (south)
        (IF (AND (OR (s) (sw)) (NOT (w)))
        (west)
        (north))))
```

Fitness Function

- Given a program, and a starting position for the robot, we run it until it has carried out 60 steps, and then count the number of cells next to the wall that are visited during these 60 steps. (Max=32, Min=0.)
- For a given program, we do ten of these runs with the robot starting at ten randomly chosen starting positions. The total count of the next-to-the-wall cels visited is taken as the *fitness* of the program. (Max = 320, Min=0.)

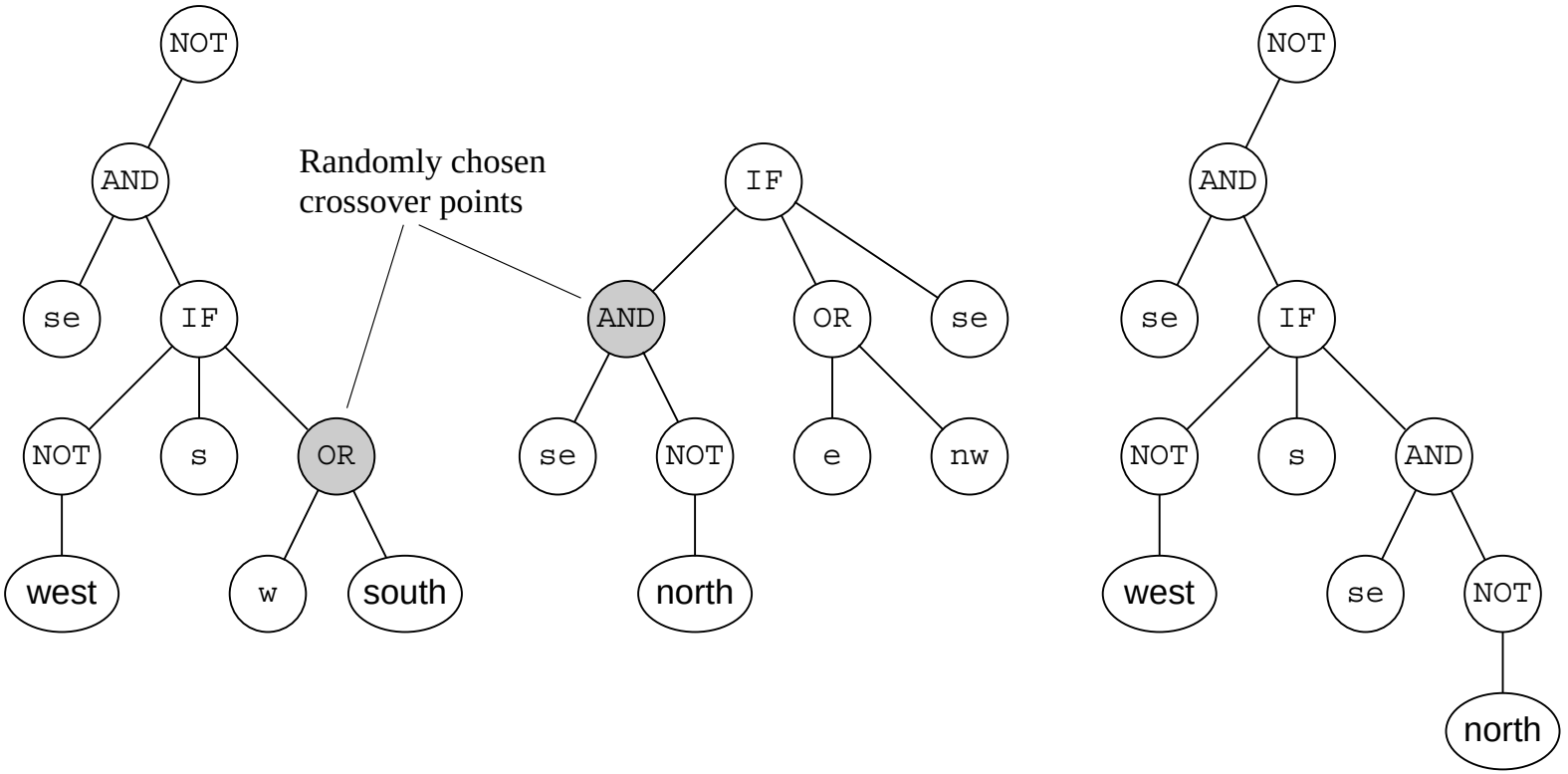
The GP Process

Generation 0: 5000 random programs.

