Machine Learning Lecture 7

Jakub Uher

February 2020

Evolutionary Algorithms

In this lecture we study about evolutionary algorithms and their use in industry, walking thru the algorithm to understand all steps, including the pseudocode of the algorithm.

- Genetic Algorithms
- Genetic Programming
- Co-evolutionary Algorithms
- Examples

Evolutionary Learning

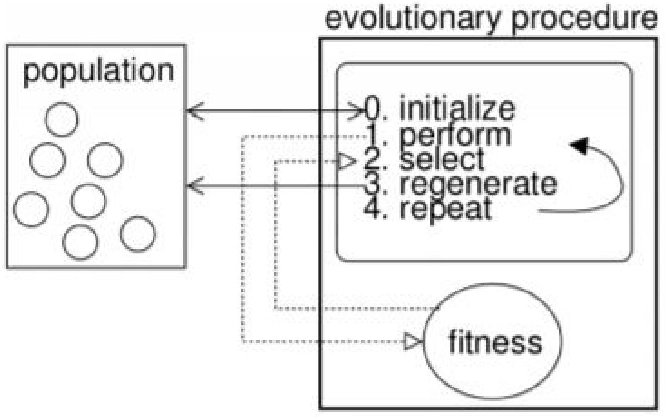


Figure 1: Evolutionary Learning diagram

- An evolutionary algorithm 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 - how to select the best candidates from the population, usually using the fitness function
  - Reproduction - what genes to use when reproducing new cells (for example DNA)
- When we have a bunch of individuals (population) each individual represents a state in the state-space of possible individuals.
- Establishing and evaluating a population can be viewed as a (massively) parallel search through this state space.

Genetic Algorithms

Genetic algorithm was introduced by John Holland in 1962-1975, it's closely related with the survival of the fittest theory, the process involves steps like, selection, evaluation and reproduction.

- Loosely based on the idea of simulated evolution - "survival of the fittest";
- In general, we hypothesize a **population** of candidate solutions to a particular problem.
- Then we **evaluate** each member of the population to decide how good that candidate is at solving the problem
- Then we **select** those members of the population that are the best (the fittest).
- Then we **reproduce** to obtain new members of the population.
- Then we start all over again, hypothesizing with this new set of candidate solutions. Until the fittest value still increases.

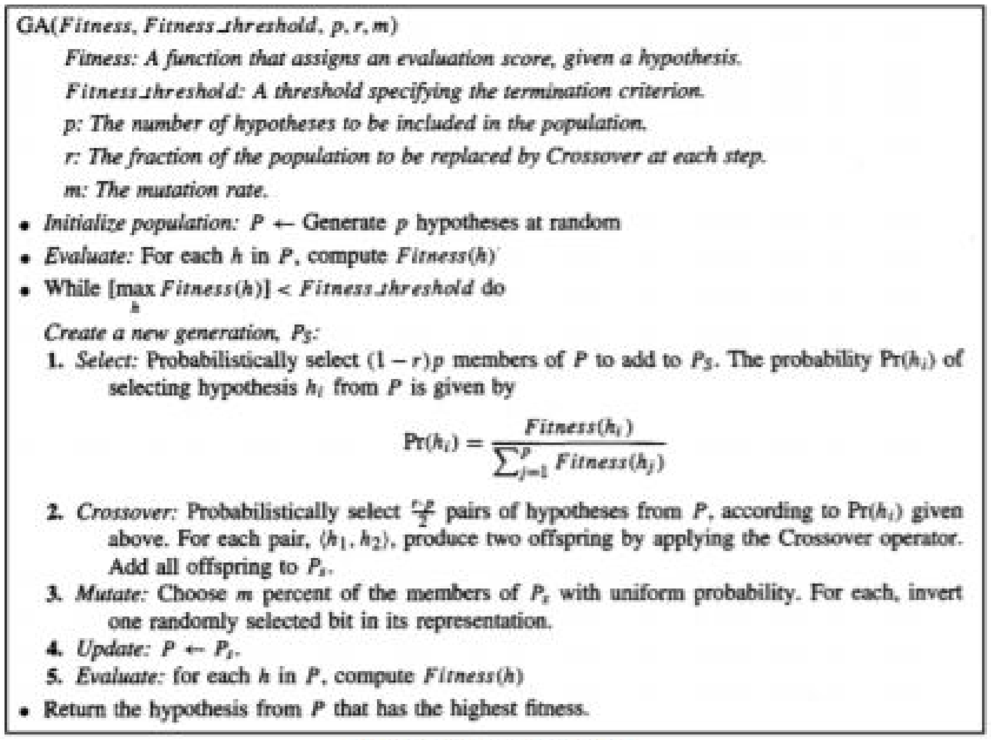
Genetic Algorithms: Advantages

- Genetic Algorithms are good for situations where it is difficult to ascertain the impact of a particular partial solution to a problem.
- Genetic Algorithms easy to implement, each candidate can be evaluated on its own so the process can run in parallel.
- Genetic Algorithms can adapt easily in dynamic environments.

