Probabilistic Incremental Program Evolution (PIPE)
Population Based Incremental Learning [Baluja, 94]

- It learns explicitly a probabilistic model of the interesting regions of the search space.
- If points in the search space are bitstrings (like “0101110”), the probabilistic model to be learned is:
  - a vector $\mathbf{p}=(p_1, p_2, \ldots, p_n)$
  - $p_i$ is the probability of generating a 1 in the $i$ position of the bitstring $x=(x_1, x_2, \ldots, x_n)$
  - Initially $\mathbf{p}=(0.5, 0.5, \ldots, 0.5)$
Population Based Incremental Learning [Baluja, 94]

- A population of $x$, is generated, their fitness is computed, and $\rho$ is updated by using the $M$ best individuals.

- Update rule: $\rho'_i = \rho_i(1-LR) + LR \cdot x_i^*$

- Eventually, $\rho$ should converge to a solution like:
  - $\rho=(0.99, 0.001, \ldots, 0.99)$
Population Based Incremental Learning [Baluja, 94]

Probability model

\[(0.1, 0.9, 0.5, 0.8)\]

Population

\[
p_i' = p_i \cdot (1-LR) + LR \cdot x_i^*
\]
Estimation of Distribution Algorithms (EDA’s)

1. Generate an initial population and evaluate them
2. Select $M$ best individuals
3. Estimate the probability distribution
4. Generate a new population (sometimes, the old population is mixed with the new one)
5. If not-termination, go to 2
State of the Art for EDA’s applied to GP

Probabilistic Incremental Program Evolution (PIPE)

- EDA’s applied to the evolution of parse trees (hierarchical programs)
- PIPE [Salustowicz, Schmidhuber, 97]
- Search in the space of tree-shaped probability distributions
- Tries to find a distribution that generates good programs (trees)
Probabilistic Incremental Program Evolution (PIPE)

- Programs made of:
  - Functions: \( F = \{F_1, \ldots, F_k\} \)
  - Terminals: \( T = \{T_1, \ldots, T_l\} \)

- The Generic Random Constant (GRC):
  - Similar to Ephemeral Random Constant (ERC)

- *Closure* required
Probabilistic Prototype Tree (PPT)
An Initial Node of the PPT

Initially:

\[ P_j(I) := \frac{P_T}{l}, \forall I : I \in T \]
\[ P_j(I) := \frac{1 - P_T}{k}, \forall I : I \in F, \]

| \( P_j(x) \) | 0.3 |
| \( P_j(R) \) | 0.3 |
| \( P_j(\) \) | 0.05 |
| \( P_j(+) \) | 0.05 |
| \( P_j(-) \) | 0.05 |
| \( P_j(*) \) | 0.05 |
| \( P_j(\%) \) | 0.05 |
| \( P_j(sin) \) | 0.05 |
| \( P_j(cos) \) | 0.05 |
| \( P_j(exp) \) | 0.05 |
| \( P_j(rlog) \) | 0.05 |

\[ R_j = 0.45 \]

\[ N_j \]
Creation of the Initial Population

- The PPT is parsed top-down, left-to-right.
- A function or terminal is selected according to its probability.
- If R is selected (the GRC), then:
  - If prob(R) > threshold, Then fixed value R
  - Else, generate a random value for R
Size of the PPT

- Empirically, it is enough with PPT three times the best solution found so far
- Initially, the PPT contains only the root node
- Nodes are created on demand (if in a leave node, a function is selected, it is necessary to create the arguments)
PPT Growth
Pruning the PPT

