

Programación Automática

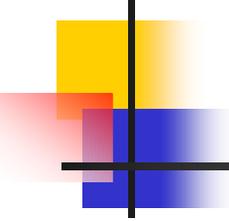
Ricardo Aler Mur

Universidad Carlos III de Madrid

<http://www.uc3m.es/uc3m/dpto/INF/aler>

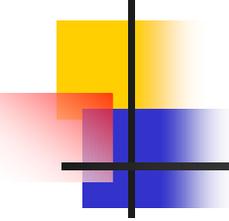
GRUPO **EVANNAI**

<http://et.evannai.inf.uc3m.es/>



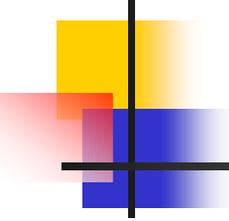
Motivación de la charla

- Muchos investigadores trabajando en el mismo objetivo (generación automática de programas de ordenador)
- Pero con técnicas diferentes (programación genética, ILP, etc.)
- Campo fragmentado
- Se intentará dar una visión unificada
- Algunas técnicas poco conocidas



Programación Automática

- Generación automática de programas de ordenador
- Diciendo **QUÉ** se quiere que haga y no el **CÓMO**



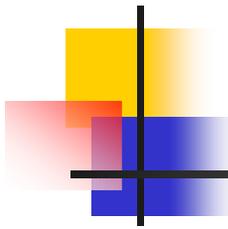
Ejemplo programación automática

- Entrada:

- $([2,1], [1,2]); ([2,3,1], [1,2,3]);$
- $([3,5,4], [3,4,5]); ([], []); \dots$

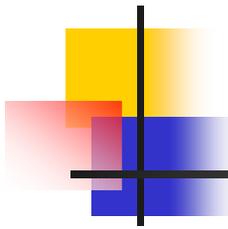
- Instrucciones primitivas:

- $(\text{dobl start end work}) (\text{wismaller } x \ y)$
- $(\text{swap } x \ y) (\text{wibigger } x \ y)$
- $(\text{e1+ } x) (\text{e- } x \ y) (\text{e1- } x)$



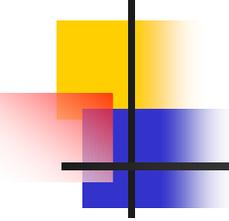
Programa obtenido

```
(dobl (wismaller (wismaller (el- *len*) *len*)
                 index)
      (dobl (wismaller index
                (wismaller
                 (el- index)
                 (el+ (el- index))))
            (el- *len*)
            (swap (swap (el- *len*) index)
                  index))
      (dobl (swap (wibigger index (e- index *len*))
                (e- index *len*))
            (el- *len*)
            (swap (wismaller (el+ index) index)
                  index))))
```



Equivalente a ...

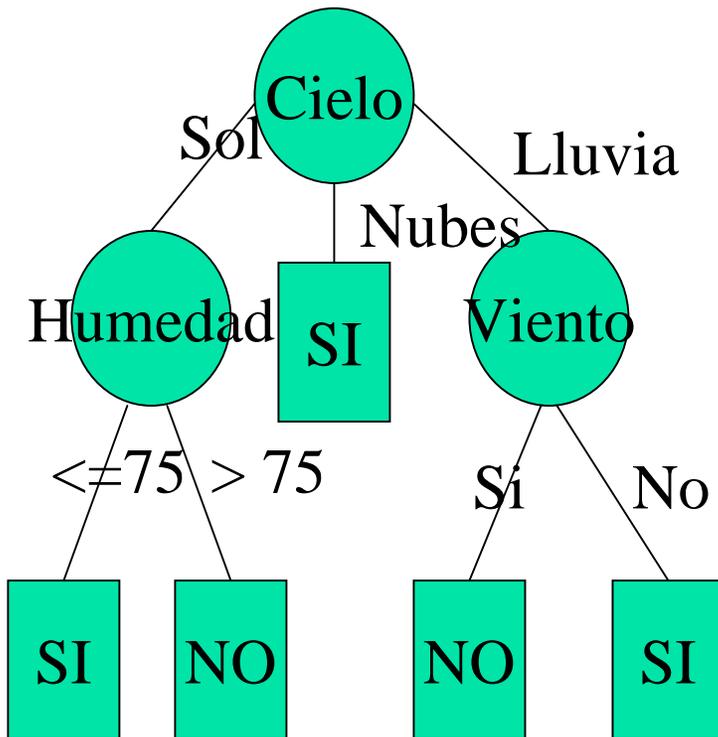
```
(dobl 0
  (el- *len*)
  (dobl 0
    (el- *len*)
    (swap (wismaller (el+ index) index)
           index))))
```



Motivación Programación Automática

- Es lo más **GENERAL** que un sistema de aprendizaje automático puede producir
- Ir más allá del Aprendizaje Automático (AA)
 - AA: aprendizaje de conceptos, "mapeo", lenguajes atributo-valor, tamaño de entrada fijo
 - PA: aprendizaje de programas, lenguajes Turing-completos (bucles, recursividad, variables, subrutinas, ...)

Aprendizaje Automático



IF Cielo = Sol

 Humedad \leq 75 **THEN** Tenis = Si

ELSE IF Cielo = Sol

 Humedad $>$ 75 **THEN** Tenis = No

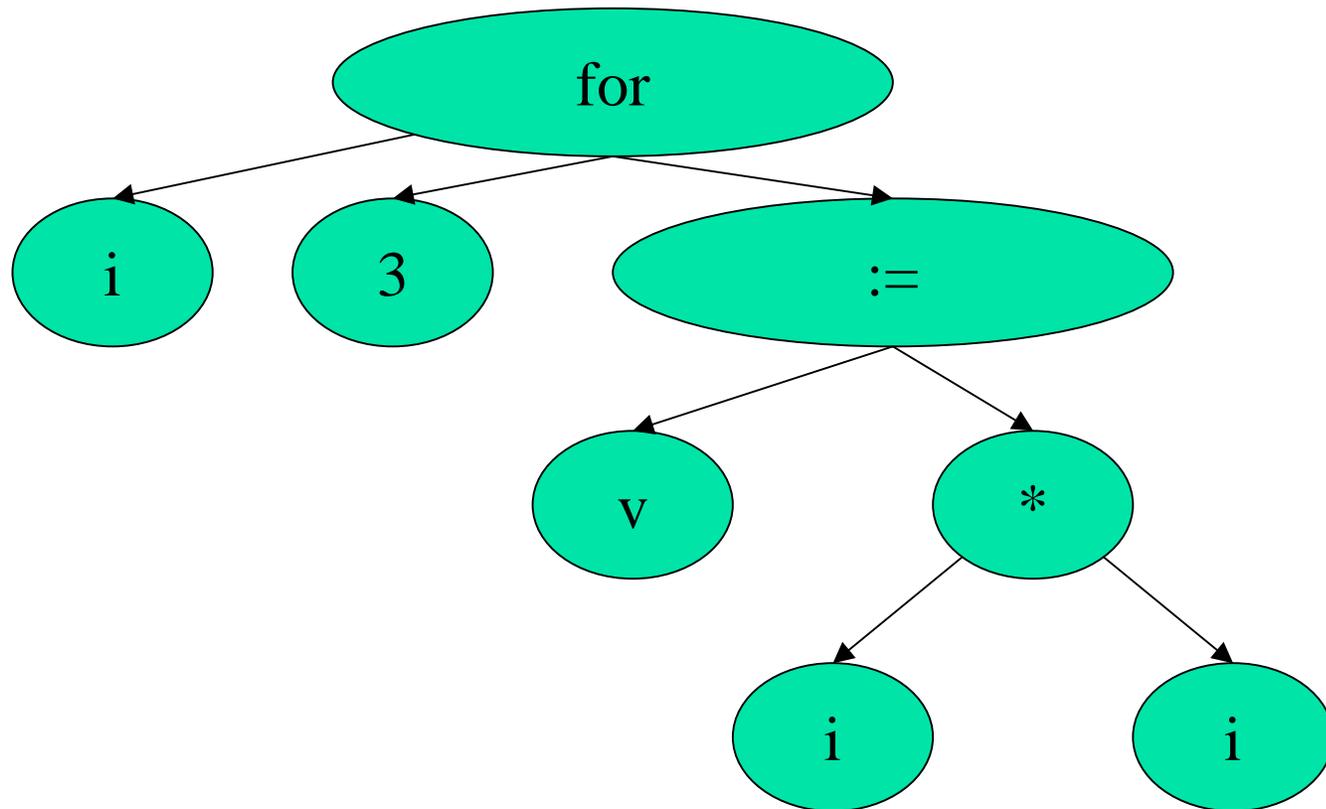
ELSE IF Cielo = Nubes **THEN** Tenis = Si