Genetic Algorithms: Components

- **Representation -** how is each candidate represented?
- **Fitness -** how is each candidate evaluated=?
- **Selection -** how are the best candidates chosen?
- **Reproduction -** how are new candidates generated?

Genetic Algorithms: Pseudocode



```
GA($Fitness$, $Fitness\_threshold$, $p$, $r$, $m$)
    Fitness: A function that assigns an evaluation score, given a hypothesis.
    Fitness_threshold: A threshold specifying the termination criterion.
    p: The number of hypotheses to be included in the population.
    r: The fraction of the population to be replaced by Crossover at each step.
    m: The mutation rate.
• Initialize population: $P \leftarrow$ Generate $p$ hypotheses at random
• Evaluate: For each $h$ in $P$, compute $Fitness(h)$
• While $[\max_h Fitness(h)] < Fitness\_threshold$ do
    Create a new generation, $P_S$:
    1. Select: Probabilistically select $(1-r)p$ members of $P$ to add to $P_S$. The probability $\Pr(h_i)$ of
           selecting hypothesis $h_i$ from $P$ is given by
               $\Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)}$
    2. Crossover: Probabilistically select $\frac{r \cdot p}{2}$ pairs of hypotheses from $P$, according to $\Pr(h_i)$ given
           above. For each pair, $\langle h_1, h_2 \rangle$, produce two offspring by applying the Crossover operator.
           Add all offspring to $P_s$.
    3. Mutate: Choose $m$ percent of the members of $P_s$ with uniform probability. For each, invert
           one randomly selected bit in its representation.
    4. Update: $P \leftarrow P_s$.
    5. Evaluate: for each $h$ in $P$, compute $Fitness(h)$
• Return the hypothesis from $P$ that has the highest fitness.
```

[Mitchell, Table 9.1]

Genetic Algorithms: Representation

Example

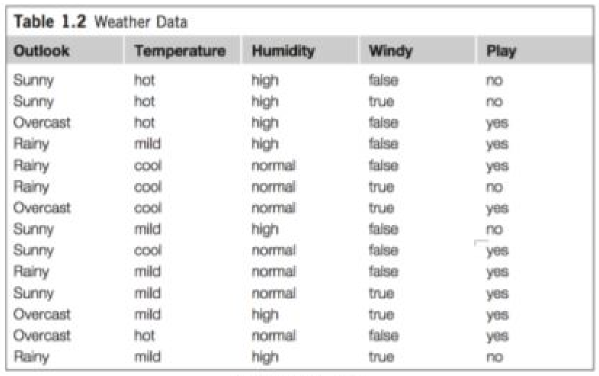


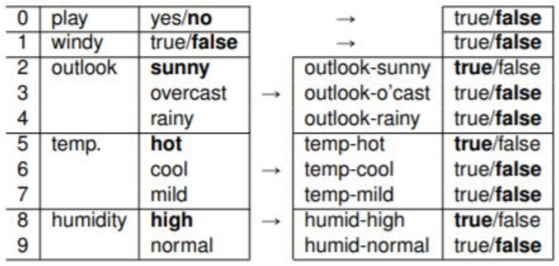
Table 1.2 Weather Data

Outlook	Temperature	Humidity	Windy	Play
Sunny	hot	high	false	no
Sunny	hot	high	true	no
Overcast	hot	high	false	yes
Rainy	mild	high	false	yes
Rainy	cool	normal	false	yes
Rainy	cool	normal	true	no
Overcast	cool	normal	true	yes
Sunny	mild	high	false	no
Sunny	cool	normal	false	yes
Rainy	mild	normal	false	yes
Sunny	mild	normal	true	yes
Overcast	mild	high	true	yes
Overcast	hot	normal	false	yes
Rainy	mild	high	true	no

[WFH Table 1.2]

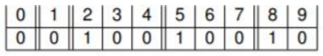
Genetic Algorithms: Representation (cont)

Example continued



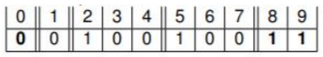
0	play	yes/**no**	→		true/**false**
1	windy	true/**false**	→		true/**false**
2	outlook	**sunny**		outlook-sunny	**true**/false
3		overcast	→	outlook-o'cast	true/**false**
4		rainy		outlook-rainy	true/**false**
5	temp.	**hot**		temp-hot	**true**/false
6		cool	→	temp-cool	true/**false**
7		mild		temp-mild	true/**false**
8	humidity	**high**	→	humid-high	**true**/false
9		normal		humid-normal	true/**false**

Becomes a 10-bit string:



0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	1	0	0	1	0

- Note that it is possible to generate genotypes that have:
  - **Multiple phenotypes**
  - **Uncertain phenontypes**
  - **Invalid phenontypes**
- For example:



0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	1	0	0	**1**	**1**

- Which could be interpreted in different ways as follows:
  - **Multiple phenotypes** rule is certain and applies to two cases: don't play (bit 0 is 0) when humidity is either high or normal.
  - **Uncertain phenotypes** rules is uncertain: humidity could be either high or normal and we don't know which (but don't play in either case)
  - **Invalid phenotype** rule is invalid: humidity can't be high or normal at the same time.
- Different strategies exist for dealing with the situation of **invalid phenontype** , such as:
  - Preventing production of invalid phenotypes
  - Reporting lowest fitness for invalid phenotypes.