- In case a symbol has a very large probability

```
Variable x
```

```
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0.97</td>
</tr>
<tr>
<td>R</td>
<td>0.008</td>
</tr>
<tr>
<td>(</td>
<td>0.001</td>
</tr>
<tr>
<td>)</td>
<td>0.003</td>
</tr>
<tr>
<td>*</td>
<td>0.006</td>
</tr>
<tr>
<td>%</td>
<td>0.002</td>
</tr>
<tr>
<td>sin</td>
<td>0.003</td>
</tr>
<tr>
<td>cos</td>
<td>0.001</td>
</tr>
<tr>
<td>exp</td>
<td>0.004</td>
</tr>
<tr>
<td>rlog</td>
<td>0.002</td>
</tr>
<tr>
<td>R</td>
<td>0.23</td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0.001</td>
</tr>
<tr>
<td>R</td>
<td>0.002</td>
</tr>
<tr>
<td>(</td>
<td>0.004</td>
</tr>
<tr>
<td>)</td>
<td>0.002</td>
</tr>
<tr>
<td>*</td>
<td>0.005</td>
</tr>
<tr>
<td>%</td>
<td>0.001</td>
</tr>
<tr>
<td>sin</td>
<td>0.95</td>
</tr>
<tr>
<td>cos</td>
<td>0.002</td>
</tr>
<tr>
<td>exp</td>
<td>0.03</td>
</tr>
<tr>
<td>rlog</td>
<td>0.003</td>
</tr>
<tr>
<td>R</td>
<td>0.77</td>
</tr>
</tbody>
</table>
```
Probabilistic Incremental Program Evolution (PIPE)

- Every generation, the PPT is updated towards the best individual (just one), so that it becomes more likely to generate similar individuals.
- Minimization
- If fitness are equal, the smaller solution is preferred.
PIPE’s Algorithm

- Two modes of learning:
  - *Generation based learning (GBL)*: Updates the PPT towards the best individual in that generation. Increases the probability that similar individuals will be generated by the PPT.
  - *Elitist learning (EL)*: Updates the PPT towards the best program *found so far* (it’s a kind of long term memory).
PIPE’s Algorithm

1. Initial Population
2. Fitness computation
3. PPT update: with probability $P_{el}$
   1. Generation Based Learning
   2. Elitist Learning
4. PPT mutation (exploration)
5. PPT pruning
1. Let \( \text{PROG}_b \) be the target program, used to update the PPT.

2. Let \( P(\text{PROG}_b) \) be the probability that the PPT currently generates \( \text{PROG}_b \).

\[
P(\text{PROG}_b) = \prod_{j : N_j \text{ used to generate } \text{PROG}_b} P_j(I_j(\text{PROG}_b))
\]
3. The desired (target) probability for PROG_b is computed

\[ P_{TARGET} = P(\text{PROG}_b) + (1 - P(\text{PROG}_b)) \cdot lr \cdot \frac{\varepsilon + \text{FIT}(\text{PROG}_b)}{\varepsilon + \text{FIT}(\text{PROG}_b)} \]

- \( lr \) = learning rate
- The quotient controls fitness-dependent learning (minimization)
- If large epsilon, learning is independent of the fitness (quotient = 1)
4. PPT is modified until \( P(\text{PROG}_b) = P_{\text{TARGET}} \)

5. Probabilities are updated in parallel.

6. \( c^{lr} (0.1) \): tradeoff between good accuracy and fast update

\[
\text{REPEAT UNTIL } P(\text{PROG}_b) \geq P_{\text{TARGET}} : \\
P_j(I_j(\text{PROG}_b)) := P_j(I_j(\text{PROG}_b)) + c^{lr} \cdot lr \cdot (1 - P_j(I_j(\text{PROG}_b)))
\]
5. Normalization of PPT (instructions not in PROG\_b, get decreased proportionally to their current value. Everything must add to 1.0)

\[
P_j(I) := P_j(I) \cdot \left(1 - \frac{\sum_{I^* \in S} P_j(I^*)}{\sum_{I^* \in S} P_j(I^*) - \sum_{I^* \in S} P_j(I^*)} \right) \quad \forall P_j(I) : I \neq I_j(P\text{ROG}_b)
\]
Finally, R constants are copied from \( \text{PROG}_b \) to PPT.
Mutating the PPT

- To explore around \( \text{PROG}_b \)
- \( \text{P}(\text{I}(\text{PROG}_b)) \) of instructions in \( \text{PROG}_b \) get mutated
- \( P_M = \) mutation probability per program
- \( P_{M_p} = \) mutation probability per node and instruction
- \( z = \) number of possible instructions
- Dividing by \( |\text{PROG}_b| \) avoids larger programs having more mutations (the square root gives more mutations to large programs. Empirical reasons)

\[
P_{M_p} = \frac{P_M}{z \cdot \sqrt{|\text{PROG}_b|}}
\]
Mutating the PPT

- **Mutation:**

\[ P_j(I) := P_j(I) + mr \cdot (1 - P_j(I)) \]

