Evolutionary Computation

Genetic Algorithms / Programming

CIS 32

Functionalia

HW 1 - What happened? (2/5 return?) - Due Wednesday 11:59pm

HW 2 Is Out - DUE **SATURDAY** before the MIDTERM.

Thursday - Project 1 Work-Day - will meet HERE rather than **0317N**

EVENING TEA: Next Monday, 5pm to 7pm, 0317 N

Today:

Reinforcement Learning - Example

Genetic Algorithms / Programming

Rewards not goals

For many tasks agents don't have short term goals, but instead accrue *rewards* over a period of time.

Instead of a plan, we want a policy π which says how the agent should act over time.

Typically this is expressed as what action should be carried out in a given state.

Express the reward an agent gets as $r(n_i, a)$

$$r(n_i, a) = -c(n_i, n_j) + \rho(n_j)$$

special reward for being in state n_j

We want an optimal policy π^* which maximizes the (*discounted*) reward at every node.

Finding the Optimum Policy

One (non-ideal) solution is to search through all policies (randomly) until a good one is discovered.

Instead, given a certain policy, one can calculate the **value** of a node - the reward an agent will get if it starts at that node and follows the policy.

Agent at n_i and follows the policy to n_j, then the agent can expect this reward (in the long-term):

$$V^{\pi}(n_i) = r(n_i, \pi(n_i)) + \gamma V^{\pi}(n_j)$$

discounting factor - adds a little long-term goal

Value Iteration

The optimum policy then gives us the action that maximizes this reward:

$$V^{\pi^*}(n_i) = \max_a \left[r(n_i, a) + \gamma V^{\pi^*}(n_j) \right]$$

If we knew what the values of the nodes were under π^*, then we could easily compute the optimal policy:

$$\pi^*(n_i) = \text{argmax}_a \left[r(n_i, a) + \gamma V^{\pi^*}(n_j) \right]$$

The problem is that we don't know these values.

But we can find them out using *value iteration*.

We start by guessing (randomly is fine) an estimated value *V(n)* for each node.

Approximating the Estimated Values

Then when we are at n_i we pick the action to maximize:

$$\text{argmax}_a \left[r(n_i, a) + \gamma V(n_j) \right]$$

that is the best thing given what we currently know.

We then update $V(n_i)$ by:

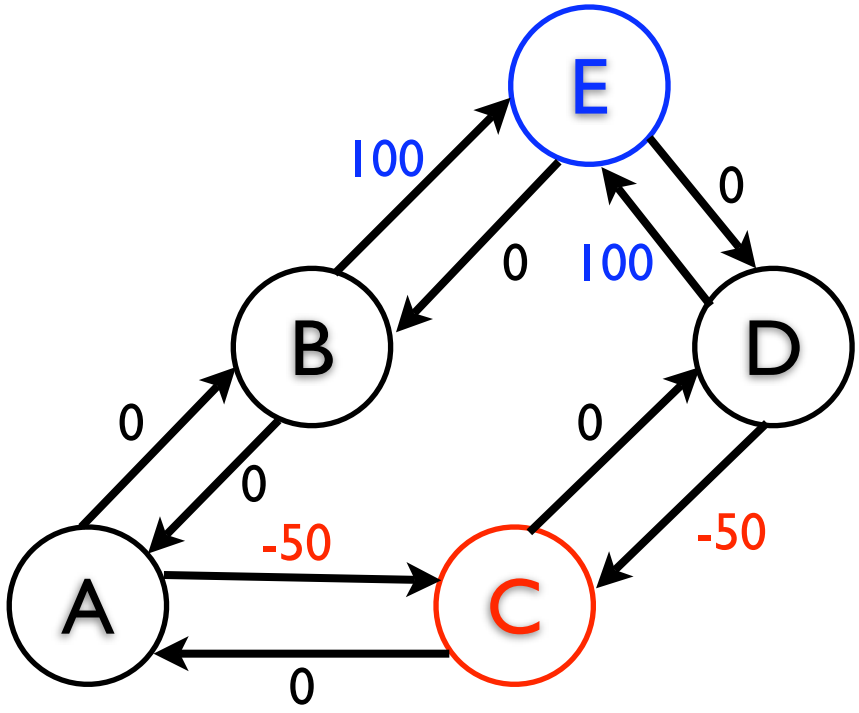
$$V(n_i) := (1 - \beta) V(n_i) + \beta \left[r(n_i, a) + \gamma V(n_j) \right]$$

Progressive iterations of this calculation make $V(n)$ a closer and closer approximation to $V^{\pi^*}(n_i)$

Intuitively this is because we replace the estimate with the actual reward we get for the next state (and the next state and the next state).

Having the factor $0 < \beta < 1$ is geared towards actions with random effects and random awards. (It sets the rate of approach to the optimum Values). In deterministic domains, we can set $\beta = 1$.

Example Q-Learning : Simplified Reinforcement Learning



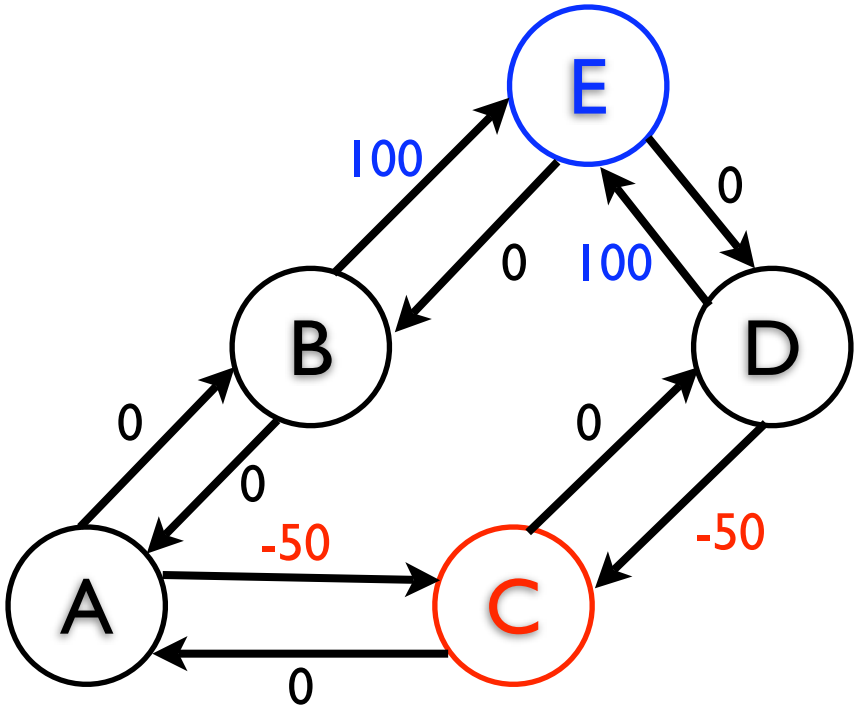
As an example we are going to look at **Q-Learning** a simplified version of Value Iteration.

Consider this State-Space with the actions being traversing to the next state.

Except for the action of moving to state C, the cost of all actions is zero.

Being in state E yields an immediate reward of 100.

Example Q-Learning



state \ action	A	B	C	D	E
A	-	0	-50	-	-
B	0	-	-	0	100
C	0	-	-	0	-
D	-	0	-50	-	100
E	-	0	-	0	-

We can represent the State Space in a Rewards/Cost matrix, which maps state-action pairs to their reward/cost.

A Reward/Cost of '-' indicates that there is no edge on the State-Space graph between the two States.