Procedure for producing generation (n+1) from generation n:

- Copy 10% of the programs from generation n to generation n+1. These programs are chosen by the following *tournament selection* process: 7 programs are randomly selected from the population, and the most fit of these seven is chosen.
- The rest (90%) are produced from generation n by a crossover operation as follows: a mother and a father is chosen from generation n by the tournament selection process, and a randomly chosen subtree of the father is used to replace a randomly selected subtree of the mother. See next page for an example.
- Sometimes a mutation operator is also used: it selects a member from generation n by the tournament selection process, and then a randomly chosen subtree of it is deleted and replaced by a random tree. When used, the mutation operation is used sparingly (maybe 1% rate).

An Example of Crossover Operation



Mother program

Father program

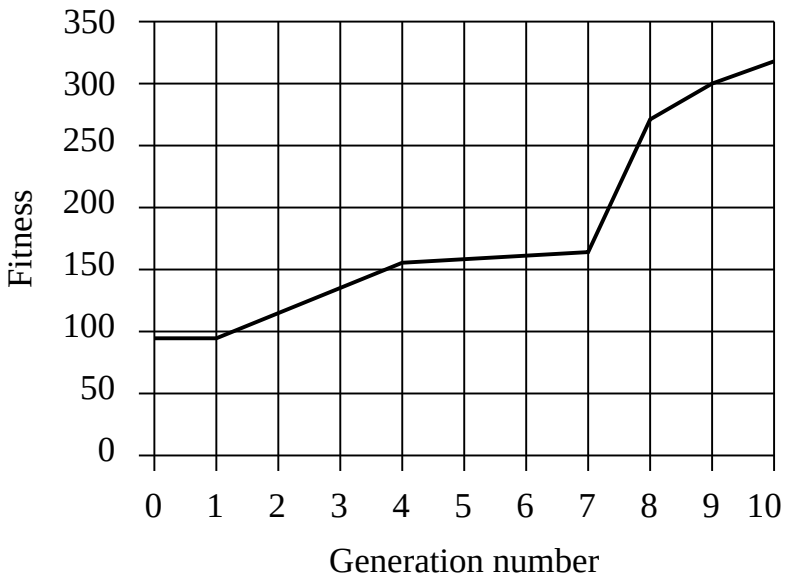
Child program

Performance of GP

The performance of GP depends on:

- the size of generation 0;
- the copy rate, crossover rate, and mutation rate;
- the parameters used in the tournament selection process.

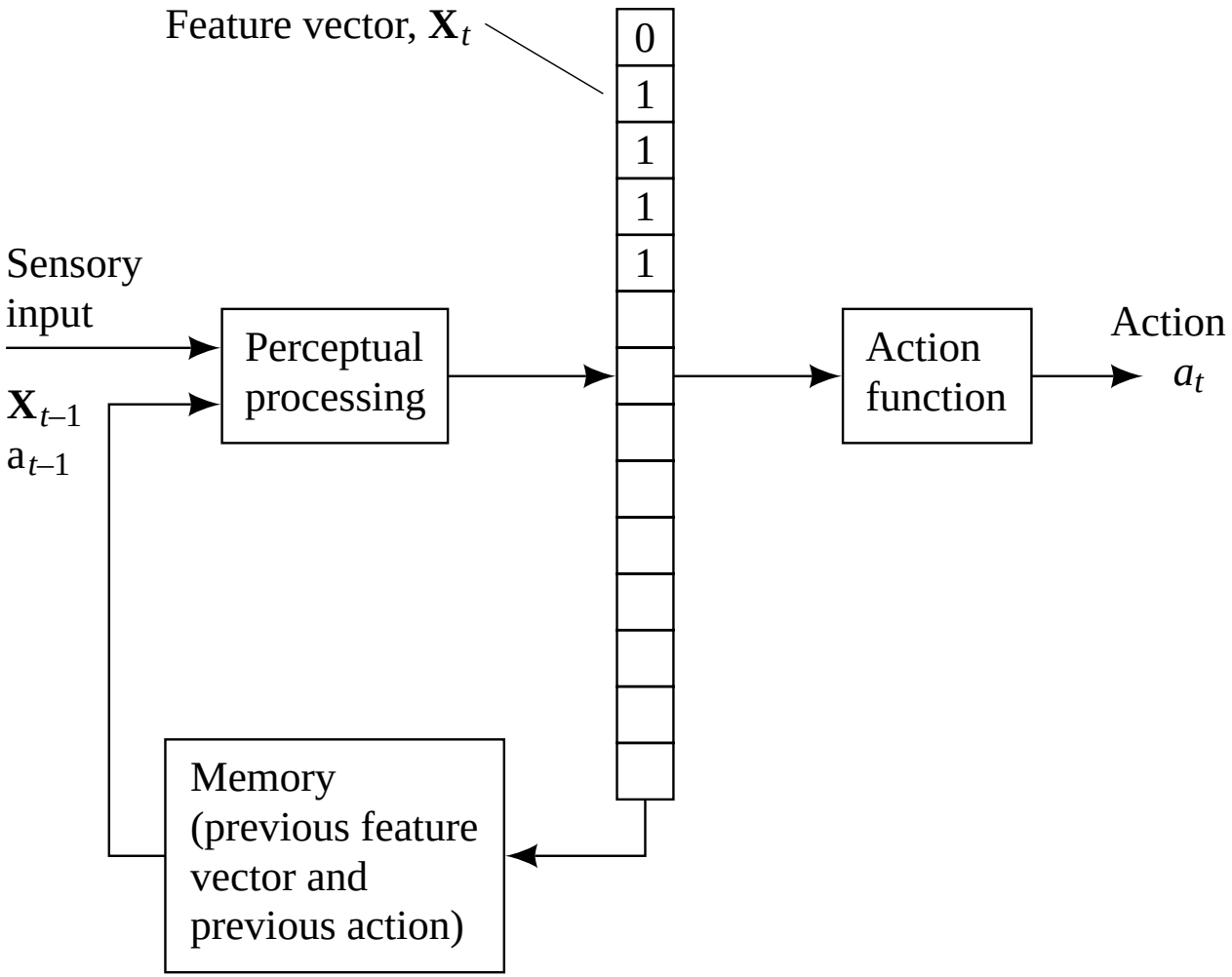
For the wall-following example, it will generate a perfect one after about 10 generations:



State Machines

- Recall that S-R agents have no memory; they just respond to the current stimuli.