Genetic Algorithms: Fitness

Fitness is the **metric of success**, it determines which candidates are potential parts of solution.

- Like a **scoring** mechanism in data mining - a way of measuring how well a rule (a algorithm) performs.
- Fitness is measured of **each candidate** solution in a population - if you have 100 members of the population, you will have to evaluate fitness value of all 100 entries.
- Examples:
  - Win rate in a game playing algorithm.
  - Speed for a robot walking algorithm.
- **Fitness is domain dependent**

Genetic Algorithms: Selection

- Classic method of selection is probabilistic:

$$Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{N} Fitness(h_j)}$$

  where

  - N is the size of the population
  - Each h is the representation of a candidate solution
  - $Fitness(h)$ is the fitness (score) of that candidate.
- This method is called **fitness proportionate selection**
- Also called **roulette wheel selection**
- **Fitness proportionate selection is domain independent.**

There are other methods of selecting like:

- **Tournament selection**:
  - Randomly select two candidates
  - Keep the candidate with the higher fitness, with probability p proportional to each candidate's fitness (the more fit candidate is more likely to survive)
- **Rank selection**:
  - Sort candidates and rank them according to their fitness
  - Keep candidates with rank highest than (or equal to) proportion p.

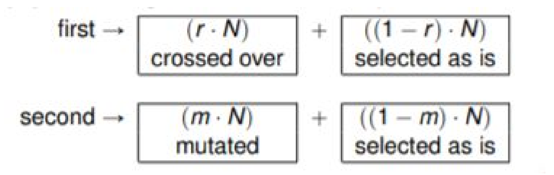
Tournament and Rank selection techniques are domain independent

Genetic Algorithms: Selection (cont)

- In the classic algorithm, a fixed **fitness threshold** is defined
- **Fixed threshold value is domain independent**
- We segment the population of N candidates into two groups:
  - P_{keep} = candidates whose fitness is $\geq$ threshold
  - $P_{replace}$ = candidates whose fitness is ¡ threshold
- Probabilistic selection is applied to P_{keep}:
  - Fitness proportionate: randomly select any candidates from P_{keep}
  - Rank: select top p from P_{keep}
  - Tournament: randomly select two candidates to compare from P_{keep}
- The proportion is known as the **exploration-exploitation** ration:
  - P_{keep} = exploitation
  - $P_{replace}$ = exploration

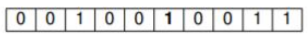
Genetic Algorithms: Reproduction

- There are two main **reproduction operators**:
  - **Crossover**
  - **Mutation**
- In classic algorithm, where population size = N.
  - **Crossover rate** = r, where 0 *leq* r 1
  - **Mutation rate** = m, where 0 *leq* m 1
- New population is generated with two steps:

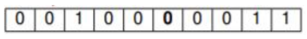


Genetic Algorithms: Reproduction – Mutation

- 1-*point* mutation
  - Randomly selected one bit in one string:

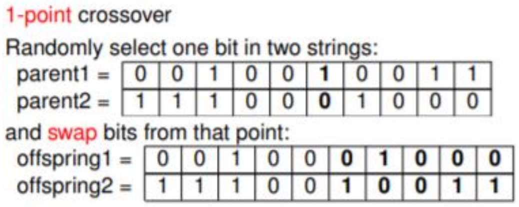


  - and **flip** it:



- *n-point* mutation: select n points ($n \leq$ length of the string)
- Called *asexual reproduction*, because there is only one parent.

Genetic Algorithms: Reproduction – Crossover


1-point crossover

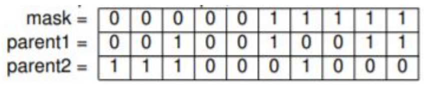
Randomly select one bit in two strings:

parent1 =	0	0	1	0	0	**1**	0	0	1	1
parent2 =	1	1	1	0	0	**0**	1	0	0	0

and swap bits from that point:

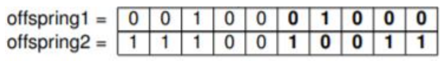
offspring1 =	0	0	1	0	0	**0**	**1**	**0**	**0**	**0**
offspring2 =	1	1	1	0	0	**1**	**0**	**0**	**1**	**1**

- *n-point* crossover: select n points ($n \leq$ length of the string, though higher values of n aren't very useful, too much variation or not enough, as n approaches length of string)
- Called *sexual reproduction*, because there are two parents.
- A common way to specify crossover points is to use **crossover mask** which indicates which bits should come from which parent
- So in our previous example, we would have:



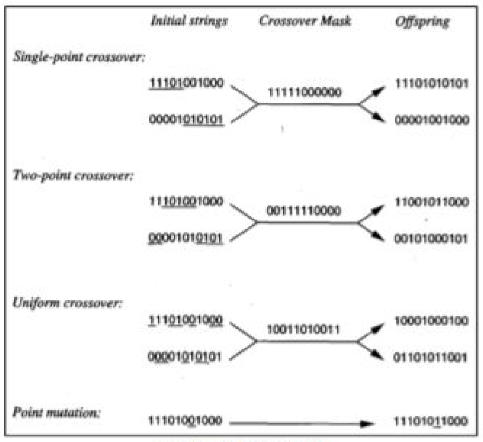
mask =	0	0	0	0	0	1	1	1	1	1
parent1 =	0	0	1	0	0	1	0	0	1	1
parent2 =	1	1	1	0	0	0	1	0	0	0

- Produces same result as on previous: slide



offspring1 =	0	0	1	0	0	**0**	**1**	**0**	**0**	**0**
offspring2 =	1	1	1	0	0	**1**	**0**	**0**	**1**	**1**

- Example



[Mitchell Table 9.2]

- **Uniform** crossover
  - Create mask by randomly select bits from each parent, using a **uniform** distribution
  - So it is equally likely to draw any bit from either parent.
- Other distributions can be used, for example if you want to weight certain bits differently from others.

Genetic Algorithms: Reproduction – Trade-offs

- Classic **trade-offs** when designing genetic algorithms are between the **population size** and the **number of generations** (iterations)
- A smaller population generally implies more generations are required, but each generation is evaluated relatively quickly.
- A larger population can mean that fewer generations are required, but each generation can take longer to be evaluated.
- Another classic **trade-off** is the **exploration-exploitation** ratio
- A higher **exploitation** ratio means that the set of candidates moves around the solution space (also called landscape) more slowly, more smoothly(without jumping around too much from one generation to the next) - solutions in a narrower search space are considered - search is conducted at a finer resolution
- A higher **exploration** ration means that the set of candidates jumps around the solution space - solutions at more disparate points in the search space are considered - search is conduced at a coarser resolution.
- **Nature** vs **Nurture**
- In biology, there is an old debate about whether **learned** behaviours are passed from parent to child in their genetic makeup.
- **Current biological evidence disavows this view**
- But computational methods have shown that GAs and GPs which learn and pass their learned information to their offspring can exhibit improved results over GA/GP parents that don't adapt.
- One example is the Baldwin effect:
  - If the environment is changing, then individuals that can adapt to the changes will survive longer
  - Individuals that learn more are compromised, initially, of incomplete genetic code, which fills in as they learn.
- Note that the inherent adaptivity in an evolutionary algorithm is in the changing population - the change membership - rather than changes in individual members.

Genetic Algorithms: Reproduction – Variations

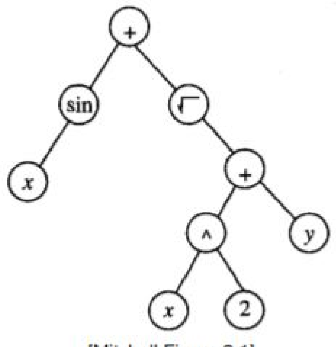
- **Real-valued GA**
- Instead of bit string, string contains real numbers
- Each number is an actual feature
- i.e., the value of the trait
- Instead of representing the presence or absence of a trait
- **Mutation** randomly changes value (because there's no notion of "flipping")
- Everything else works as with bit-string representations

Genetic Programming

Key difference between Genetic algorithm and Genetic programming is the representation.

- Representation is a **LISP s-expression** - an actual program which can be drawn as a tree structure.

Genetic Programming: Representation



[Mitchell Figure 9.1]

- S-Expression: = (+ (sin x) (sqrt (+ (pow * 2) y)))
- Equation: sin x = $\sqrt{x^2 + y}$

Genetic Programming: Fitness

- Fitness is determined by running the program:

$$fitness = sinx + \sqrt{x^2 + y}$$

- And then fitness (i.e., value returned by program) can be used directly:

$$if(fitness \geq fitness_{threshold})$$

  to determine if candidate program belongs to

  - P_{keep}
  - $P_{replace}$

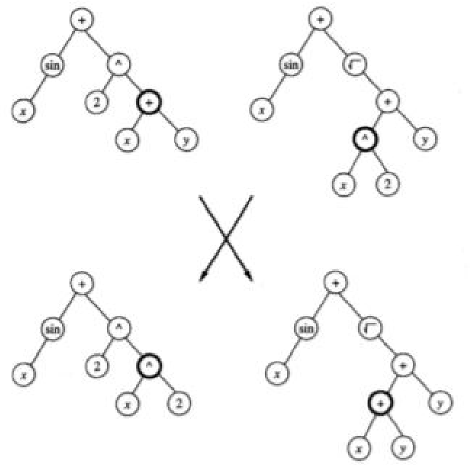
Genetic Programming: Selection

- Selection is performed in the same way as with Genetic Algorithms

Genetic Programming: Reproduction

- Genetic programming does not use **mutation** instead it uses **crossover**.
- With **crossover**, one (or more) **nodes** in the tree are selected from two parents.
- And then the sub-trees starting with those nodes are swapped.