Example Reinforcement Learning

state \ action	A	B	C	D	E
A	-	0	-50	-	-
B	0	-	-	0	100
C	0	-	-	0	-
D	-	0	-50	-	100
E	-	0	-	0	-

state \ action	A	B	C	D	E
A	0	0	0	0	0
B	0	0	0	0	0
C	0	0	0	0	0
D	0	0	0	0	0
E	0	0	0	0	0

Q-Values Matrix

We keep track of a **Q-Values Matrix : Q(state, action)**

...which are the Expected Values ($V(n_i)$ mentioned before) (but the "states" are written as state-action pairs).

Learning the Q-Matrix

With a Discounting factor set to 0.8 in our case.

We will now calculate Q-values for the state by:

1. Starting in a random initial state.

2. Choose a random action.

3. Update the Q-value for that state:

$$Q(state, action) = Reward(state, action) + Discount * Max\{Q(next\text{-}state, all\ actions\}$$

4. Go to the appropriate state.

5. If we are not at the "GOAL" state (if there is a GOAL state) GO TO **2.**

state \ action	A	B	C	D	E
A	-	0	-50	-	-
B	0	-	-	0	100
C	0	-	-	0	-
D	-	0	-50	-	100
E	-	0	-	0	-

Reward

state \ action	A	B	C	D	E
A	0	0	0	0	0
B	0	0	0	0	0
C	0	0	0	0	0
D	0	0	0	0	0
E	0	0	0	0	0

Q-Matrix

Set Goal to **E** (so we have a place to stop our episode).

Randomly Select State B as our Initial State, and we'll randomly select goto-E as our action.

$Q(B,E) = \text{Reward}(B,E) + \text{Discount}*(\text{Max}\{Q(E,B), Q(E,D)\})$

$Q(B,E) = 100 + (0.8) * (\text{Max}\{0,0\}) = 100$

state \ action	A	B	C	D	E
A	-	0	-50	-	-
B	0	-	-	0	100
C	0	-	-	0	-
D	-	0	-50	-	100
E	-	0	-	0	-

Reward

state \ action	A	B	C	D	E
A	0	0	0	0	0
B	0	0	0	0	100
C	0	0	0	0	0
D	0	0	0	0	0
E	0	0	0	0	0

Q-Matrix

We are at the GOAL **E** state. Now we will start again.

Randomly Select State **A** as our Initial State, and we'll randomly select goto-**B** as our action.

$Q(A,B) = \text{Reward}(A,B) + \text{Discount}*(\text{Max}\{Q(B,A), Q(B,D), Q(B,E)\})$

$Q(B,E) = 0 + (0.8) * (\text{Max}\{0,0,100\}) = 80$

Moving towards Convergence on the Q-Matrix

Now, if we did many parallel searches (i.e. starting at every node, and considering every action with the Initial Empty Q Matrix).

You would get this Q-matrix of values:

state \ action	A	B	C	D	E
A	0	80	-50	0	0
B	0	0	0	80	100
C	0	0	80	0	0
D	0	80	-50	0	100
E	0	80	0	80	0

Q-Matrix

Moving towards Convergence on the Q-Matrix

Eventually the Q-Matrix would converge to something that the now Learned-Agent would be able to use:

state \ action	A	B	C	D	E
A	-	222	128	-	-
B	178	-	-	222	278
C	178	-	-	222	-
D	-	222	128	-	278
E	-	222	-	222	-

Q-Matrix

Moving towards Convergence on the Q-Matrix

Eventually the Q-Matrix would converge to something that the now Learned-Agent would be able to use:

state \ action	A	B	C	D	E
A	-	222	128	-	-
B	178	-	-	222	278
C	178	-	-	222	-
D	-	222	128	-	278
E	-	222	-	222	-

Q-Matrix

With this Estimated Value Matrix of Current State and Action Pairs (Q-Values), what sequence of steps would this agent take starting at state **A** and taking two steps?

Discoveries / Metaphors from Biology

Lamarck and others:

- Species "transmute" over time (gene's change during organism's lifetime)

Darwin and Wallace:

- Consistent, heritable variation among individuals in population
- Natural selection of the fittest

Mendel and genetics:

- A mechanism for inheriting traits
- genotype → phenotype mapping

Watson and Crick:

- Information strings! (DNA)

Nature is good at evolving robust agents.

Can we borrow such mechanisms to build artificial agents?

• We will look at two models:

– Genetic algorithms

– Genetic programming

RL Additional Resources

Reinforcement Learning Examples

TD-Gammon: A very successful program that can play BackGammon. Agent begins knowing nothing about BackGammon and plays 2 million games. (Reaches a draw against humans)

Elevator Scheduling: Large skyscraper Elevator Scheduling is a complex problem. Through simulation on the input (all buttons pressed and where the elevators currently are), an agent can maximize the throughput of people through the system. (Better than FSM controls)

Additional Resources

If you are interested in more Reinforcement Learning, a Paper is online (accessible from the syllabus section).

Q-Learning example is based on a Reinforcement Learning Tutorial:

`http://people.revoledu.com/kardi/tutorial/ReinforcementLearning/index.html`

Genetic algorithms

```
genetic-algorithm(population,fitness)
{
   repeat
   {
      parents := selection(population,fitness)
      population := reproduction(parents)
   }
   until(enough fit individuals)
      return(fittest individual)
}
```

Genetic Algorithms As Search

This is *just* a fancy way of doing search.

• We code some part of the agent (e.g. action selection function) and decide how to do:

- selection
- reproduction.

• When we have a bunch of individuals (as we typically do), each individual represents a state in the state-space of possible individuals.

• Establishing and evaluating a population is a (massively) parallel search though this space.

Components of the Genetic Algorithm

- To use the approach we have to instantiate:
  - What is the fitness function?
  - How is an individual represented?
  - How are individuals selected?
  - How do individuals reproduce?
- While these are to some extent domain dependent, we will look at some typical ways of doing this.

Fitness function

- The fitness function is the most domain dependent item.
- It is a function that takes an individual as an argument and returns a real number.
- In the example of our wall following robot a function could be:
  - The average number of moves out of *n* for which the robot makes the right action selection.
  - The average number of moves out of *n* for which the robot is adjacent to the boundary.
- Fitness functions often take time to evaluate. (i.e. Our robot would have to run a trial to see how it behaves)

Representation

• The classic representation is one in which features are coded as a binary "chromosome".

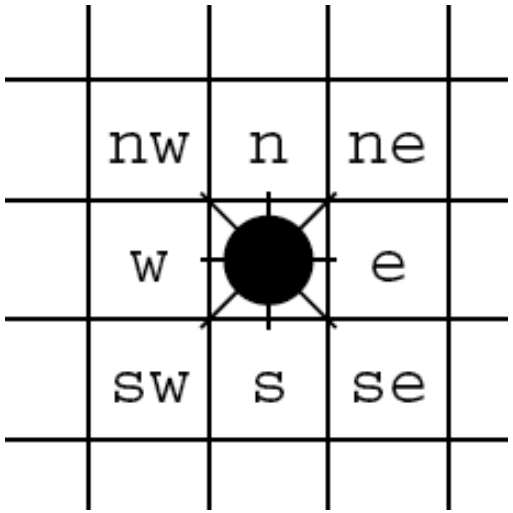
(i.e. we code a sequence like 01110110 rather than AATGTCAT.)

• In our robot example, we could code up the action representation as a list of condition/action pairs:

- One possible combination of sensor readings (followed by)
- The appropriate action.

• Sensor readings could be

strings `n, ne, . . . , nw`.



• Actions could be two digit binary numbers, 00 = `north` etc.

Selection

- Selection is usually a two stage process.
- First we limit the population:
  - Cull unfit individuals to limit the population size.
- Then we select individuals to breed:
  - Random selection weighted towards fit individuals;
  - With replacement (so very fit individuals can breed several times).