ELSE IF Cielo = Lluvia

 Viento = Si **THEN** Tenis = Si

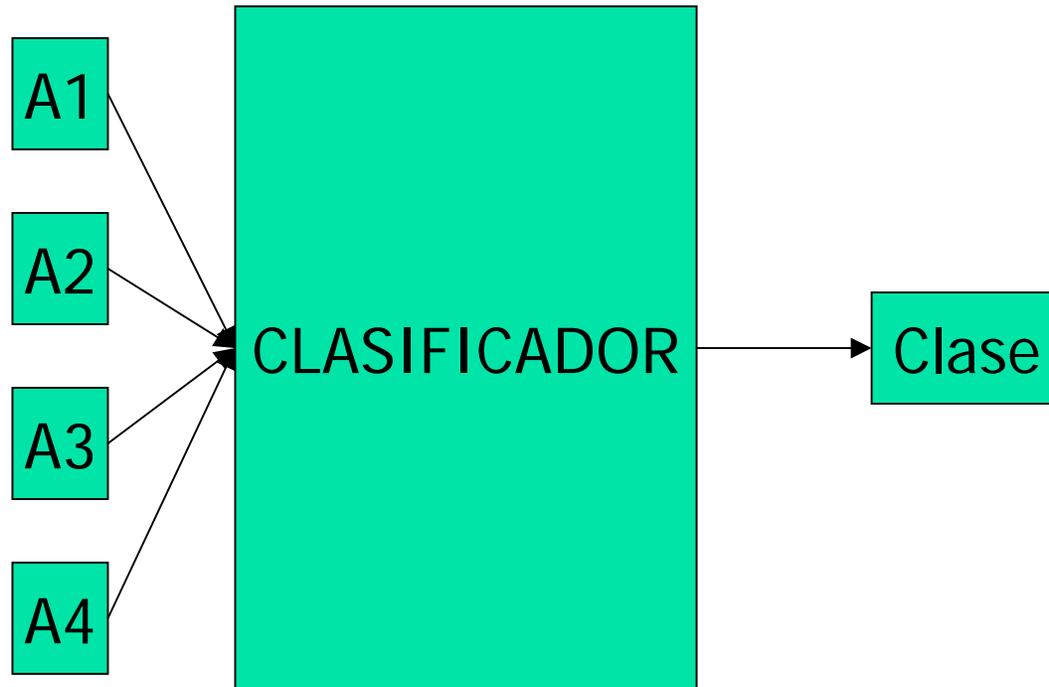
ELSE Tenis = No

Programación Automática

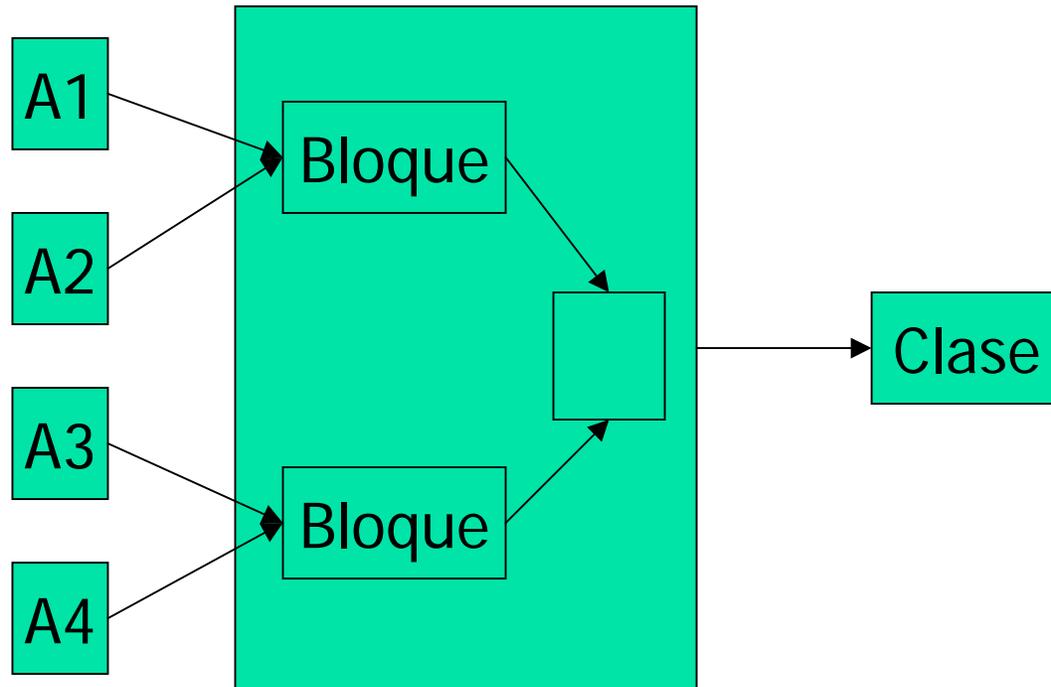


Problema de clasificación típico

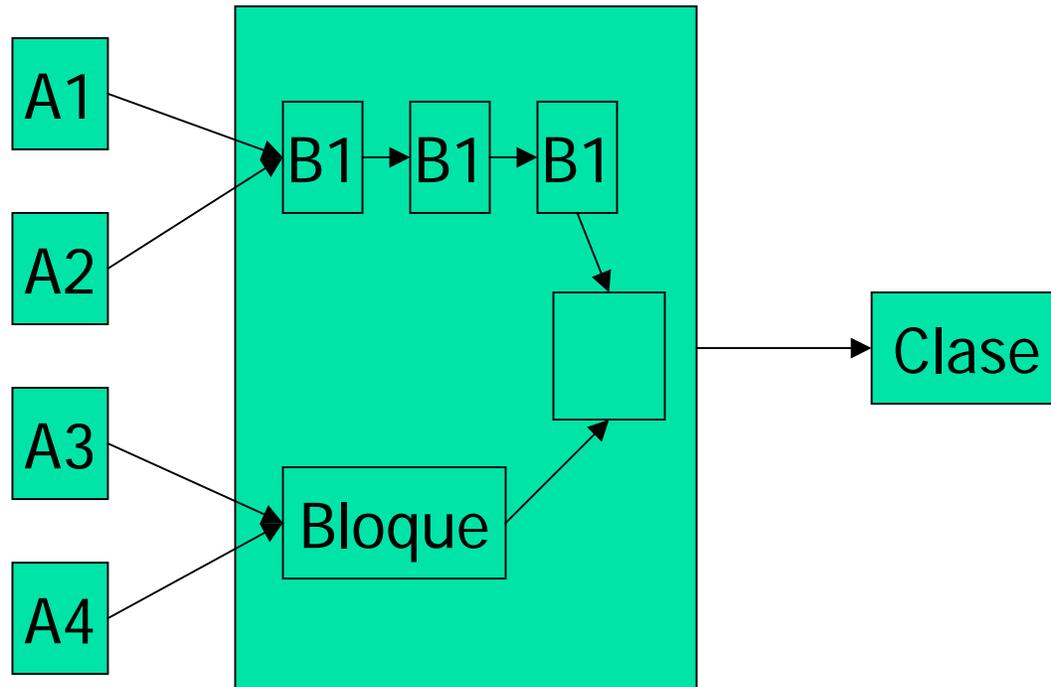
Tamaño de entrada fijo

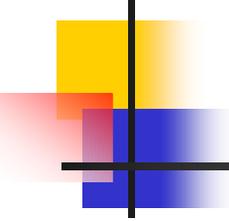


Subrutinas



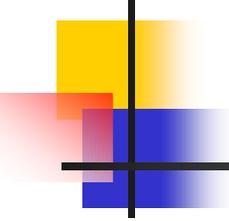
Bucles





Tamaño de entrada variable

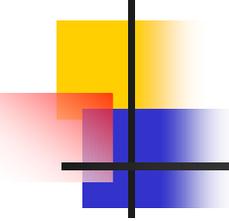
- Ordenar listas de números de cualquier tamaño:
 - $([2,1], [1,2])$
 - $([2,3,1], [1,2,3])$
 - $([3,5,4], [3,4,5])$
 - $([], [])$



Recursividad

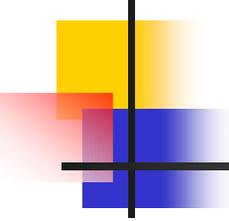
Problemas donde la solución natural es recursiva

$$n! = \begin{cases} n * (n - 1)! & \text{si } n > 0 \\ 1 & \text{si } n = 0 \end{cases}$$



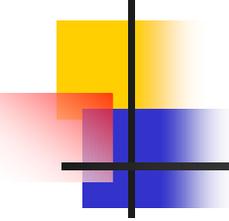
Extensiones al Aprendizaje Automático

- Uso de primitivas genéricas
- Tamaño variable a la entrada
- Uso de variables para almacenar resultados parciales
- Uso de subrutinas para reutilizar código
- Ir más allá del “mapping”: inclusión del tiempo, iteración, recursividad, bucles



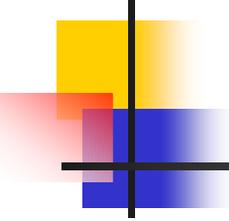
¿Para qué problemas?

- Computación cuántica
- Computación paralela
- Programación de agentes inteligentes (Robosoccer)
- En general, problemas difíciles de programar, fáciles de especificar, que requieran nuevos algoritmos



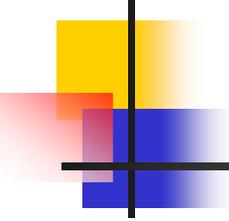
Tipos de Programación Automática

- **Deductiva:** Generar un programa a partir de una especificación a alto nivel (que sea más sencilla que el programa).
- **Inductiva:** Generación de programas a partir de ejemplos de uso
- **Deductiva-inductiva** (híbrida, multiestrategia, ...)



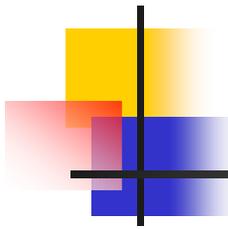
Prog. Automática Deductiva

- Sistemas Transformacionales y Deductivos (especificaciones en lenguajes formales, lógica de predicados: Refine, KIDS; Manna & Waldinger 92)
- Análisis de programas, optimización y transformación. Técnicas de compiladores:
 - memoization, orden sentencias, recursividad cola, ...
- Asistentes de programación (Apprentice)
- Programación Gráfica (conexión de componentes)



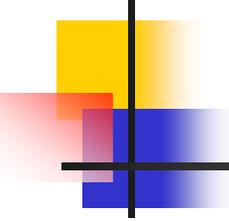
Ejemplo: Amphion [Stickel, 95]

- Dominio astronómico
- Interface gráfico para escribir especificaciones
- Se convierten a un teorema a demostrar
- Que es probado por un demostrador automático de teoremas (SNARK)
- La demostración es utilizada para componer un programa a partir de una librería de subrutinas



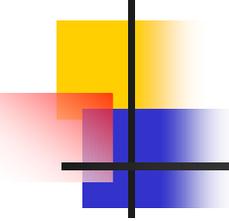
¿Dónde está la sombra de IO?

```
(all (time-voyager-2-c)
     (find (shadow-point-c)
           (exists
            (time-sun sun-spacetime-loc time-io io-spacetime-loc
                    time-jupiter jupiter-spacetime-loc time-voyager-2
                    voyager-2-spacetime-loc shadow-point jupiter-ellipsoid
                    ray-sun-to-io)
            (and
             (= ray-sun-to-io
              (two-points-to-ray
               (event-to-position sun-spacetime-loc)
               (event-to-position io-spacetime-loc)))
             (= jupiter-ellipsoid
              (body-and-time-to-ellipsoid jupiter
              time-jupiter))
             (= shadow-point
              (intersect-ray-ellipsoid ray-sun-to-io jupiter-ellipsoid))
             (lightlike? jupiter-spacetime-loc voyager-2-spacetime-loc)
             (lightlike? io-spacetime-loc jupiter-spacetime-loc))
```