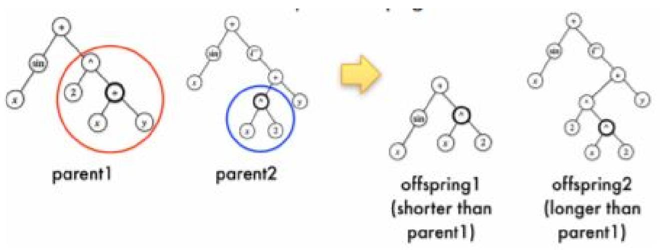
Genetic Programming: Reproduction – Crossover



- parent1: $\sin x + 2^{x+y}$
- parent2: $\sin x + \sqrt{x^2 + y}$
- offspring1: $\sin x + 2^{x^2}$
- offspring2: $\sin x + \sqrt{(x + y) + y}$

Genetic Programming: Reproduction – Code Bloat

- One significant issue with GP is **code bloat** –when the size of evolving programs grow extremely large (the length of the s-expression)
- Consider the modified example:

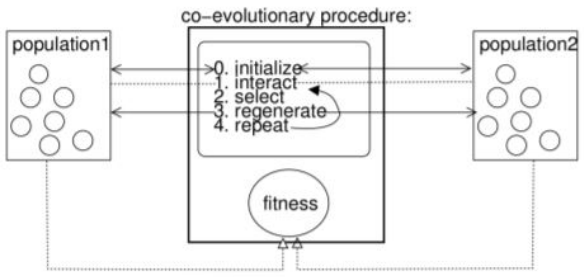

parent1
parent2
offspring1 (shorter than parent1)
offspring2 (longer than parent1)

- One approach to code bloat is to limit the maximum length of an s-expression that results from crossover.

1 Co-evolution

- With traditional **evolutionary algorithms**, we have one evolving population.
- The success of candidates solutions in the population is measured against a **fixed fitness function** - a fitness threshold.
- with co-evolution, there are **two evolving populations**
- The fitness of a candidate solution is measured by comparing members of opposing populations
- The result is an **arm's race**, where one population–as a whole–reaches to surpass the other population.

Co-evolutionary: Learning



- **Advantages**: There are distinct advantages with co-evolution where there is no easy way to define a fixed fitness function, or one simply doesn't exist.
- **Required**: There still has to be a way to compare two solutions and decide between them which is better.
- **Usage**: *Game Playing* is a classic example, where two candidate solutions play games against each other and the candidate that wins more games is declared the fittest.

Co-evolutionary: Components

Basic components in co-evolution

- **Representation**: could be a GA or a GP
- **Fitness**: members of opposing populations are compared against each other.
- **Selection**: the same as evolution
- **Reproduction**: same as evolution, but parents are chosen within the same population, (no cross-population sex)

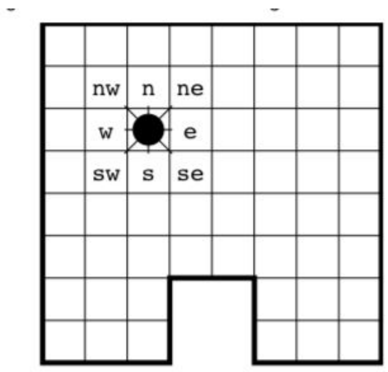
Issues in co-evolution

- Collusion
- When members of a population "live and let live", effectively: nobody tries to win. so each keeps getting better, but nobody becomes the best.
- **Mediocre Stable State** (MSS) also called **suboptimal equlibria** where the population settle into a portion of the landscape and do not change (convergence) - but this might not be the optimal portion off the landscape.

Examples

Wall-following robot

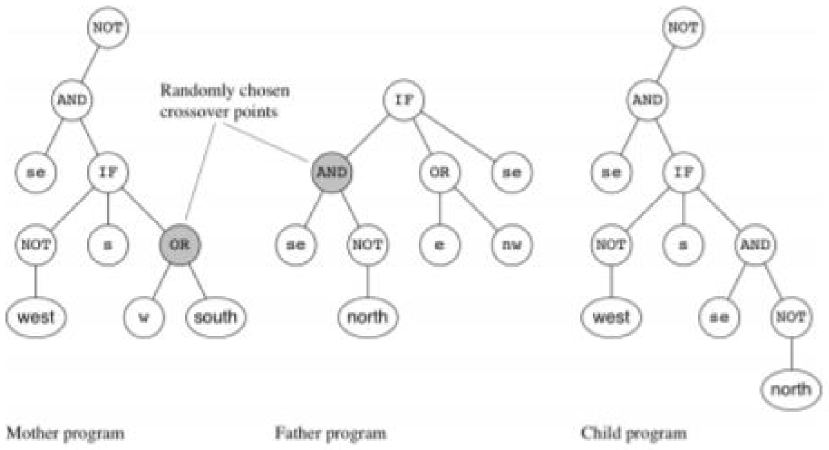
- Using GP to evolve a wall-following robot.