- Often one needs more powerful agents. *State machines* are agents who can remember the action that they have just done, and the state that the previous environment is in, and can have some mental states.
- Action function of a state machine is then a mapping of the current sensory inputs, the state of the environment at the previous time step, the action that the machine has just taken, and the current mental states.

Basic Architecture of State Machines



Sensory-Impaired Bounday-Following Robot

Consider again our boundary-following robot, assume now that it's somewhat sensory-impaired: its sensory inputs are only (s_2, s_4, s_6, s_8). Can you design an S-R agent for following a bounday based only on these four sensors?

Let $w_1 - w_8$ be the features defined as:

- $w_i = s_i$ when $i = 2, 4, 6, 8$;
- $w_1 = 1$ iff at the previous time step, $w_2 = 1$, and the robot moved east;
- $w_3 = 1$ iff at the previous time step, $w_4 = 1$, and the robot moved south;
- $w_5 = 1$ iff at the previous time step, $w_6 = 1$, and the robot moved west;
- $w_7 = 1$ iff at the previous time step, $w_8 = 1$, and the robot moved north.

Using these features, the following production system will do the job:

$$
\begin{aligned}
w_2\overline{w4} &\rightarrow \textit{east}, & w_4\overline{w6} &\rightarrow \textit{south},\\
w_6\overline{w8} &\rightarrow \textit{west}, & w_8\overline{w2} &\rightarrow \textit{north},\\
w_1 &\rightarrow \textit{north}, & w_3 &\rightarrow \textit{east},\\
w_5 &\rightarrow \textit{south}, & w_7 &\rightarrow \textit{west},\\
1 &\rightarrow \textit{north}
\end{aligned}
$$