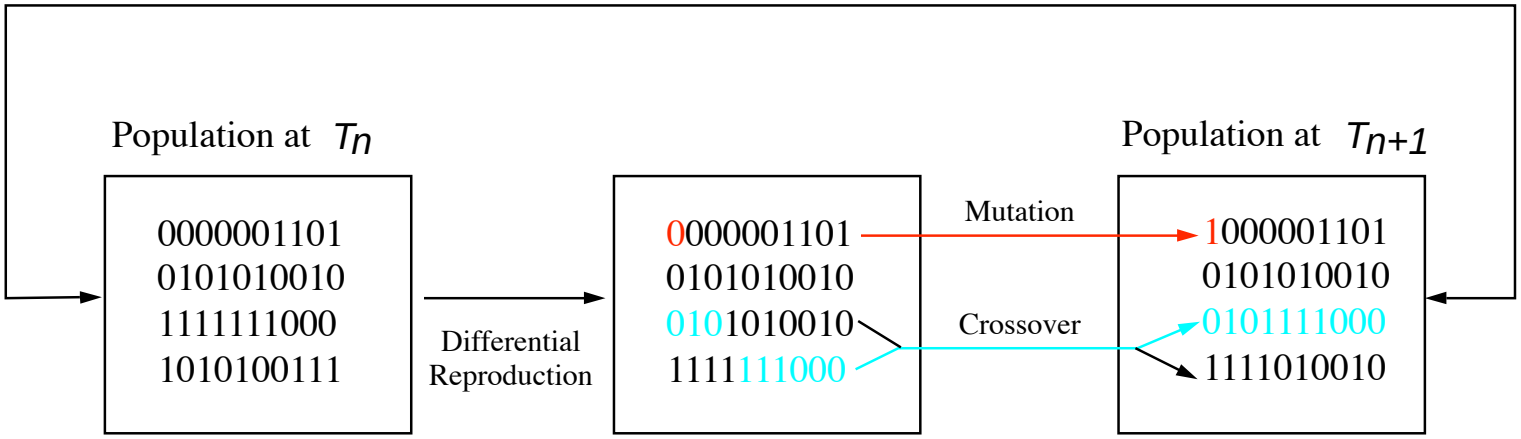
Reproduction

- Two basic parts to reproduction:
  - Crossover
  - Mutation.
- First take two parents P1 and P2, and pick a number *n* between 1 and *N* = length of "chromosome".
- Create two "children", C1 and C2.
- C1 is the first *n* bits of P1 and the last *N* - *n* bits of P2.
- C2 is the first *n* bits of P2 and the last *N* - *n* bits of P1.

Crossover and Mutation

- Cross-over is analogous to state-space transitions in state-space search.
- Taking fit individuals and combining their features is a form of best-first search.
- It makes small "hill climbing" steps up the fitness function.
- However it can get stuck in local maxima.
- Mutation is a way of "jumping" to new areas of search space.
- We "mutate" random bits by flipping them.

The Art of Genetic Algorithms



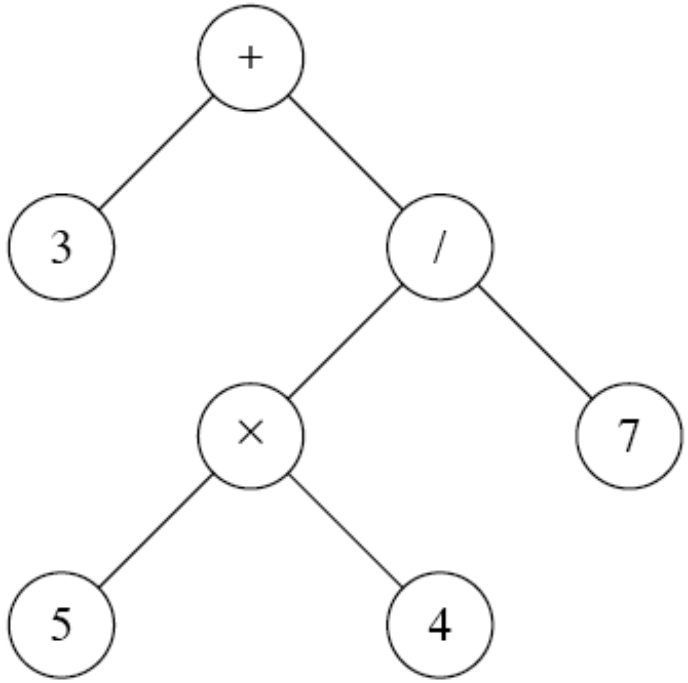
F (0000001101) = 0.000
F (0101010010) = 0.103
F (1111111000) = 0.030
F (1010100111) = – 0.277

- Again we have a lot of possible parameters to play with:
  - Fitness rating;
  - Selection probability;
  - Mutation rate;
  - Crossover point;
  - etc.

Genetic programming

- Genetic algorithms only allow us to evolve some part of the agent program.
- We need to code up the "chromosome" and decode to get the agent itself.
- However, we can do evolution on more complex objects.
- In genetic programming we do evolution on programs themselves.

• We can't get completely away from some representation:

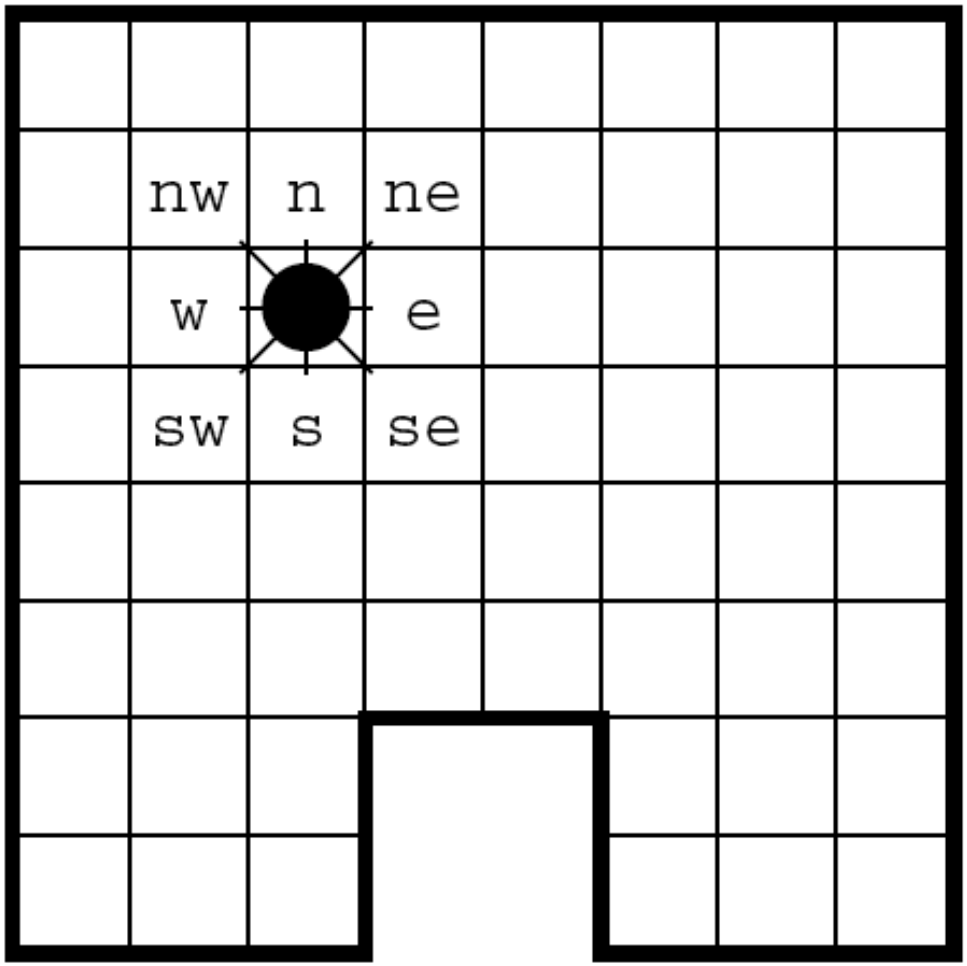


• However, in a suitable language (Lisp) we can execute such functions directly:

```
(+ 3 (/ (x 5 4) 7))
```

Other languages will need a little translation.