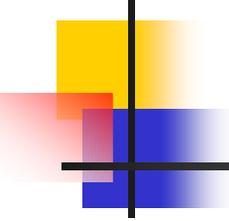
Ventajas/desventajas PA deductiva

- +: Se garantiza que los programas generados son **correctos**
- +: Más rápidos de programar, menor coste
- +: Mantenibilidad
- - : Es **difícil escribir especificaciones correctas y completas**, especialmente para problemas mal definidos



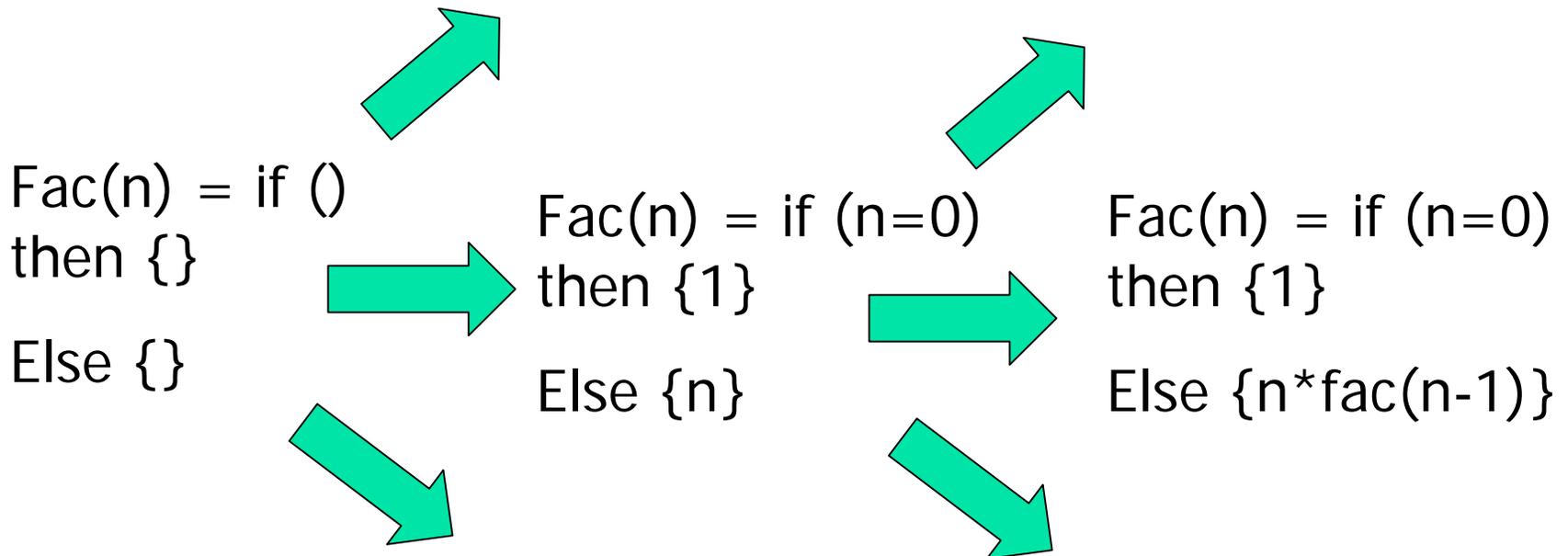
Ejemplo programación automática inductiva

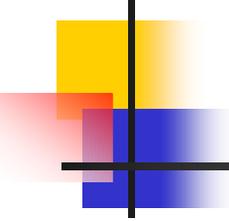
- Entrada:
 - $([2,1], [1,2]); ([2,3,1], [1,2,3]);$
 - $([3,5,4], [3,4,5]); ([], []); \dots$
- Instrucciones:
 - $(\text{dobl start end work}) (\text{wismaller } x \ y)$
 - $(\text{swap } x \ y) (\text{wibigger } x \ y)$
 - $(e1+ x) (e- x \ y) (e1- x)$
- Especificaciones sencillas, pero difícil garantizar que el programa es correcto completamente



Idea general PA inductiva

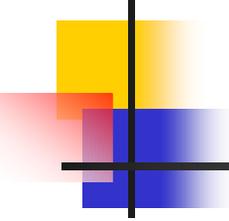
- Búsqueda en el espacio de programas de ordenador (transformar y probar, de manera incremental)





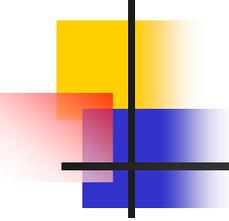
Problemas PA inductiva

- **Problema 1:** dado un lenguaje, el número de programas posibles crece exponencialmente
- **Problema 2:** los programas recursivos, iterativos, o con bucles son “**frágiles**”
- **Problema 3:** un programa puede no terminar (o tardar mucho)
- **Problema 4:** no se garantiza que el programa obtenido sea completamente correcto



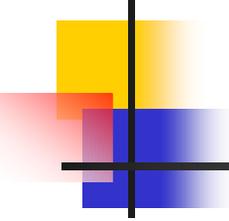
Programación Automática Inductiva

- Los comienzos: Síntesis de programas LISP
- “Programming by Example” o “Programming by Demonstration”
- Programación Lógica Inductiva (ILP)
- GAs:
 - Programación Genética
 - Grammatical Evolution
 - Immune Programming
- EDAs:
 - Evolución Incremental de Programas
 - Programación Automática Bayesiana
- A*: Programación Funcional Inductiva
- Levin search: Optimal Ordered Problem Solver



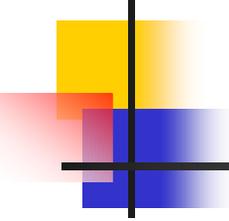
Síntesis de programas LISP

- Summers P. **1977**. "A Methodology for LISP Program Construction from Examples," Journal of the ACM
- $[(A), A]; [(A B), B]; [(A B C), C]$
- T1: $A = \text{primero}((A))$
- T2: $B = \text{primero}(\text{resto}((A B)))$
- T3: $C = \text{primero}(\text{resto}(\text{resto}((A B C))))$
- TK: $\text{ultimo} = \text{primero}(\text{resto} \dots \text{resto}(\text{lista}))$



Programming by demonstration (by example)

- Intenta programar siguiendo la relación maestro-alumno.
- El usuario muestra a la máquina como resolver el problema con uno o varios **ejemplos**
- La **traza** (pasos) de la ejecución está disponible
- El sistema intenta construir el procedimiento genérico
- El sistema puede ejecutar lo aprendido, para que el maestro **le corrija** ante los errores
- **Gráfico, interactivo**



PBD. Ejemplos

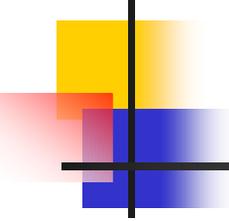
- Pygmalion [Cainfield, 1975]:
 - Se le muestra a la máquina cómo programar el factorial de manera gráfica, con un caso concreto (6!)
 - Ejecuta programas parcialmente definidos y descubre condiciones faltantes (ej: el caso base), que puede añadir el usuario
- Tinker [Lieberman, 1981]
 - Aprender a apilar bloques en el mundo de los bloques
 - Importancia de proporcionar una buena secuencia de ejemplos

PBD. Otras aplicaciones (macros)

- **Predictive calculator** (observa al usuario teclear y predice operaciones)
- **Smallstar**: aprende a hacer tareas compuestas en el desktop. Repeticiones y condicionales se añaden a mano
- **TELS**: Cambios de formatos en textos (bucles y condicionales)
- **Eager**: Aprende a detectar acciones repetitivas. Predice la siguiente y el usuario confirma o rechaza
- Predicción de URLs, generación de Web wrappers,
...

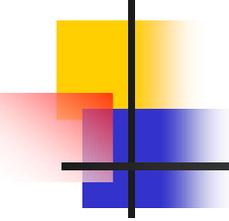
PBD. Stagecast (creación de juegos)





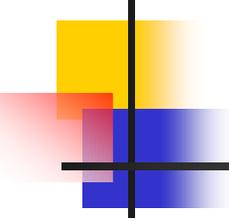
PBD. Bibliografía

- Watch What I do. Programming by Demonstration [Cypher, 93]
- Your Wish is My Command. Programming by Example [Lieberman, 01]



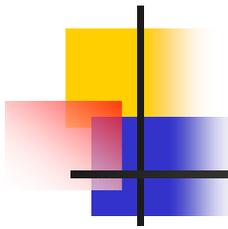
PBD. Conclusiones

- En la práctica, PBD se centra en la obtención de macros por observación
- Dominios específicos, relativamente sencillos (no es programación general)
- Dificultad: **es difícil capturar las intenciones del usuario** por mera observación (ej: ¿el usuario quiere llevar los ficheros a la carpeta “backup” siempre, o sólo en este caso concreto?)
- Buena idea: utilización de trazas (además de (entrada, salida))



Programación Lógica Inductiva (ILP)

- Aprendizaje Automático para inducir expresiones en lógica de predicados (PROLOG)
- Lenguaje más expresivo que el atributo-valor: cláusulas de Horn (“subrutinas”, recursividad, *background knowledge*, ...)
- Lenguaje compacto: ventajas y desventajas
- Orientación principal: aprendizaje inductivo de conceptos
- Técnicas:
 - Bottom-up: específico a general (GOLEM, CIGOL)
 - Top-down: general a específico (FOIL,)
 - Proposicional: (LINUS)