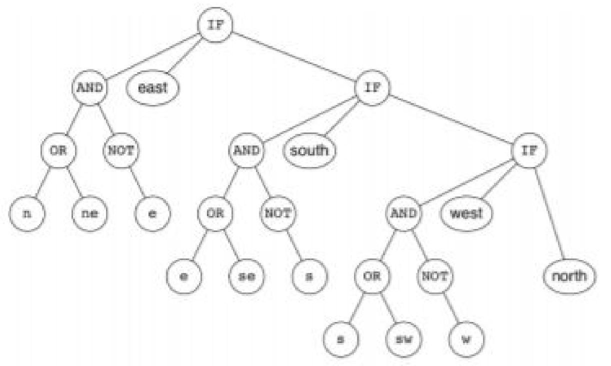
- Build the program up from four primitive functions:
  - AND(x, y) = 0 if x = 0; else y
  - OR(x, y) = 1 if x = 1; else y
  - NOT(x) = 0 if x = 1; else 1
  - IF(x, y, z) = y if x = 1; else z
- and four actions (moving by one cell):
  - North
  - East
  - South
  - West
- We must ensure that all expressions and sub-expressions have values for all possible arguments,m or terminate the program
- This makes sure 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...
- For example: it may divide by zero, or generate a negative number where only a positive number makes sense
- So we always need to have some kind of error handling to deal with the output of individual programs.

Wall-following robot: Reproduction

- Three basic steps
  - Evaluate the fitness function
  - Select the most fit
  - Breed the most fit
- How do we breed 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 shows that the GP-format is somewhat clumsy
- However, as we shall see, this program is relatively compact when compared with the programs that will be generated by GP.



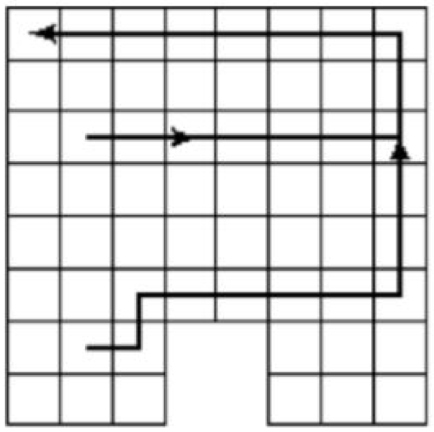
```
(IF (AND (OR (n) (ne)) (NOT (e)))
    (east)
    (IF (AND (OR (e) (se)) (NOT (s)))
        (south)
        (IF (AND (OR (s) (sw)) (NOT (w)))
        (west)
        (north))))
```

Wall-following robot: Learning to follow walls

1. Stat with 500 random programs
2. Fitness is evaluated by running each program on the task
3. Run the program 60 times and count the number of cells next to the wall visited
4. Worst possible program gets 0
5. Best possible program gets 320
6. Do 10 runs from random start points
7. Total count is fitness
8. Then we need to breed
9. take 500 programs and added them to the next generation as follows
10. Choose them by **tournament selection**:
    - Pick 7 at random
    - Add the most fit to the next generation.
11. Then create 4500 children into the next generation - parents chosen by tournament selection.
12. Mutate by replacing a randomly chosen sub-tree with a randomly generated sub-tree.

Wall-following robot: Generation 0

- The most fit member of the randomly generated initial programs has a fitness of 92, and has the kind of behaviour shown below.
- The program itself is given in the next slide.

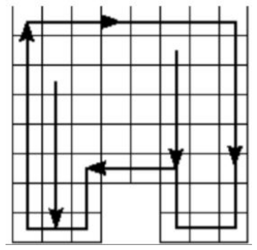


Wall-following robot: Generation 0 (cont)

```
(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)))
```

Wall-following robot: Generation 10

- The most fit member of generation 10 has fitness of close to the maximum 320
- The program follows wall perfectly, heading south until it reaches the boundary.

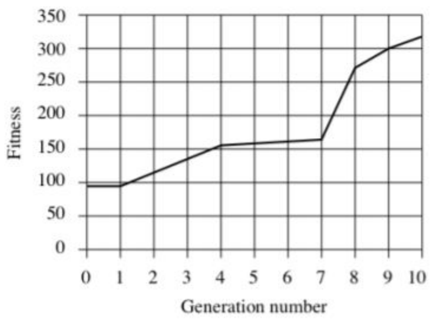


- The best program from generation 10

```
(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)))))))
```

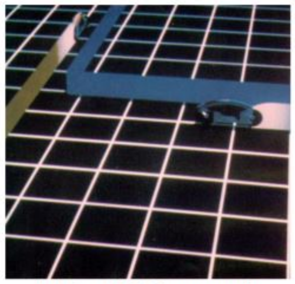
Wall-following robot: Results of learning

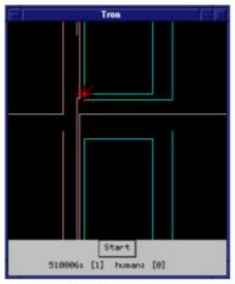
- This graph shows how the fitness of individuals grows quite sharply over the ten generations.