Let's look at how GP can be used to evolve the wall following robot.



- We build the program up from four primitive functions:

  1. **`AND(x, y)`** = 0 if x = 0; else y
  2. **`OR(x, y)`** = 1 if x = 1; else y
  3. **`NOT(x)`** = 0 if x = 1; else 1
  4. **`IF(x, y, z)`** = y if x = 1; else z

and four actions:

  1. **`north`** move one cell up the grid
  2. **`east`** move one cell right in the grid
  3. **`south`** move one cell down the grid
  4. **`west`** move one cell left the grid

Note

We must ensure that all expressions and sub-expressions have values for all possible arguments, or terminate the program.

This ensures that any tree constructed so a function is correctly formed will be an executable program.

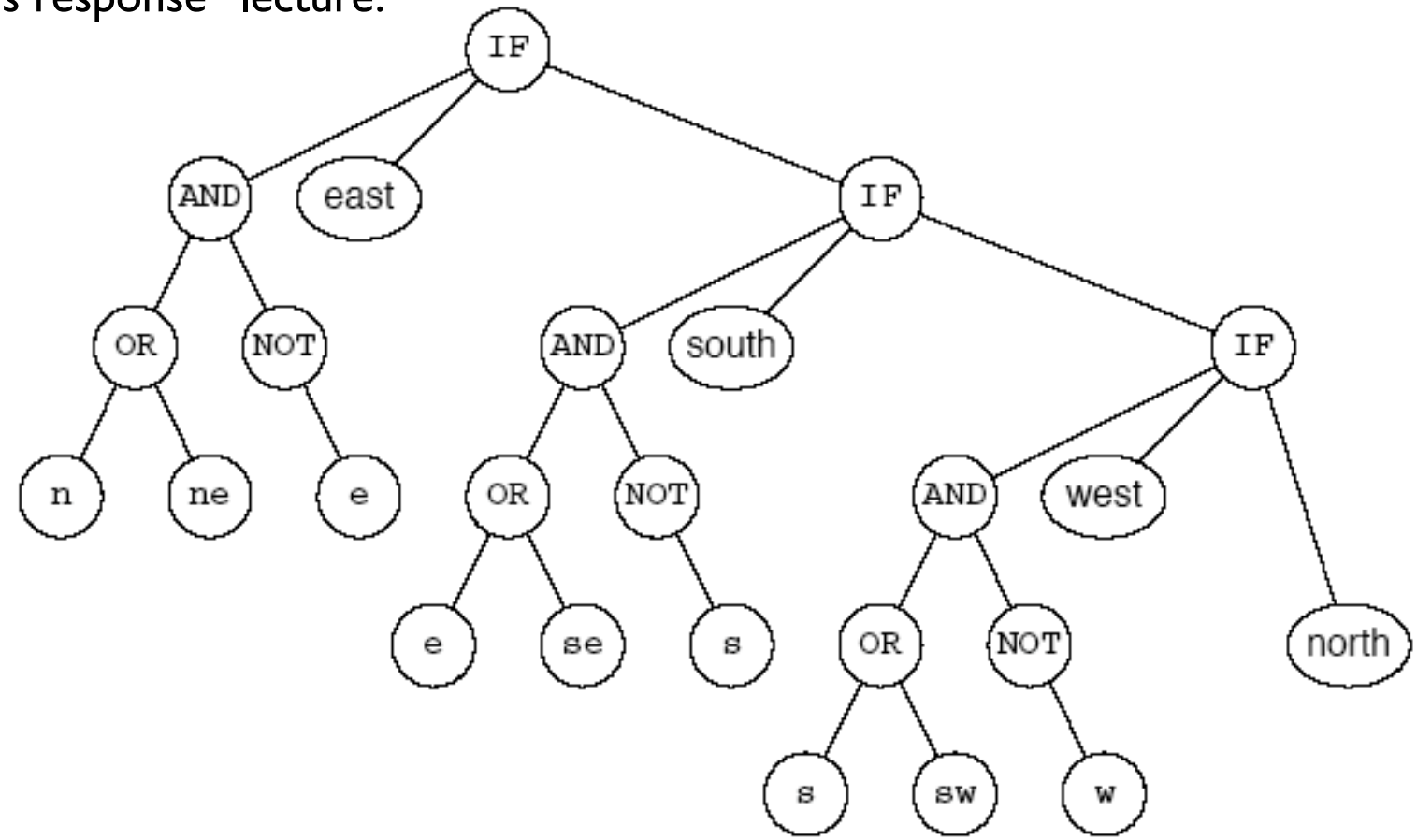
Even if the program is executable, it may not produce "sensible" output.

It may divide by zero, or generate a negative number where only a positive number makes sense (as when setting a price).

So we always need to have some kind of error handling to deal with the output of individual programs..

• To give us an idea of what we are looking for, the following slide gives an example program in the GP tree-format.

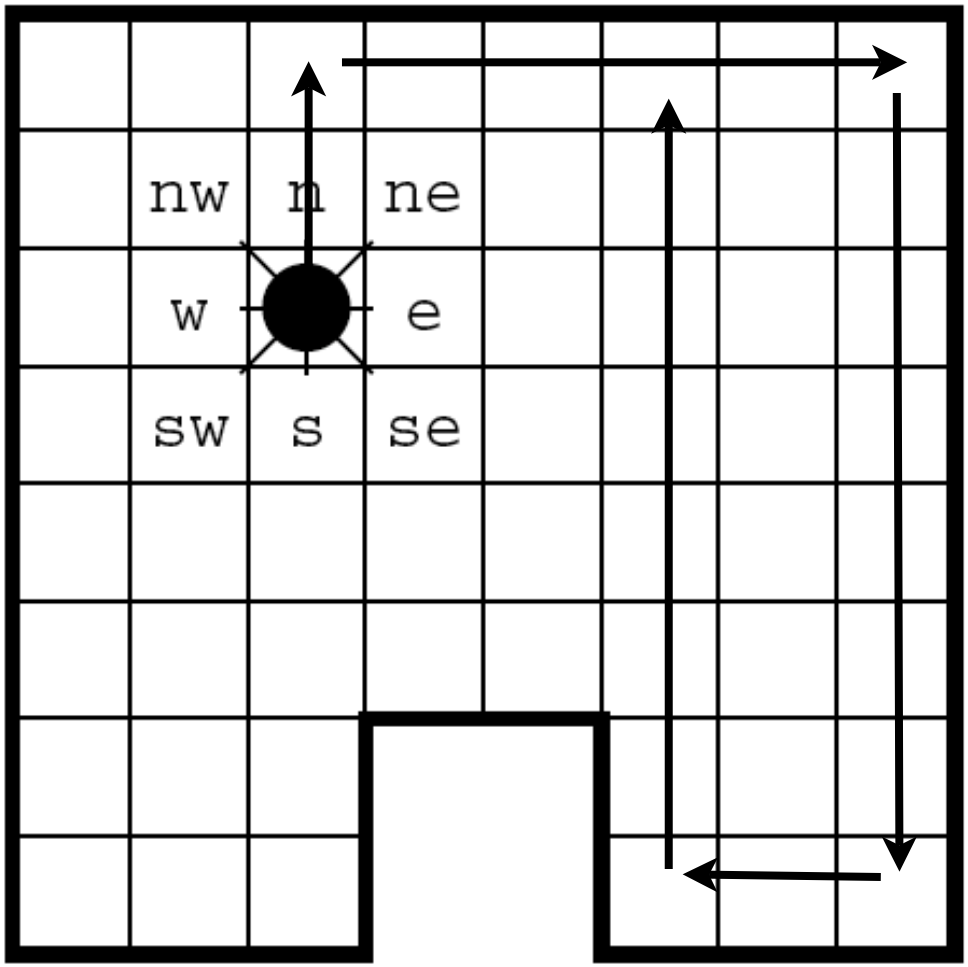
• This program (check it) implements the same wall following program that we looked at in the "stimulus response" lecture.




```
(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))))
```