Ejemplo ILP

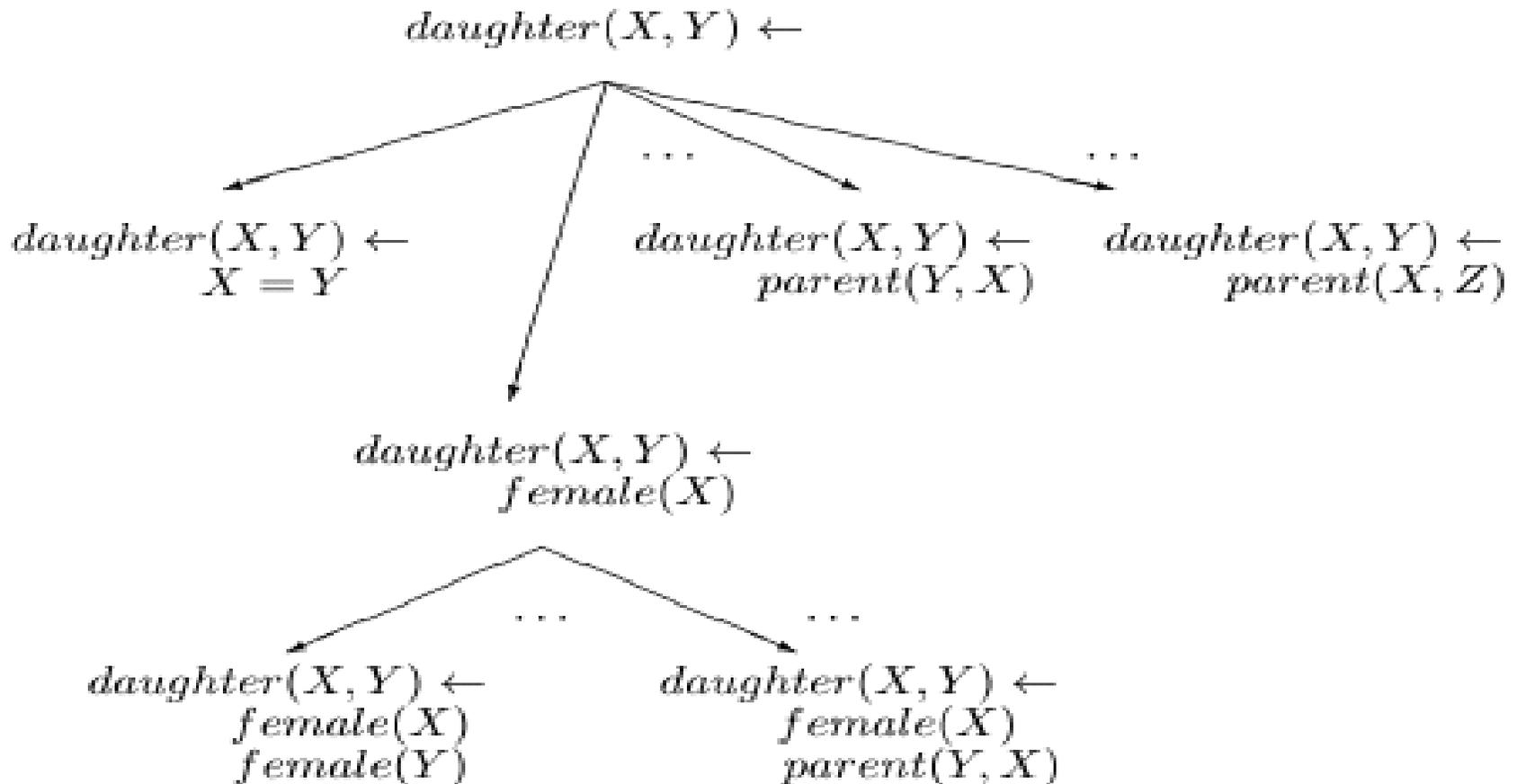
<i>Training examples</i>		<i>Background knowledge</i>	
<i>daughter(mary, ann).</i>	\oplus	<i>parent(ann, mary).</i>	<i>female(ann).</i>
<i>daughter(eve, tom).</i>	\oplus	<i>parent(ann, tom).</i>	<i>female(mary).</i>
<i>daughter(tom, ann).</i>	\ominus	<i>parent(tom, eve).</i>	<i>female(eve).</i>
<i>daughter(eve, ann).</i>	\ominus	<i>parent(tom, ian).</i>	

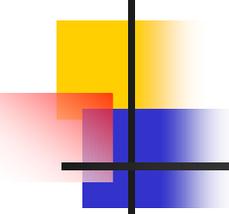
Conocimiento obtenido:

daughter(X, Y) ← female(X), mother(Y, X).

daughter(X, Y) ← female(X), father(Y, X).

Búsqueda de arriba a abajo





Ejemplo ILP recursivo

Positive examples E^+ :

```
ancestor(akel, andrej).  
ancestor(andrej, boris).  
ancestor(boris, miha).  
ancestor(akel, boris).  
ancestor(akel, miha).  
ancestor(andrej, miha).
```

Negative examples E^- :

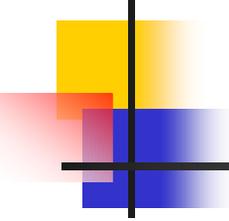
```
ancestor(akel, akel).  
ancestor(andrej, akel).  
ancestor(boris, andrej).  
ancestor(boris, akel).  
ancestor(miha, boris).  
ancestor(miha, akel).
```

Background knowledge

```
parent(akel, andrej).  
parent(andrej, boris).  
parent(boris, miha).
```

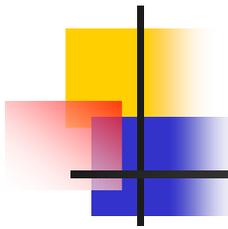
Conocimiento obtenido:

```
ancestor(X, Y) :-  
    parent(X, Y).  
ancestor(X, Y) :-  
    parent(X, Z),  
    ancestor(Z, Y).
```



Aplicaciones ILP

- Predicción de la estructura secundaria de proteínas
- Aprendizaje de modelos cualitativos
- Aprendizaje de reglas de diagnosis de enfermedades reumáticas
- Diseño por elementos finitos
- **Síntesis de programas lógicos**



ILP. Program Synthesis

Ejemplos + y -

$\text{subset}([], [])$

$\text{subset}([], [a, b])$

$\text{subset}([d, c], [c, e, d])$

$\text{subset}([h, f, g], [f, i, g, h, j])$

$\neg \text{subset}([k], [])$

$\neg \text{subset}([n, m, m], [m, n])$

Background knowledge: $\text{select}(a, [2, a, 3, 4], [2, 3, 4])$

$\text{select}(X, [X|Xs], Xs) \leftarrow$

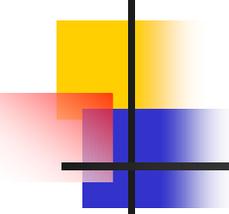
$\text{select}(X, [H|Ys], [H|Zs]) \leftarrow \text{select}(X, Ys, Zs)$

Programa obtenido

$\text{subset}([], Xs) \leftarrow$

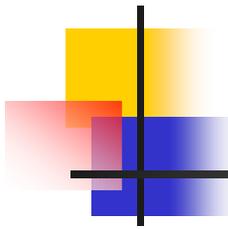
$\text{subset}([X|Xs], Ys) \leftarrow \text{select}(X, Ys, Zs), \text{subset}(Xs, Zs)$





Tipos de sintetizadores

- **Sin esquemas:** TIM, MARKUS, SPECTRE, MERLIN, WIM, FILP, ...
- **Con esquemas:** SYNAPSE, DIALOG, METAINDUCE, CRUSTACEAN, CLIP, FORCE2, SIERES, ...
- Pierre Flener, Serap Yilmaz. 1997. **Inductive Synthesis of Recursive Logic Programs: Achievements and Prospects** *Journal of Logic Programming*
- Flener et al 1994. **ILP and Automatic Programming: Towards Three Approaches**



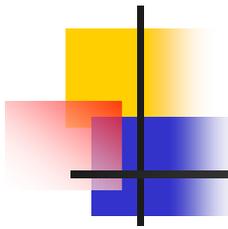
WIM (topdown, sin esquemas)

[Popelinsky, 95]

	Number of positive examples	Number of hypotheses tested
member	2	3
concat	2	6
append	3	6
reverseConcat	2	5
reverseAppend	3	14
split	2	7
sublist	4	15
union	4	15
quicksort	7	2307

Table 1: *Wim* results if no assumption was needed

5 segundos a 5 minutos

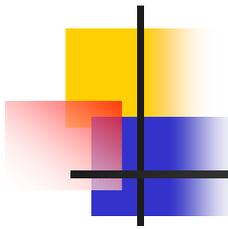


WIM. Resultados (1 query)

	Number of positive examples	Number of hypotheses tested
append	2	171
delete	2	21
last	2	99
plus	3	54
lessOrEqual	3	66
length	3	20
extractNth	3	27

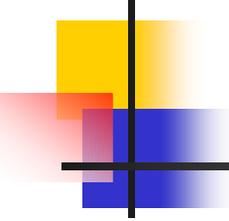
Table 2: *WiM* results with 1 membership query

Interactivo



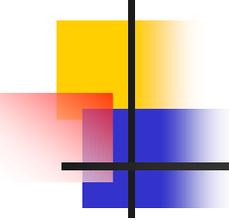
Código de quicksort

```
, logical program for quicksort
qsort([X|Xs], Ys) :-
    partition(Xs, X, S1, S2),
    qsort(S1, S1s),
    qsort(S2, S2s),
    append(S1s, [X|S2s], Ys).
qsort([], []).
```



Limitaciones

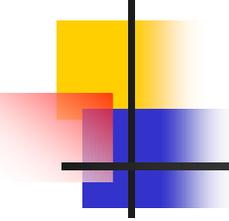
- ILP sin esquemas puede aprender *quicksort* a partir de ejemplos de *sort* además de *partition* y *append* como *background knowledge*. Pero está demasiado dirigido
- Los ejemplos son especificaciones demasiado débiles (ambiguas)
- A pesar de su nombre (ILP), no se han conseguido resultados importantes en inducción de programas



SYNAPSE (con esquemas)

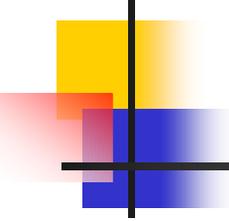
[Flener, 95]