- **Normalization:**

\[ P_j(I) := \frac{P_j(I)}{\sum_{I^* \in S} P_j(I^*)} \quad \forall P_j(I) : I \in S \]
Results PIPE

- Symbolic regression: PIPE better than GP in 24% of runs, and worse in 33%. Larger variance.

- 6-bit parity problem:
  - More successful runs (70% vs. 60%)
  - Faster (52476 vs. 120000 evaluations)
  - Smaller (61 vs. 90 nodes)

PIPE Applied to Soccer

- Domain similar to Robosoccer
- Actions:
  - Simple: go_forward, turn_to_ball, turn_to_goal, shoot
  - Complex: goto_ball, goto_goal, goto_own_goal, goto_player, goto_opponent, pass_to_player, shoot_to_goal
PIPE Applied to Soccer

- **BRO**: Biased Random Opponent. Biased random player (scores 75 goals to an static opponent)
- **GO**: Good Opponent. Scores 417 goals to BRO
- **PIPE**: fitness computed by playing against BRO
- **COPIPE**: co-evolution fitness
- **TD-Q**: Reinforcement learning
- A PPT is learned for every action
## Results PIPE vs. BRO (simple actions)

<table>
<thead>
<tr>
<th>Team size</th>
<th>GO</th>
<th>PIPE</th>
<th>CO-PIPE</th>
<th>TD-Q</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average goals ± st.d.</td>
<td>417 ± 6</td>
<td>320 ± 42</td>
<td>212 ± 97</td>
</tr>
<tr>
<td></td>
<td>Average BRO goals ± st.d.</td>
<td>0 ± 0</td>
<td>10 ± 7</td>
<td>20 ± 10</td>
</tr>
<tr>
<td></td>
<td>Achieved after games</td>
<td>n.a.</td>
<td>3300</td>
<td>3000</td>
</tr>
<tr>
<td>Maximum score difference</td>
<td>481</td>
<td>359</td>
<td>310</td>
<td>70</td>
</tr>
<tr>
<td>3</td>
<td>Average goals ± st.d.</td>
<td>481 ± 8</td>
<td>373 ± 86</td>
<td>324 ± 62</td>
</tr>
<tr>
<td></td>
<td>Average BRO goals ± st.d.</td>
<td>0 ± 1</td>
<td>14 ± 6</td>
<td>14 ± 11</td>
</tr>
<tr>
<td></td>
<td>Achieved after games</td>
<td>n.a.</td>
<td>3300</td>
<td>3200</td>
</tr>
<tr>
<td>Maximum score difference</td>
<td>364</td>
<td>481</td>
<td>357</td>
<td>154</td>
</tr>
<tr>
<td>11</td>
<td>Average goals ± st.d.</td>
<td>367 ± 18</td>
<td>512 ± 129</td>
<td>393 ± 53</td>
</tr>
<tr>
<td></td>
<td>Average BRO goals ± st.d.</td>
<td>3 ± 1</td>
<td>31 ± 23</td>
<td>36 ± 27</td>
</tr>
<tr>
<td></td>
<td>Achieved after games</td>
<td>n.a.</td>
<td>3100</td>
<td>1900</td>
</tr>
</tbody>
</table>
## Results PIPE vs. BRO

(complex actions)

<table>
<thead>
<tr>
<th></th>
<th>GO</th>
<th>PIPE</th>
<th>CO-PIPE</th>
<th>TD-Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum score difference</td>
<td>364</td>
<td>530</td>
<td>536</td>
<td>46</td>
</tr>
<tr>
<td>Average goals ± st.d.</td>
<td>367 ± 18</td>
<td>551 ± 215</td>
<td>539 ± 220</td>
<td>76 ± 140</td>
</tr>
<tr>
<td>Average BRO goals ± st.d.</td>
<td>3 ± 1</td>
<td>21 ± 35</td>
<td>3 ± 4</td>
<td>30 ± 29</td>
</tr>
<tr>
<td>Achieved after games</td>
<td>n.a.</td>
<td>1200</td>
<td>1200</td>
<td>900</td>
</tr>
</tbody>
</table>
PIPE Limitations