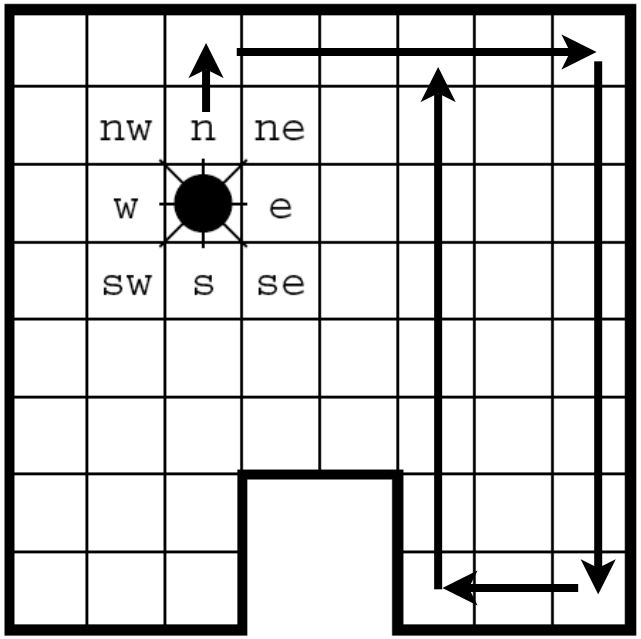
Fitness

- The basic way we do GP is like GA.
- What kind of fitness function could we use for the boundary follower?



Fitness

- The basic way we do GP is like GA.
- What kind of fitness function could we use for the boundary follower?



Example Fitness Function:

1. Run the Program 60 steps, count the number of boundary cells that are visited. (32 would be the highest)

2. Do 10 trials, each starting in a random spot.

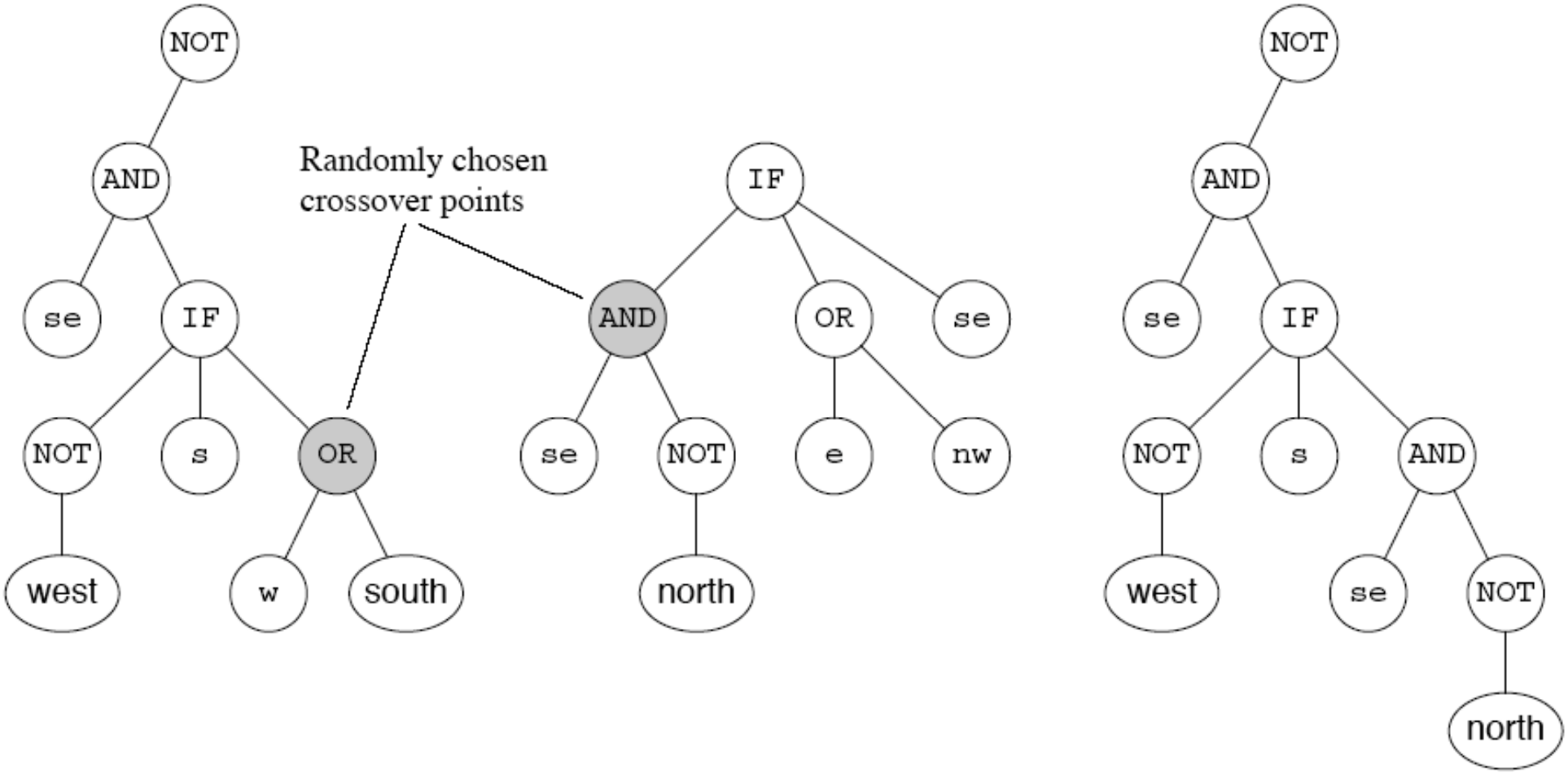
3. 320 is the Highest Possible Fitness (a boundary-follower that always visits all 32 boundary cells). 0 is the worst possible fitness.

Reproduction

- Do selection of the most fit (top 10% of the batch)
- Breed them (90% of the next generation are the children)
- But how do we breed programs?

Reproduction

A randomly selected subtree from the father replaces a randomly selected subtree from the mother.



Mother program

Father program

Child program

Running the first trial

- The GP-format is somewhat clumsy.

```
(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))))
```

- However, as we shall see, this program is relatively compact when compared with the programs that will be generated by GP.

- We begin to select from 5000 programs in the first batch.

Tournament Selection

- Then we need to breed by **tournament selection**:

  Take 500 (10%) fittest programs and add them to the next generation.

  Pick 7 (<1%) programs at random and add them to the next generation.

- **Reproduction**: Then create 4500 (90%) children into the next generation—parents chosen by tournament selection

  (Children Program's subject to basic syntax check).