- Objetivo:
 - Compress ([a,a,b,b,a,c,c,c], [a,2,b,2,a,1,c,3])
- A partir de ejemplos:
 - Compress ([],[])
 - Compress ([a], [a,1])
 - Compress ([b,b], [b,2])
 - Compress ([c,d], [c,1,d,1])
 - Compress ([e,e,e], [e,3])
 - Compress ([f,f,g], [f,2,g,1])
 - Compress ([j,k,1], [j,1,k,1,1,1])



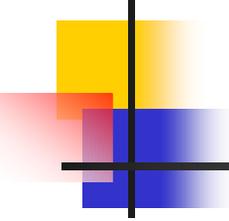
SYNAPSE. Propiedades

- A partir de propiedades:
 - Compress ($[X]$, $[X,1]$)
 - $X=Y \rightarrow$ Compress($[X,Y]$, $[X,2]$)
 - $X \langle \rangle Y \rightarrow$ Compress($[X,Y]$, $[X,1,Y,1]$)
- Interactivo



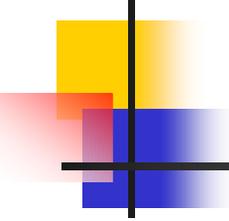
SYNAPSE. Esquemas

- Ej: divide y vencerás
- $R(X, Y)$ sii $Minimal(X), Solve(Y)$
- $R(X, Y)$ sii
 - $1 \leq k \leq c$
 - *Non-Minimal* (X)
 - *Decompose* (X, HX, TX)
 - *Discriminate* _{k} (HX, TX, Y)
 - $R(TX, TY)$
 - *Process* _{k} (HX, HY)
 - *Compose* _{k} (HY, TY, Y)



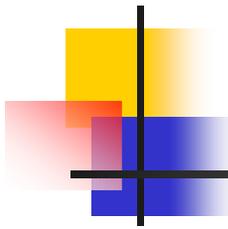
SYNAPSE. Algoritmo

- Fase de expansión:
 - Creación de una primera aproximación
 - Síntesis de *Minimal* y *Non-Minimal*
 - Síntesis de *Decompose*
 - Inserción de los átomos recursivos
- Fase de reducción:
 - Síntesis de *Solve*
 - Síntesis de *Process_k* y *Compose_k*



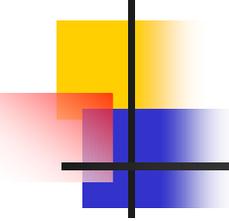
SYNAPSE. Otros problemas resueltos

- *Delete (E,L,R)* [6 ejemplos, 3 propiedades]
- *Sort (L,S)* [10 ejemplos, 1 propiedad, *split, partition*] obtuvo insertion-sort, merge-sort, quicksort



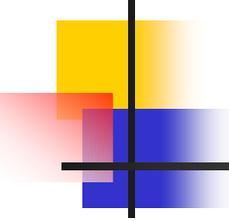
ILP para síntesis de programas. Conclusiones

- Recursividad y subrutinas se trata de manera natural
- Permite un tratamiento formal de los algoritmos
- Buena idea: búsqueda general/específica, específica/general
- Buena idea: esquemas
- Permite aprender programas sencillos
- No se ha ido mucho más allá



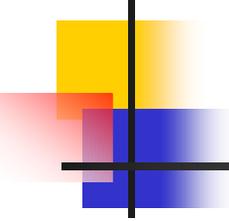
Otros resultados de ILP

- J. Stahl. 1993. Predicate Invention in ILP – An Overview. ECML (criticism)
- Rao. 2005. “Learning Recursive Prolog Programs with Local Variables from Examples”. ICML (one-recursive)



Programación Genética

- Algoritmos genéticos para evolucionar programas
- M. Cramer. 1985. [A Representation for the Adaptive Generation of Simple Sequential Programs](#), Proc. of an Intl. Conf. on Genetic Algorithms and their Applications.

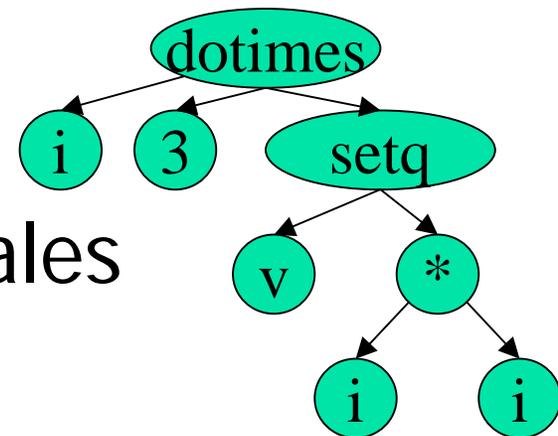


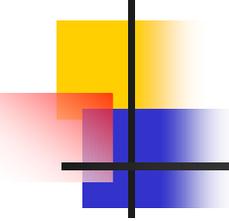
Programación Genética

- John R. Koza
- Non-Linear Genetic Algorithms for Solving Problems. United States Patent [4,935,877](#). Filed May 20, 1988. Issued June 19, 1990.
- 1992. [Genetic Programming: On the Programming of Computers by Means of Natural Selection](#). MIT Press.
- 1994. [Genetic Programming: On the Programming of Computers by Means of Natural Selection](#) MIT Press.
- 1999. [Genetic Programming III: Darwinian Invention and Problem Solving](#)
- 2003. [Genetic Programming IV: Routine Human-Competitive Machine Intelligence](#)

Representación de programas

- En Programación Genética se suele usar LISP
- Datos y programas se representan de la misma manera (listas o expresiones-S):
 - (dotimes i 3 (setq v (* i i)))
 - '(3 4 5 (a b c))
- Lenguaje = funciones + terminales

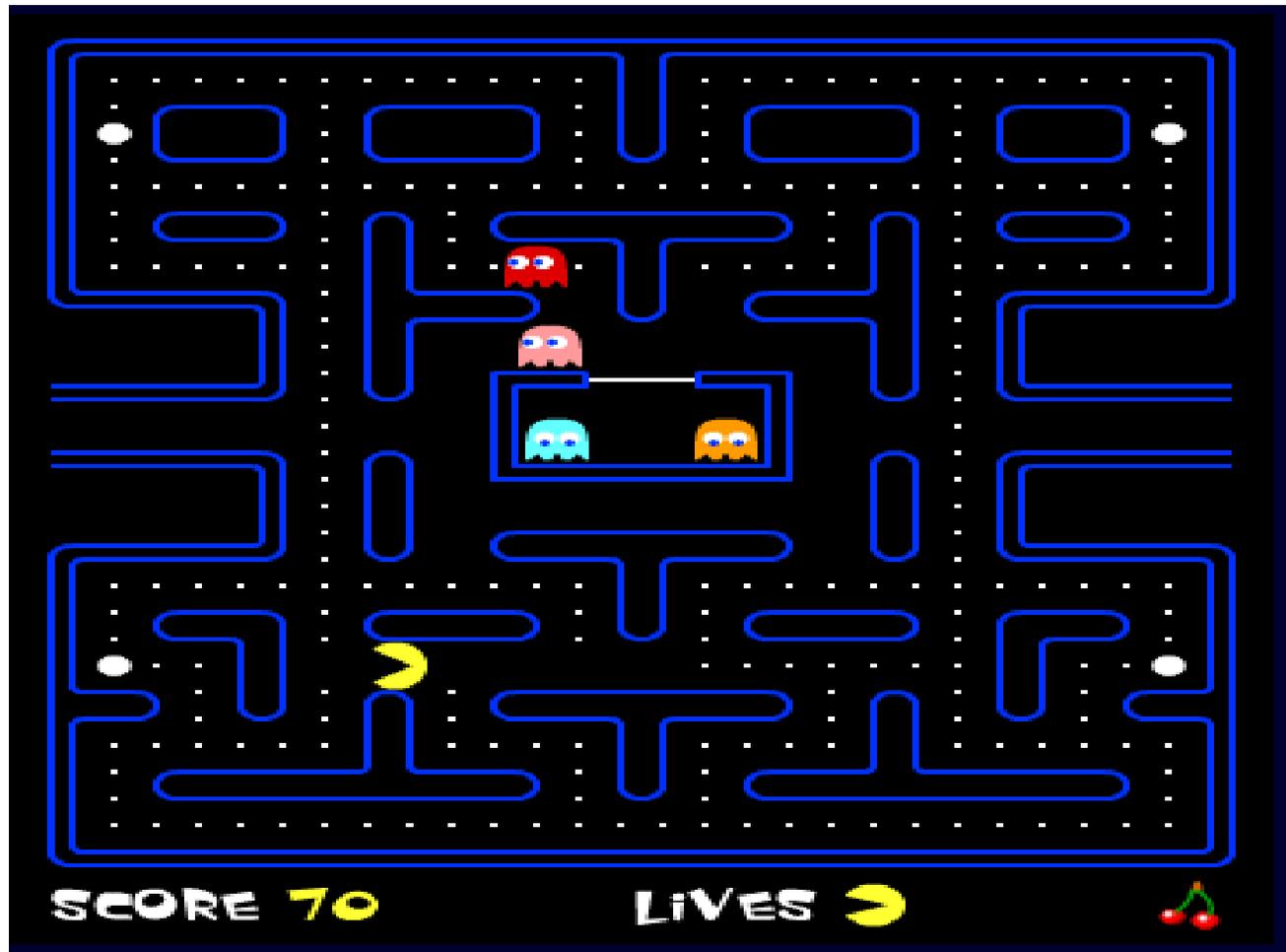


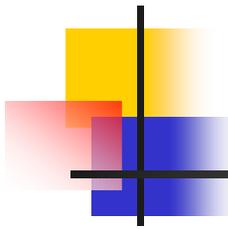


Ejemplo: Paridad par

- Generar un programa de ordenador que tome 10 bits y diga si el número de 1's es par:
 - Paridad-par(1,0,0,0,1,0,0,1,1) -> TRUE
- En términos de:
 - Funciones: AND, OR, NAND, NOR, NOT
 - Terminales: D0, D1, D2, ..., D9
- **QUÉ:** casos de prueba entrada/salida ($2^{10} = 1024$ casos)