- Probability distribution in a node is independent of other nodes
Estimation of Distribution Programming (EDP)

- Joint probability of father and children nodes
EDP Algorithm

1. Create population
2. Eval individuals
3. Estimate distribution
4. If termination, go to 7
5. Generate a new population
6. Replace old population
7. Return best individual
Bayesian Network

\[ P(X_3 = x_3 | X_1 = x_1, X_2 = x_2, \cdots, X_6 = x_6, X_7 = x_7) \]
\[ = P(X_3 = x_3 | X_1 = x_1, X_2 = x_2). \]  \( (1) \)
Possible Bayesian Networks
Cost of Different Bayesian Networks

\[ n \cdot m^{i+1} \times \text{(memory size of int)} \]

- \( m = \text{num. possible symbols} \)
- \( n = \text{num. nodes in the tree} \)
- \( i = \text{num. dependencies} \)

<table>
<thead>
<tr>
<th>#</th>
<th>Graph topology</th>
<th>Memory size</th>
<th>Adequate sampling size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Only parent node</td>
<td>( n \cdot m^2 \times \text{INT} )</td>
<td>( m^2 )</td>
</tr>
<tr>
<td>2</td>
<td>Only parent node</td>
<td>( n \cdot m^3 \times \text{INT} )</td>
<td>( m^3 )</td>
</tr>
<tr>
<td>3</td>
<td>Parent and brother nodes</td>
<td>( n \cdot m^4 \times \text{INT} )</td>
<td>( m^4 )</td>
</tr>
<tr>
<td>4</td>
<td>Parent and brother nodes</td>
<td>( n \cdot m^5 \times \text{INT} )</td>
<td>( m^5 )</td>
</tr>
</tbody>
</table>
Computing Probabilities

\[ P(X_i = x | C_i = c) = \frac{\sum_{j=1}^{N} \delta(j, X_i = x | C_i = c)}{N} \]

where

\[ \delta(j, X_i = x | C_i = c) = \begin{cases} 
1 & \text{if } X_i = x \text{ and } C_i = c \\
0 & \text{else} 
\end{cases} \]

j: individual j
Fitness Weighted Probabilities

\[ P(X_i = x | C_i = c) = \frac{\sum_{j=1}^{N} F_j \delta(j, X_i = x | C_i = c)}{\sum_{j=1}^{N} F_j} \]

Unlike PIPE, the probability distribution is generated directly from the population, without considering the previous distribution.
Selected Individuals

Note: only father and brother nodes are considered

Ricardo Aler. ICML’06 Automatic Inductive Programming Tutorial
Distribution from the Selected Individuals

Table 2: Estimating $P(X_3 = x_3 | X_1 = x_1, X_2 = x_2)$

<table>
<thead>
<tr>
<th>$X_3$</th>
<th>$X_1$</th>
<th>$X_2$</th>
<th>Frequency</th>
<th>Probability ($= P$)</th>
<th>Modified probability ($= P'$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>0.5</td>
<td>1</td>
<td>0.5</td>
<td>0.49</td>
</tr>
<tr>
<td>*</td>
<td>+</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>0.02</td>
</tr>
<tr>
<td>0.5</td>
<td>+</td>
<td>0.5</td>
<td>1</td>
<td>0.5</td>
<td>0.49</td>
</tr>
<tr>
<td>+</td>
<td>*</td>
<td>+</td>
<td>1</td>
<td>0.25</td>
<td>0.26</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>+</td>
<td>1</td>
<td>0.25</td>
<td>0.26</td>
</tr>
<tr>
<td>0.5</td>
<td>*</td>
<td>+</td>
<td>2</td>
<td>0.5</td>
<td>0.48</td>
</tr>
<tr>
<td>+</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>0</td>
<td>0.03</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>0</td>
<td>0.03</td>
</tr>
<tr>
<td>0.5</td>
<td>*</td>
<td>*</td>
<td>2</td>
<td>1.0</td>
<td>0.94</td>
</tr>
</tbody>
</table>
Distribution Adjustment

\[ P' = (1 - \alpha)P(X_i = x | C_i = c) + \alpha P_{\text{default}}(X_i = x | C_i = c). \]

\[ P_{\text{default}}(X_i = x | C_i = c) = \frac{1}{\text{node symbol number}}. \]