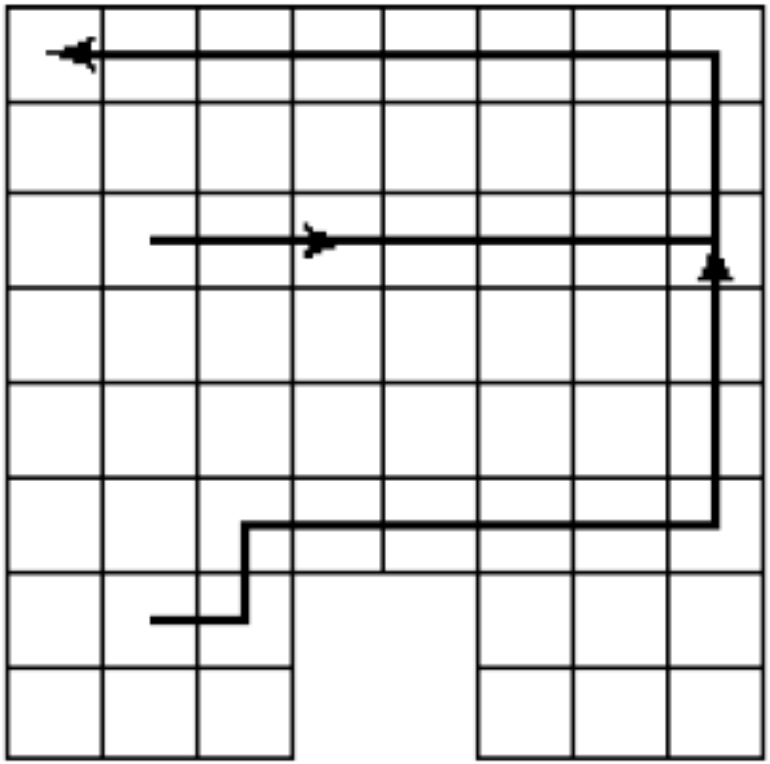
- **Mutation**: Subject a sparse number (~1%) of members from the resultant generation to a **mutation operation**:

  1. Delete a randomly selected subtree.

  2. Replacing a randomly chosen subtree with a random subtree.

Generation 0

The most fit member of the randomly generated initial programs has a fitness of 92, and has the kind of behavior shown below.



Here is the resulting program →

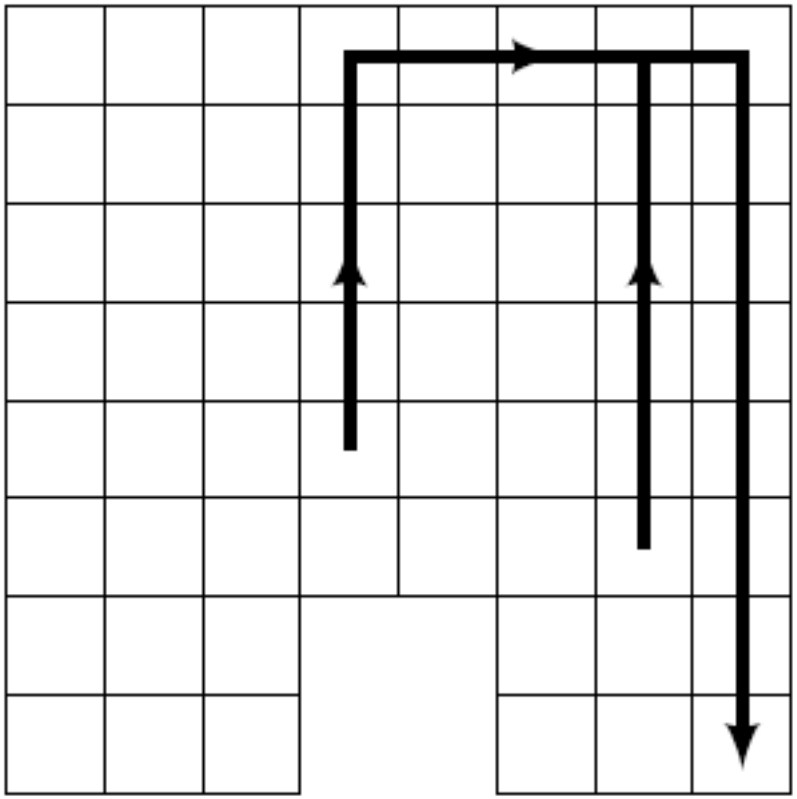
Fittest Program

```
(AND (NOT (NOT (IF (IF (NOT (nw))
                       (IF (e)(north) (east))
                       (IF (west)(0) (south))
                   (OR (IF (nw)(ne)(w))
                       (NOT (sw)))
                   (NOT (NOT (north)))))))
     (IF (OR (NOT (AND (IF (sw)(north)(ne))
                       (AND (south)(1))))
             (OR (OR (NOT (s))
                     (OR (e)(e)))
                 (AND (IF (west)(ne)(se))
                      (IF (1) (e)(e)))))
         (OR (NOT (AND (NOT (ne))(IF (east)(s)(n))))
             (OR (NOT (IF (nw)(east)(s)))
                 (AND (IF (w)(sw)(1))
                      (OR (sw)(nw)))))
         (OR (NOT (IF (OR (n)(w))
                      (OR (0)(se))
                      (OR (1)(east))))
             (OR (AND (OR (1)(ne))
                      (AND (NW)(east)))
                 (IF (NOT (west))
                     (AND (west)(est))
                     (IF (1)(north)(w)))))))
```

Not so optimal.

Generation 2

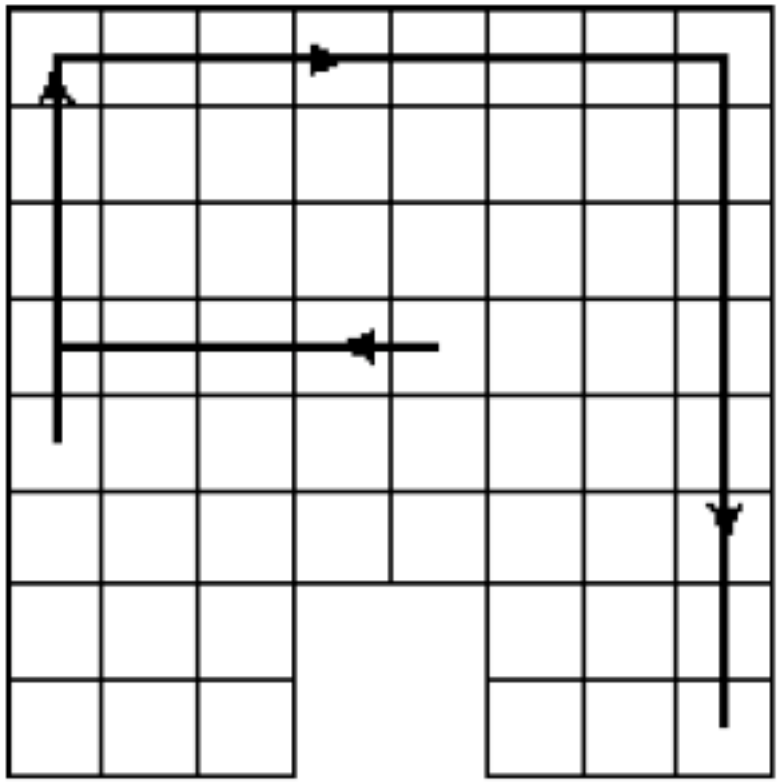
The most fit member of generation 2 has fitness 117.



```
(NOT (AND (IF (ne)
              (IF (se)(south)(east))
              (north))
          (NOT (NOT (e)))))
```

Generation 6

The most fit member of generation 6 has fitness 163.