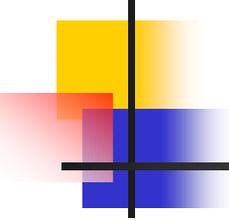
Ejemplo: Pac-Man





Ejemplo: Estrategia para Pacman

- Funciones:
 - si-obstaculo, si-punto, si-punto-gordo, si-fantasma, (son del tipo if-then-else)
 - secuencia2, secuencia3, secuencia4, ...
- Terminales: avanzar, girar-izquierda, girar-derecha
- **QUÉ**: comer todos los puntos del tablero en un tiempo límite



Ejemplo de programa en el Pacman

(si-fantasma

(secuencia3 (girar-izquierda)

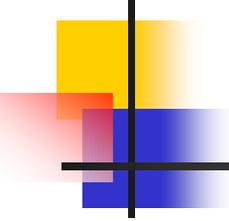
(girar-izquierda)

(avanzar))

(si-punto-gordo

(avanzar)

(girar-derecha)))

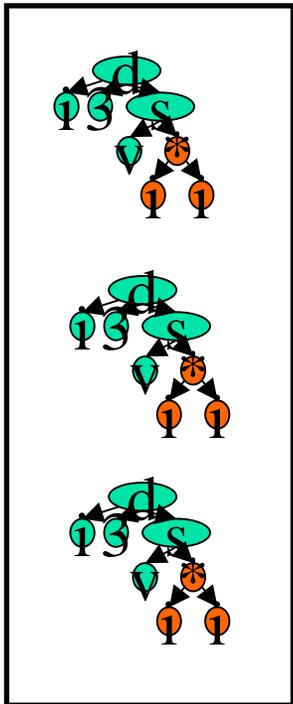


Algoritmo Programación Genética

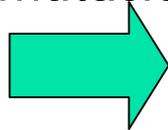
1. Creación de una población de programas de ordenador **aleatoria**, utilizando funciones y terminales
2. Ejecución de todos los programas y evaluación de los mismos (función de *fitness*)
3. Selección de los mejores programas
4. Creación de una nueva población aplicando los operadores genéticos a los seleccionados
5. Volver a 2 hasta encontrar un “buen” programa

Evolución

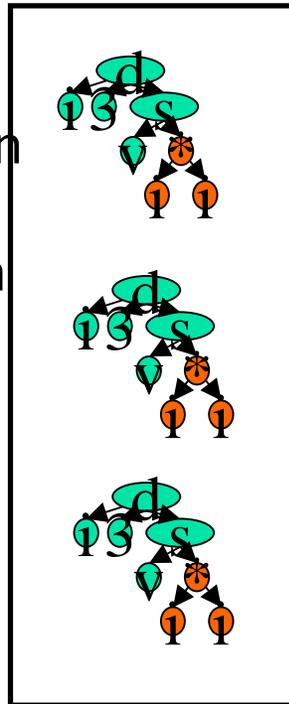
Generación 0



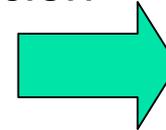
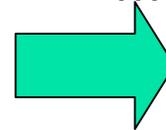
Selección,
cruce,
mutación



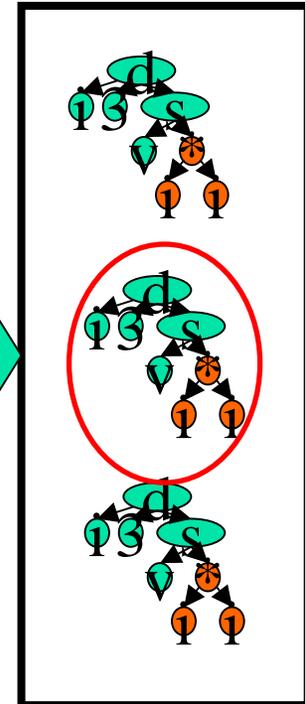
Generación 1



Selección,
cruce,
mutación

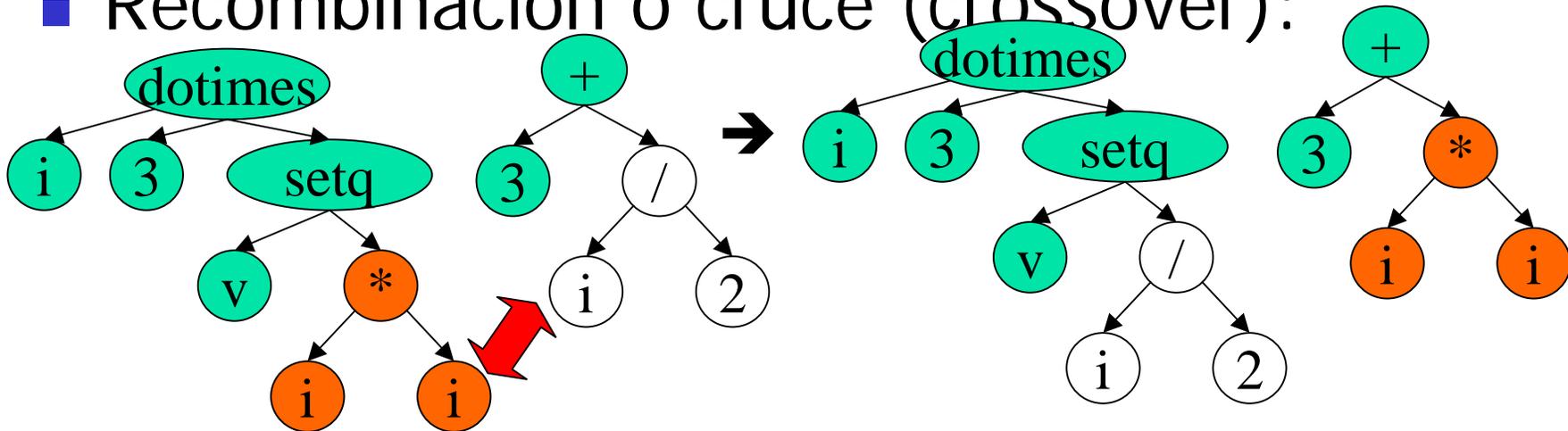


Generación N



Operadores genéticos

- Reproducción
- Mutación (subárbol)
- Recombinación o cruce (crossover):



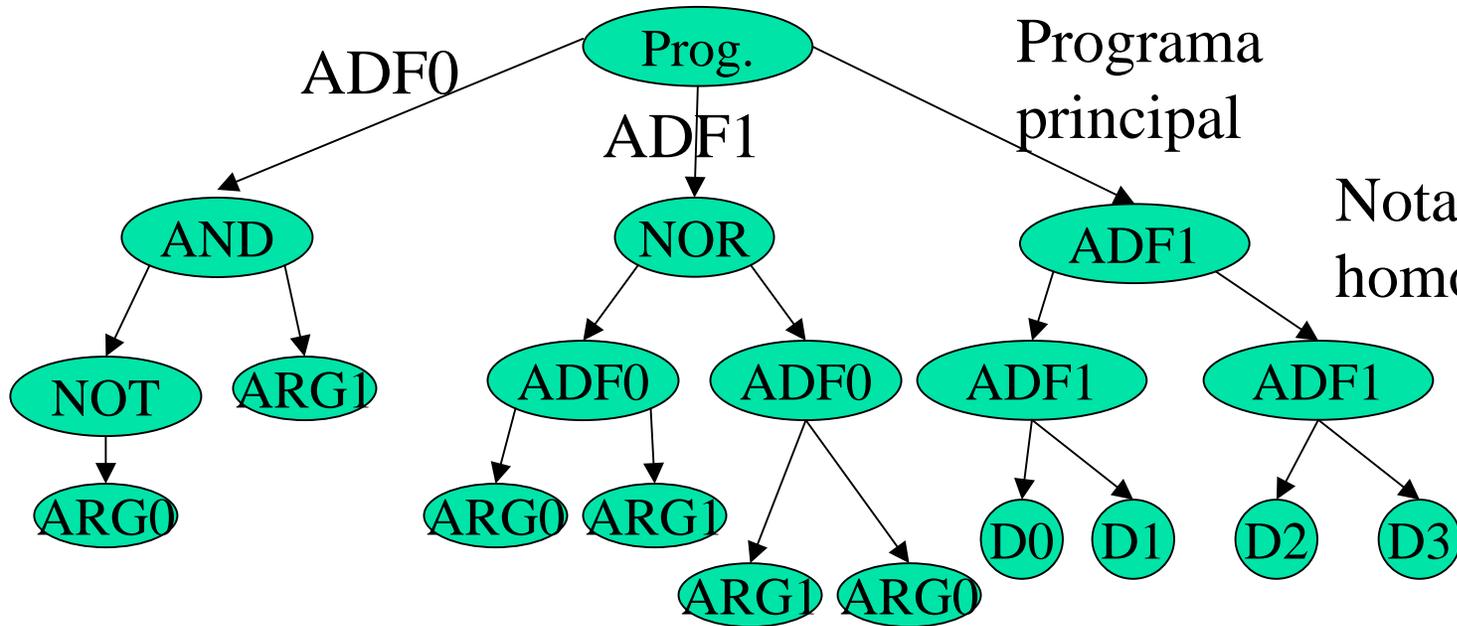
Esfuerzo computacional paridad-par

Paridad	Esfuerzo (evaluaciones)	Tiempo (horas)
3	96.000	
4	384.000	
5	6.528.000	
6	70.176.000	5h (P-1.5GHz)

Notas: No todas las evaluaciones consumen el mismo tiempo

La mayor parte del tiempo se consume en *fitness*

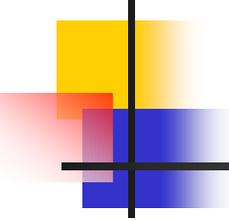
ADFs (Automatically Defined Functions: subrutinas)



Nota: cruce homólogo

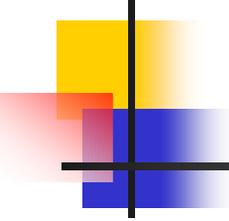
Esfuerzo y tamaño en paridad-par

Paridad	Esfuerzo sin ADF	Esfuerzo con ADF	Tamaño sin ADF	Tamaño con ADF
3	96.000	64.000 (x1,5)	44,6	48,2
4	384.000	176.000 (x2,18)	112,6	60,1
5	6.528.000	464.000 (x14,07)	299,9	156,8
6	70.176.000	1.344.000 (x52,2)	900,8	450,3
7 a 11	NO	SI		