Tron internet experiment: History of learning to play games

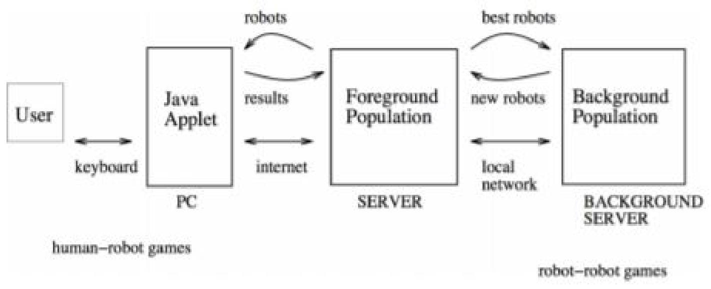
- Learning from humans
  - Checkers
  - too many games are needed
  - Humans are noisy
  - Humans are learning
- Learning from computers
  - Backgammon
  - Lack of generalisation
  - Deceptive landscape
  - Premature convergence
- The idea takes advantage of the internet
- Population of users collectively support a **fitness function**
- Population of robots (agents) collectively embody an **intelligent opponent**
- The first instance of human-agent co-evolution over the internet
- Can claim to be the first example of crowd-sourcing of research data.


(a) the 1982 Disney movie



(b) the game

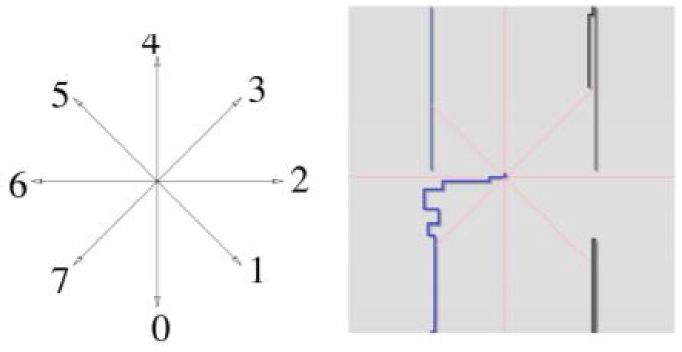
System Architecture: overview



Tron representation

- Each robot (agent) is a Genetic Program (GP)
- A LISP-like **s-expression**
- But code2d in Java (foreground)
- Or C (background)

Tron sensors



Tron GP representation

- GP operators: +, -, *, %, IFLTE
- Robot actions: Left, Right, Straight
- GP parameters: Real numbers
- Plus robot sensors: $s_0, s_1, ..., s_7$
- Maximum depth of tree: 17
- Maximum length of s-expression: 512 symbols.

Exploration vs exploitation

- Foreground population:
  - Agents vs humans
  - 100 members
  - 10 ”newbies” x 10 games (from background)
  - 18% exploration
  - 90 ”veterans” * 5 games
  - 82% exploration
- Background population:
  - Agents vs agents
  - 1000 members
  - 25 members training set
  - 15 best from foreground
  - 10 best from background
  - 1000 * 25 games
  - Best half mate and generate a new bottom half.

Fitness: Foreground population

- Fitness function -for foreground agents vs humans: where

$$F(a) = \sum_{h:p(h,a)>0} \left(\frac{l(h,a)}{p(h,a)} - \frac{l(h)}{p(h)}\right)\left(1 - e^{\frac{-p(h)}{10}}\right)$$

  - $l(h,a)$ = number of games lost by each human h against agent a.
  - $p(h,a)$ = number of games between human and agent.
  - $l(h)$ = total numbers of games lost by human h.
  - $p(h)$ = total number of games played by human h.
  - Exponential is confidence measure: decrease scores of humans who haven’t played many games.
- Uses **fitness sharing**: increasing reward for doing better than average and decreasing reward for doing worse than average.

Fitness: Background population

- Fitness function for background agents to decide which agents go to foreground:

$$F_T(a) = \sum_{a' \in T : pt(a,a') > 0} \frac{pt(a,a')}{l(a')}$$

  where

  - T = Training set
  - $pt(a, a') = \{0,\ 1,\ 0.5\}$ if a {loses, wins, draws } to a'
  - $l(a')$ = number of games lost by a'

  which rewards more for winning against better players.