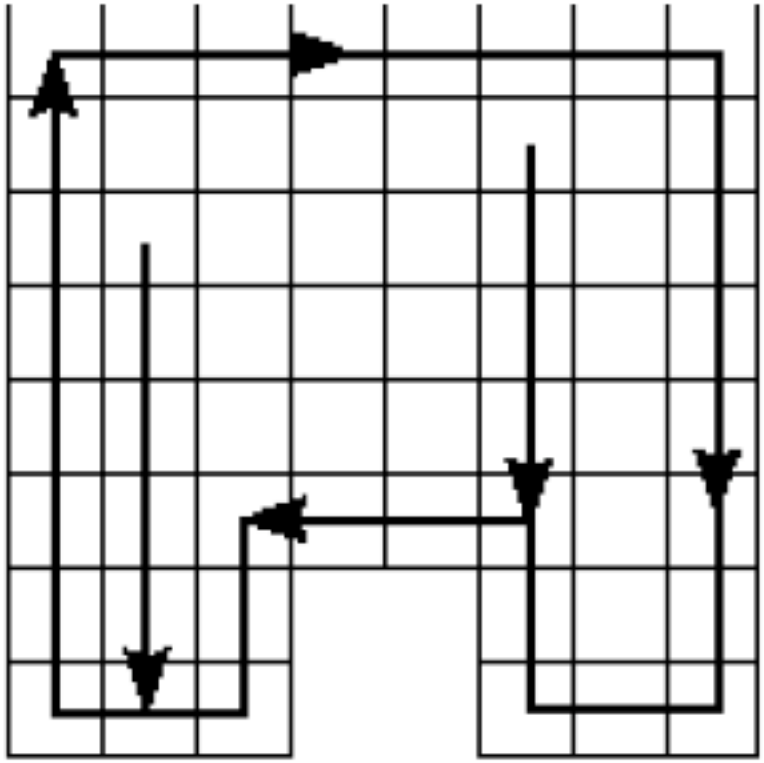
The program follows the wall perfectly, but gets stuck in the bottom right-hand corner.

the best program from Generation 6:

```
(AND (NOT (NOT (IF (IF (NOT (nw))
                       (IF (e)(north) (east))
                       (IF (west)(0) (south))
                   (OR (IF (nw)(ne)(w))
                       (NOT (sw)))
                   (NOT (NOT (north)))))))
     (IF (OR (NOT (AND (IF (sw)(north)(ne))
                       (AND (south)(1))))
             (OR (OR (NOT (s))
                     (OR (e)(e)))
                 (AND (IF (west)(ne)(se))
                      (IF (1) (e)(e)))))
         (OR (NOT (AND (NOT (ne))(IF (east)(s)(n))))
             (OR (NOT (IF (nw)(east)(s)))
                 (AND (IF (w)(sw)(1))
                      (OR (sw)(nw)))))
         (OR (NOT (IF (OR (n)(w))
                      (OR (0)(se))
                      (OR (1)(east))))
             (OR (AND (OR (1)(ne))
                      (AND (NW)(east)))
                 (IF (NOT (west))
                     (AND (west)(est))
                     (IF (1)(north)(w)))))))
```

Generation 10

The most fit member of generation 10 has fitness of close to the maximum 320.



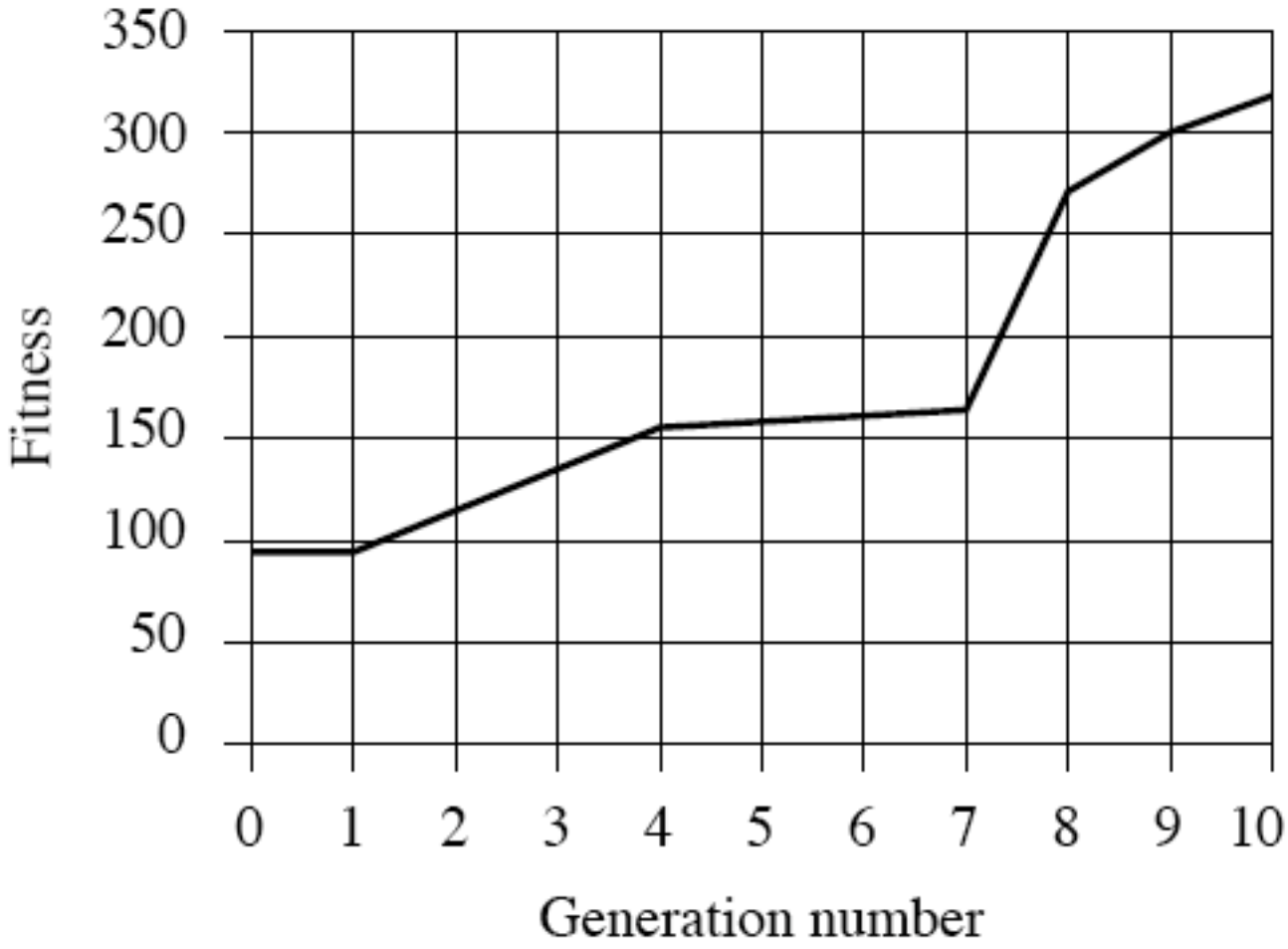
The program follows the wall perfectly, heading south until it reaches the boundary.

the best program from Generation 10

```
(IF (IF (IF (se)(0)(ne))
        (OR (se)(east))
        (IF (OR (AND (e)(0))
                (sw))
            (OR (sw)(0))
            (AND (NOT (NOT (AND (s)(se))))
                 (se))))
    (IF (w)
        (OR (north)
            (NOT (NOT (s))))
        (west))
    (NOT (NOT (NOT (AND (IF (NOT (south))
                            (se)
                            (w))
                        (NOT (n)))))))
```

Graph

This graph shows how the fitness of individuals grows quite sharply over the ten generations.



Genetic Algorithms for Problem Solving

GA/GP is a stochastic process and arrive at an optimized solution.