It is some sort of a-priory probability. Thus, all probabilities become different than 0

(number of symbols)
Program Generation

- First, the best $k$ individuals are selected from the previous generation
Individual Generation
The “Max Problem”

Obtain the maximum value using +, *, y 0.5

Easy for EDP-GP because 0.5 must be at the bottom, + in the middle and * at the top

![Diagram showing a tree structure with nodes labeled 0.5, 2, 4, 16, 256, and 65536. The nodes are connected by * and + operators.]

Figure 6: The maximum value by a limited depth tree.
Results. Max Problem

![Graph showing average fitness values over generations for different GP configurations.](image)

Ricardo Aler. ICML'06 Automatic Inductive Programming Tutorial
Results. 6-Multiplexer

Here EDP works slightly worse
Recursive Bayesian Networks

Figure 10: The method used in experiments

Figure 11: Recursive single Bayesian network

Figure 12: Recursive multi-Bayesian network
Limitations. Building Blocks at Different Places

\[ f(x) = x^3 + x^2 + x. \]
Extended EDP (XEDP) [Yanai & Iba, 05]

- It is like EDP, but in addition, it uses a recursive (position independent) bayesian network
XEDP. Redes bayesianas

Dist. Recursiva (relative position)  Conditional Distribution (absolute position)

\[ P(Y_1, Y_2, \ldots, Y_{\text{max}} | Y, Y_p) \]

\[ P(\text{Children} | \text{father, grandfather}) \quad P(\text{children} | \text{father}) \]
XEDP. Generating Individuals

- It combines the two distributions (absolute and relative)
  1. Generate a program $T$ using the absolute distribution
  2. Generate a subtree $S$ using the relative distribution
  3. Replace a subtree of $T$ by $S$
XEDP. Experiments

- XEDP
- GP
- Type A: uses only the absolute distribution
- Type B: uses only the relative distribution
- Type C: XEDP, but replaces the subtree by a random one
- Type D: XEDP, but only the absolute distribution without father-son dependencies (like PIPE)
Results Max Problem

Figure 2: Cumulative probability of success for the Max problem.
Results 6-multiplexer

Figure 3: Cumulative probability of success for the Boolean 6-bit multiplexer problem.
Results Wall-following

Figure 4: Cumulative probability of success for the Wall-following problem.
Extended Compact Genetic Programming (ECGP)


- It uses Marginal Product Models

- Divides the PPT into several subtrees, considered independent

- Joint probabilities are computed for every subtree (no independence assumed within subtrees)
EDA’s with Grammars

- Learning Stochastic Context Free Grammars (SCFG)
- Each rewriting rule has a weight that indicates the probability to be used
GMPE Algorithm

- Randomly Generate Initial Population
- Evaluate and Select
- Learn SCFG model from Selected Individuals
- Sample SCFG Model to Obtain New Population
Results. Max Problem

<table>
<thead>
<tr>
<th>Method</th>
<th>No. Eval/Gen</th>
<th>Gen</th>
<th>No. Eval/Run</th>
<th>Succeed</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMPE</td>
<td>30</td>
<td>453</td>
<td>13,590</td>
<td>60%</td>
<td>7.4</td>
</tr>
<tr>
<td>GP</td>
<td>200</td>
<td>500</td>
<td>100,000</td>
<td>&lt; 60%</td>
<td></td>
</tr>
</tbody>
</table>
Conclusions EDA-GP

- Probability distributions are explored explicitly
- They can also learn stochastic context-free grammars
- Not well tested yet, but it seems that they are equivalent or better than GP