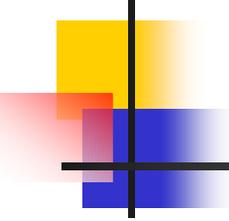
Trabajos sobre evolución de subrutinas

- Angeline PJ and Pollack JB. 1992. The Evolutionary Induction of Subroutines, *The Proceedings of the 14th Annual Conference of the Cognitive Science Society*.
- Rosca & Ballard. 1996. Discovery of Subroutines in Genetic Programming. *Advances in Genetic Programming II*.
- Ricardo Aler, David Camacho, Alfredo Moscardini. 2004. *"The Effects of Transfer of Global Improvements in Genetic Programming"*. Computing and Informatics



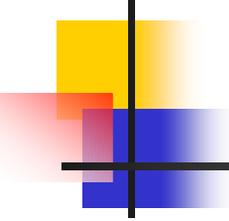
Oros elementos

- Uso de memoria: añadir funciones para leer y escribir en memoria
 - Ej: (assigna-m 3.0) (lee-m)
 - Ej: (assigna-array-m 5 3.0) (lee-array-m 5)
- Uso de bucles: añadir función que implementa el bucle:
 - (bucle contador inicio fin cuerpo)
 - (bucle i 1 10 (assigna-m (* lee-m i)))
- Idem con recursividad



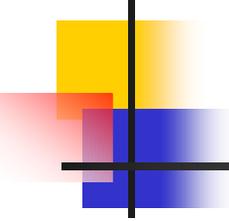
Limitaciones de los bucles/recursividad

- Con éxito en algunas aplicaciones
- A diferencia de las ADFs, no tan bien estudiados
- Incrementan **enormemente** el tiempo de evaluación de la fitness (incluso no terminación)
- Programas recursivos o iterativos son extremadamente **frágiles**
- Ejecutar individuos en paralelo y cancelar aquellos mucho peores que los ya encontrados
- Limitar el número de ciclos o llamadas recursivas (de manera proporcional al tamaño de la entrada)
- Recursión mediante **map**, **foldr**, ...



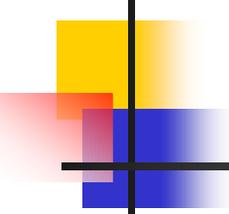
Trabajos en recursividad

- Wong. 2005. Evolving Recursive Programs by Using Adaptive Grammar Based Genetic Programming.
- Yu. 2001. Hierarchical Processing for Evolving Recursive and Modular Programs Using Higher Order Functions and Lambda Abstractions
- En: Genetic Programming and Evolvable Machines



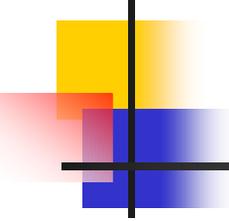
Variantes de PG

- PG tipada [Montana, 94]
- Evolución de estructuras de datos [Langdon, 98]
- **Evolución de código máquina** [keller, 96], [friedrich, 97]
- Uso de gramáticas [Leung, 95]
- Immune Programing [Musilek, 06]: hipermutación adaptativa



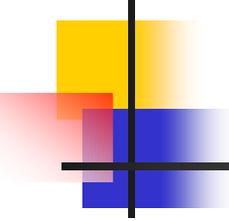
Criterios para “human-competitive”

- **(A)** Patentado previamente o podría ser una patente
- **(B)** Igual o mejor que un resultado científico ya publicado
- **(C)** Igual o mejor que resultados almacenados en bases de datos expertas
- **(D)** El resultado es publicable
- **(E)** Igual o mejor que resultados recientes en un problema atacado por una serie de algoritmos previos
- **(F)** Igual o mejor que algo que fue considerado un éxito en su momento
- **(G)** El resultado soluciona un problema de dificultad probada en un campo
- **(H)** El resultado gana o empata en una competición con humanos o programas desarrollados por humanos



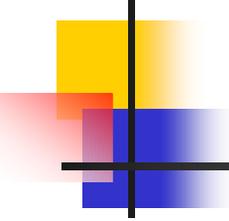
Éxitos de la PG

- Creación de algoritmos cuánticos mejores que los existentes (B, F, D)
- Aplicación a la Robosoccer (H)
- Aplicaciones a bioinformática (B, E)
- Aplicaciones a la síntesis de circuitos y antenas (A, B, D, E, F, G)
- Paralelización de programas de ordenador



PG en computación cuántica

- Creation of a better-than-classical quantum algorithm for the Deutsch-Jozsa “early promise” problem (B, F)
- Creation of a better-than-classical quantum algorithm for Grover’s database search problem (B, F)
- Creation of a quantum algorithm for the depth-two AND/OR query problem that is better than any previously published result (D)
- Creation of a quantum algorithm for the depth-one OR query problem that is better than any previously published result (D)
- Creation of a protocol for communicating information through a quantum gate that was previously thought not to permit such communication (D)
- Creation of a novel variant of quantum dense coding (D)



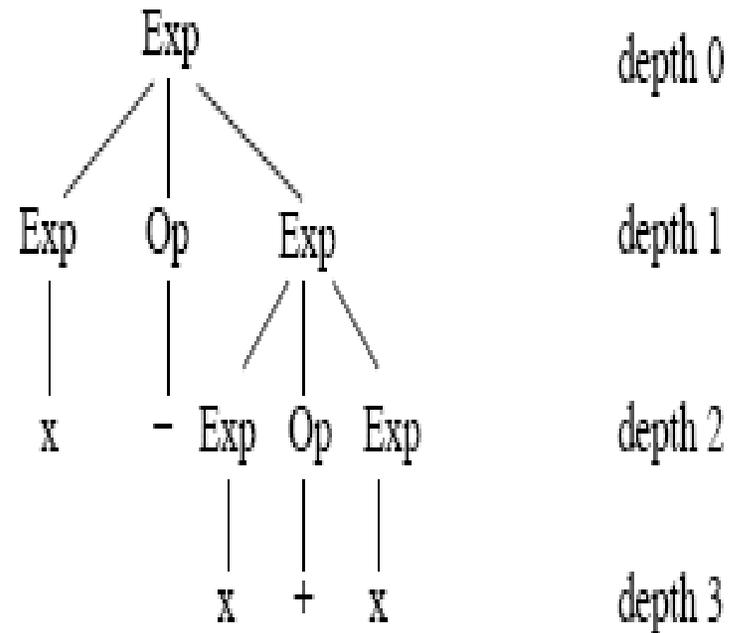
Conclusiones Programación Genética

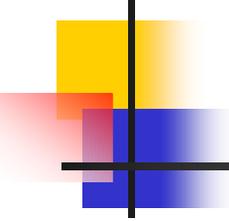
- Evolución de expresiones funcionales (sin bucles)
- Dificultad con los bucles y la recursividad
- Buena idea: utilidad probada de las ADFs
- Éxito en problemas reales
- Operadores genéticos tal vez poco apropiados (¿crossover = headless chicken operator?)

PG con gramáticas

Se utilizan gramáticas con las que se generan programas correctos

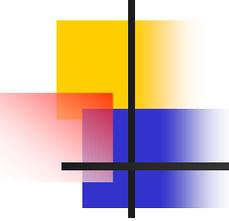
S	→	Exp		(0)
Exp	→	Exp Op Exp		(1)
	→	Pre Exp		(2)
	→	x		(3)
Op	→	+		(4)
	→	-		(5)
	→	×		(6)
	→	/		(7)
Pre	→	sin		(8)
	→	cos		(9)
	→	e [^]		(10)
	→	ln		(11)





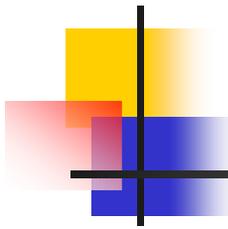
PG con gramáticas

- M. L. Wong, K. S. Leung. 1995. Genetic Logic Programming and Applications. IEEE Expert, 10(5).
- P. A. Whigham. 1995. Grammatically-based Genetic Programming. Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications.



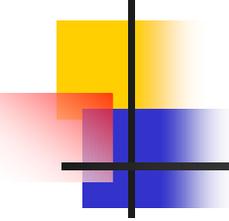
PG con gramáticas

- Para restringir el espacio de búsqueda a individuos correctos, context free grammars (CFG)
- Pasos:
 1. Generación de la población inicial con la gramática
 2. Evaluación
 3. Aplicación de operadores genéticos respetando la gramática
 4. Vuelve a 2



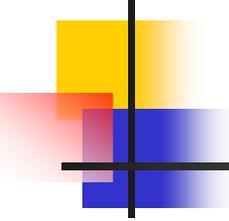
Adaptación de las gramáticas

- P. A. Whigham. 1995. Inductive Bias and Genetic Programming.
- Adaptar las probabilidades de uso de las reglas a partir de los mejores individuos (Stochastic CFG)
- Crear nuevas reglas a partir de los mejores individuos encontrados



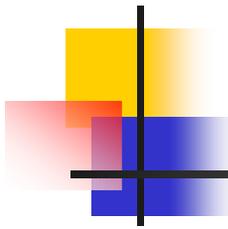
Grammatical evolution

- Ryan C., Collins J.J., O'Neill M. 1998. Grammatical Evolution: Evolving Programs for an Arbitrary Language. EUROGP.
- Evolucionan listas de enteros con un algoritmo genético, para determinar qué reglas utilizar, en caso de duda



Probabilistic Incremental Program Evolution (PIPE)