(Mutation/Crossover - *simulated annealing* in system's engineering)

Best suited for those tasks which cannot be solved through analytical means, or problems where efficient ways of solving them have not been found (Heitkoetter, Joerg and Beasley, Daveid, 1995)

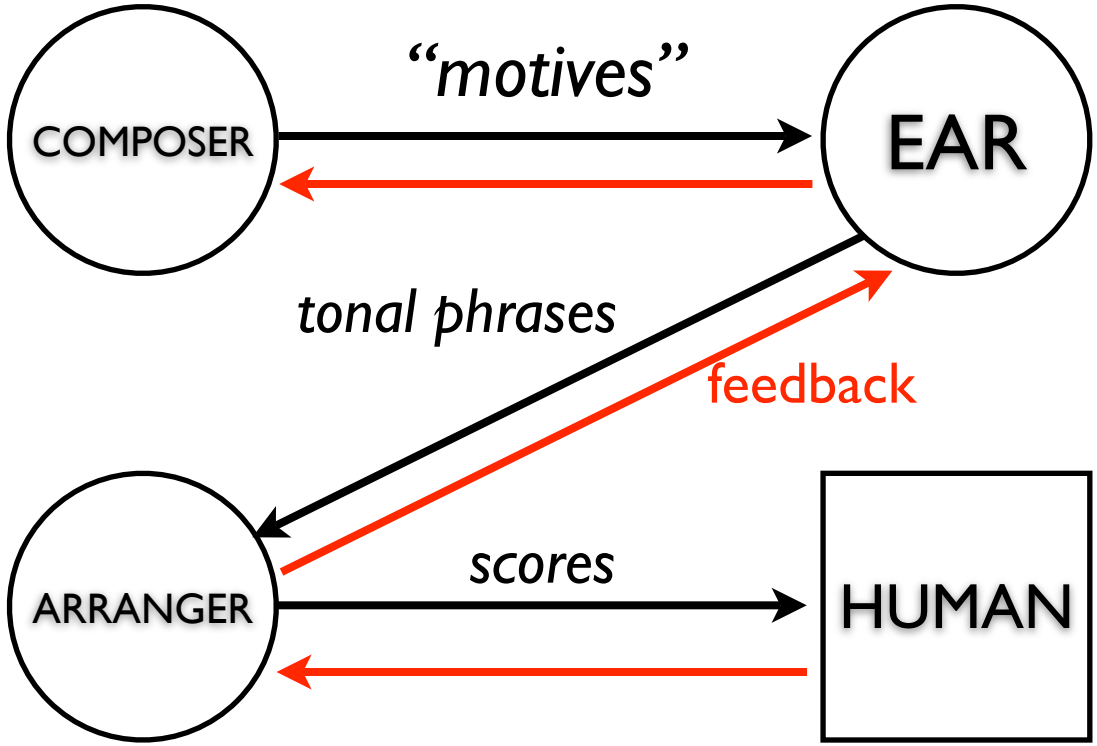
Successful Case Studies:

Timetabling: University of Edinburgh, scheduling for student's exams. Its fitness function take all of the student's schedules and rates the fitness of the current exam timetable. (Abramson & Abela 1991)

Scientific Design: Design a turbine blade. The many design factors of a blade - shape, thickness, and it's twist make up the "chromosome". (Taubes, 1997).

... more Genetic Algorithms for Problem Solving

Music Composition: Bruce L Jacob (University of Michigan), 3 modules utilizing Genetic Algorithms.



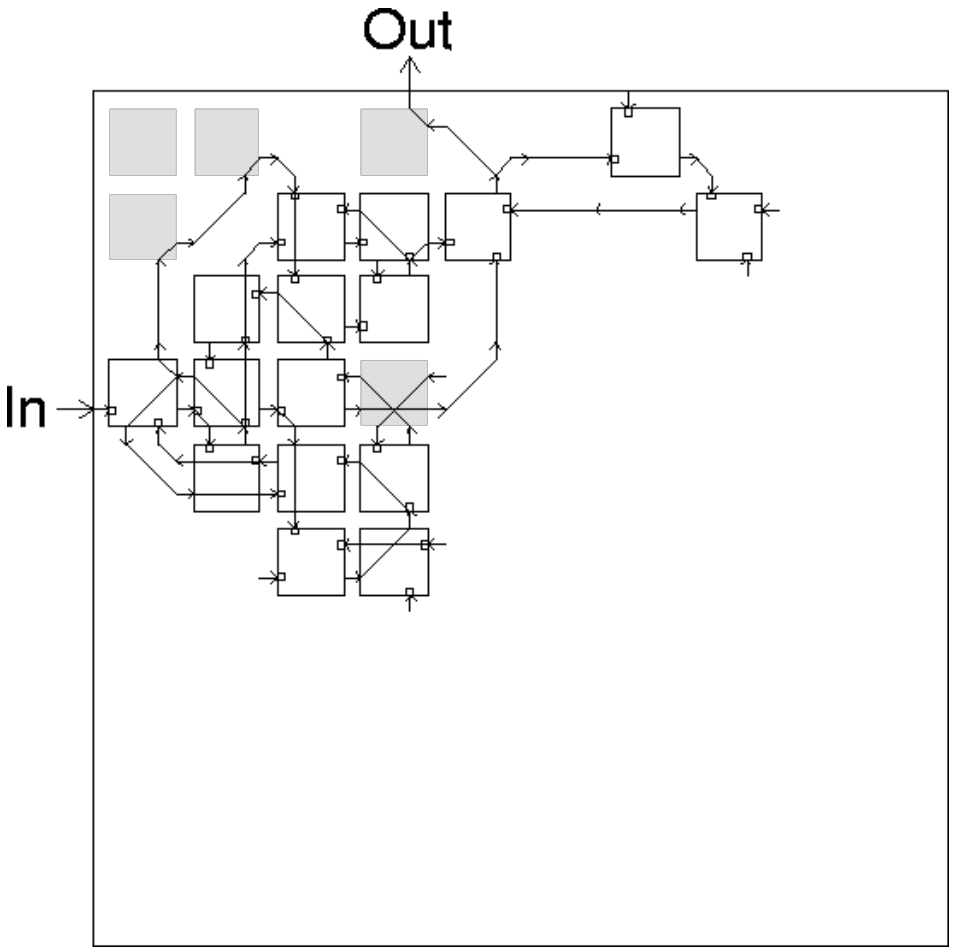
Chromosomes assigned by creator's tastes.

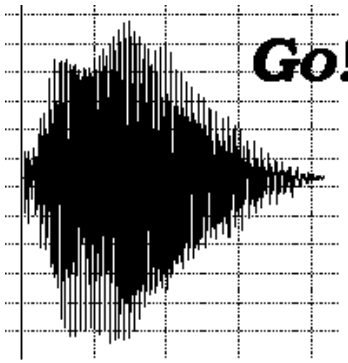
Output is Scored Music : ultimately performed by humans.

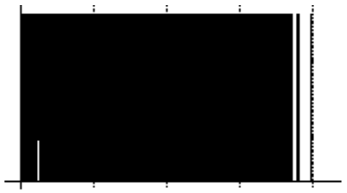
Hardware Evolution

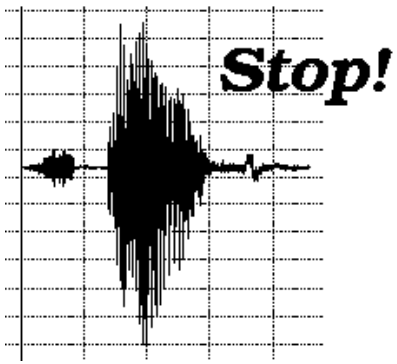
FPGA - Field Programmable Gate Arrays ("soft" logic hardware)

Adrian Thompson (University of Sussex, UK) - evolved configurations of logic gates. Created audio "discriminator" circuit:









Threshold

Idea is to build evolutionary robust hardware logic (*self-healing*)

Summary

- This lecture has introduced *evolutionary computing* techniques.
- These are techniques in which (parts of) agents evolve.
- We looked at two techniques:
  - Genetic algorithms
  - Genetic programming
- Note that evolutionary techniques are sometimes taken to include neural networks.
- Genetic algorithms and genetic programming give us a way to learn in an *unsupervised way*.