- Fitness function for background agents to replenish training set:

$$F_{T,T'}(a) = \sum_{a' \in T} \frac{pt(a,a')}{(1 + \sum(pt(a'',a') : a'' \in T'))}$$

  where

  - T = (old) training set
  - T' = new training set (initialised to empty set)
  - $pt(a, a') = \{0,\ 1,\ 0.5\}$ if a {loses, wins, draws } to a'

- De-values winning against agents that are already in the training set ($a'' \epsilon\ T'$

Reproduction: The next generation

In the foreground:

- 15 members go from foreground to background (equation 1)
- 10 members go from background to foreground (equation 2)

In the background:

- All 1000 members play against 25-member training set
- Rank selection: bottom half replaced with **random crossover** of players in top half.
- Training set is replenished using fitness sharing (equation 3)

Example

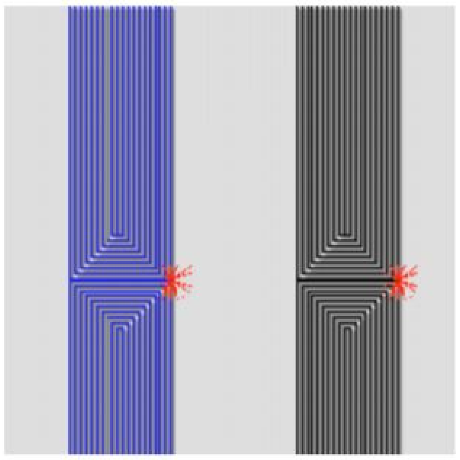
- S-expression

```
(* s7 (IFLTE s0 0.92063 s7 (- (% s3 (- (+ 0.92063 (
    IFLTE 0.92063 s5 0.92063 (LEFT_TURN))) (IFLTE (-
    (IFLTE s2 s6 (RIGHT_TURN) (LEFT_TURN)) (IFLTE (+(
    LEFT_TURN) (LEFT_TURN)) s6 s5 s6)) s7 (RIGHT_TURN
    ) s6))) (RIGHT_TURN))))
```

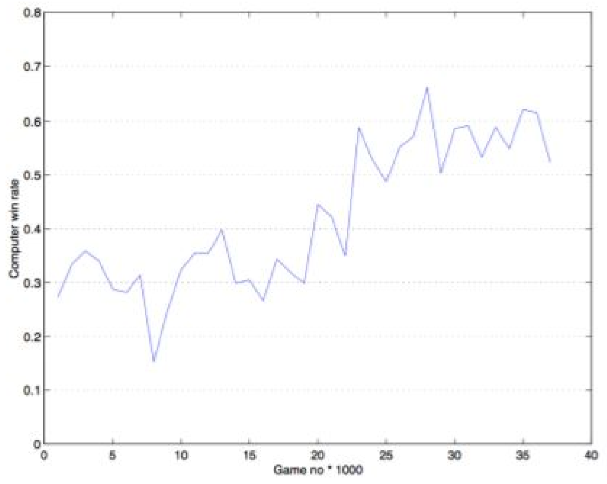
- Rough translation into pseudo-code:

```
if ( FRONT < 0.92063 ):
  go straight
elif ( 0.92063 >= REAR_RIGHT ):
  turn left
elif ( LEFT < RIGHT ):
  turn right
else:
  turn left
```

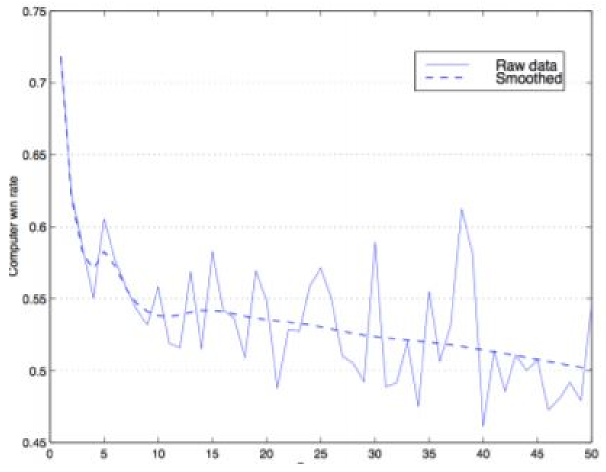
Problems: collusion



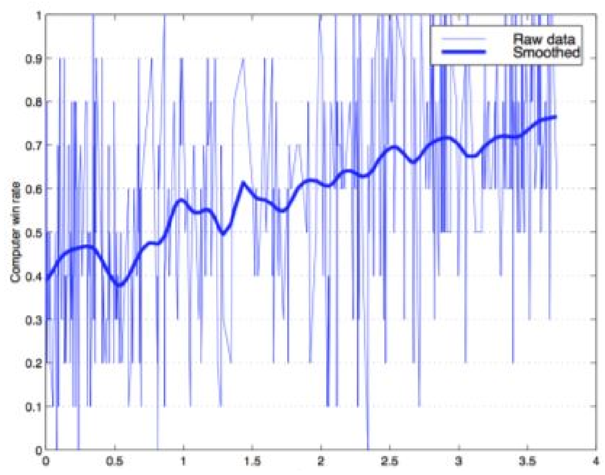
Results: Computer win rate - agents get better



Results: Loosing to human over humans first 50 games



Results: Learning - agents beating humans over their first 10 games


Computer win rate
Raw data
Smoothed