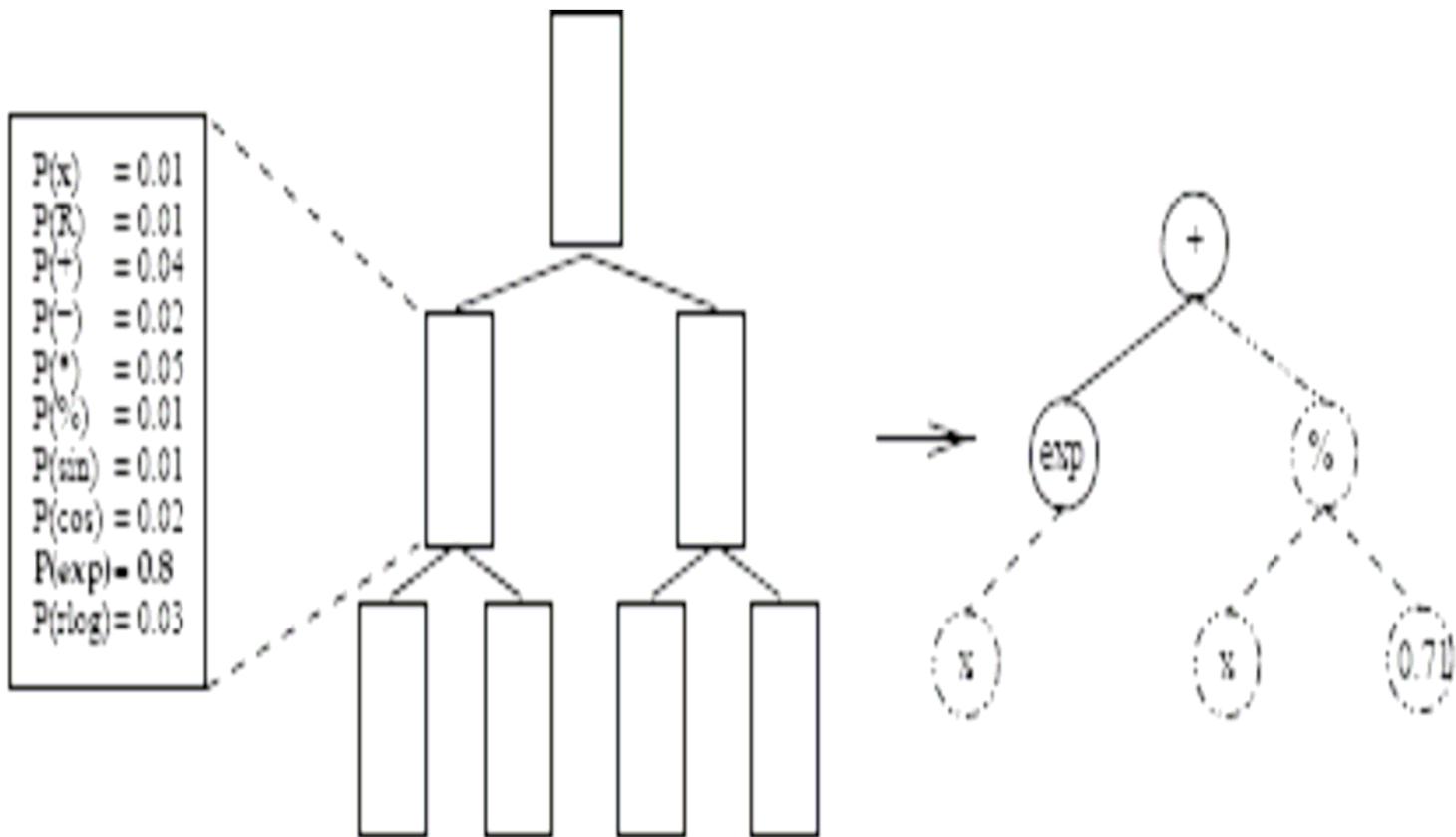
- Aplicación de algoritmos EDAs para la evolución de programas
- PIPE [Salustowicz, Schmidhuber, 97]
- Evolución de árboles (parse trees)
- Búsqueda en el espacio de distribuciones de probabilidad de árboles
- Busca encontrar una distribución que genere buenos árboles

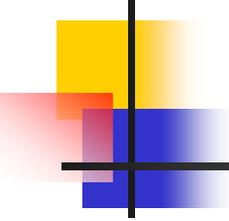


Probabilistic Incremental Program Evolution (PIPE)

- Se almacena una distribución de probabilidad sobre el conjunto de programas (probabilistic prototype tree)
- Cada generación, se incrementa la probabilidad de generar el mejor programa
- Evaluación paralela de programas, terminar cuando aparezca el mejor

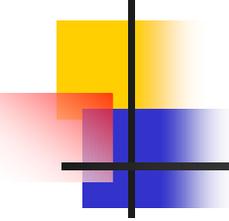
Probabilistic Prototype Tree (PPT)





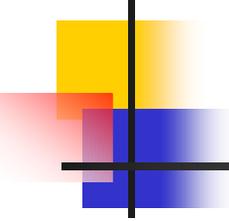
Algoritmo PIPE

1. Población inicial
2. Evaluación de la población
3. Aprendizaje (adaptación del PPT):
 1. Incrementar la prob. de que PPT genere el mejor programa de la generación actual
 2. Incrementar la prob. De que PPT genere el mejor programa encontrado hasta el momento
4. Mutación del PPT (exploración alrededor del mejor)
5. Poda del PPT (ramas con prob. muy baja)



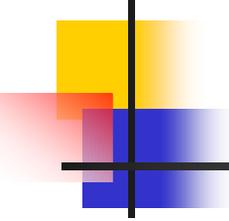
Resultados PIPE

- Regresión simbólica: PIPE mejor que GP en el 24% de las ejecuciones y peor en el 33%.
Mayor varianza
- 6-bit parity problem:
 - Más ejecuciones válidas (70% vs. 60%)
 - Más rápido (52476 vs. 120000 evaluaciones)
 - Más pequeños (61 vs. 90 nodos)
- [R. P. Salustowicz](#), [M. A. Wiering](#), [J. Schmidhuber](#). 1998. Learning Team Strategies: Soccer Case Studies. Machine Learning



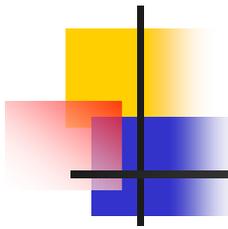
Limitaciones PIPE

- La distribución de probabilidad de cada nodo es independiente de la de los demás
- Los buenos subárboles (building blocks) son dependientes de la posición en el árbol, no pueden ser movidos a otros lugares



Extended Compact Genetic Programming (ECGP)

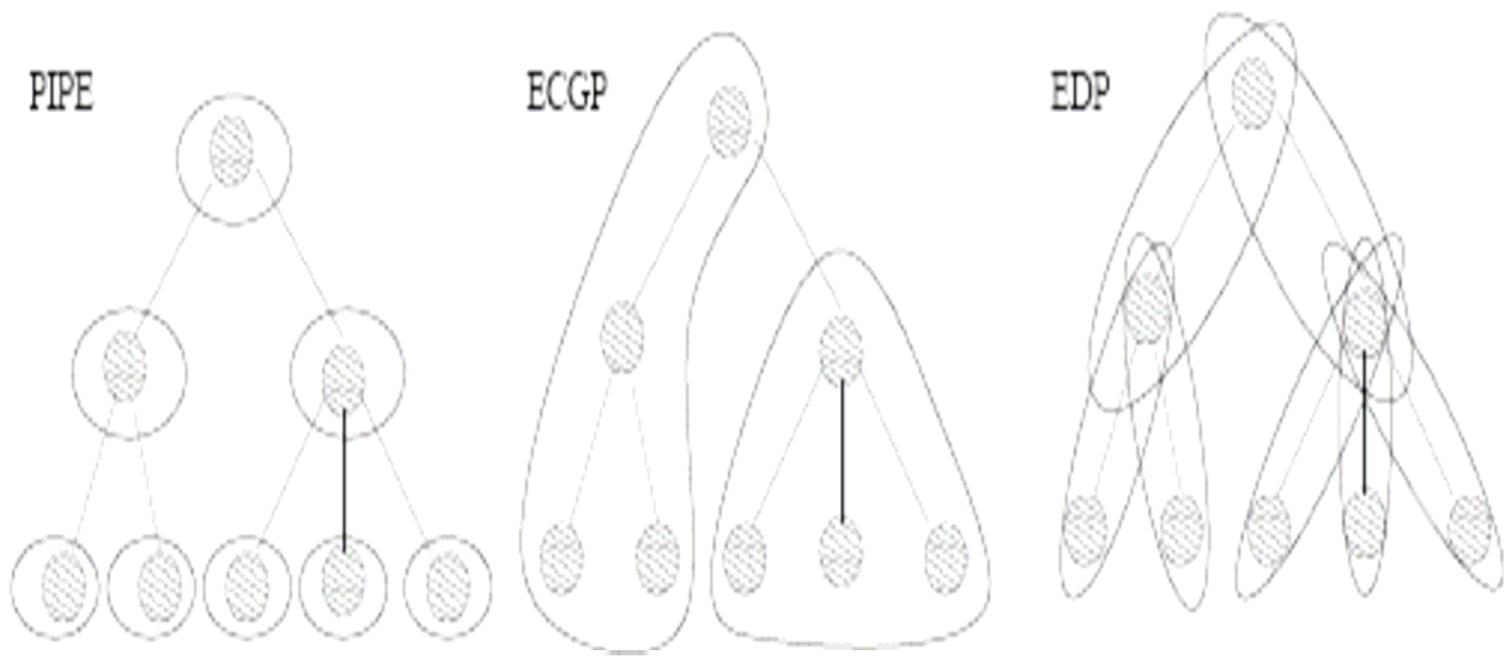
- K. Sastry, D.E. Goldberg. 2003. [Probabilistic model building and competent genetic programming.](#)
Genetic Programming Theory and Practice
- Utiliza Marginal Product Models
- Parte el PPT en varios subárboles, supuestamente independientes
- Cada subárbol con probabilidades conjuntas (no asume independencia de los nodos)

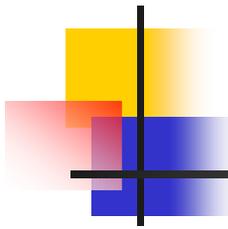


Estimation of Distribution Programming (EDP)

- Yanai, Iba. 2003. Estimation of distribution programming based on Bayesian network. CEC 2003.
- Utiliza probabilidad conjunta de nodo padre e hijo

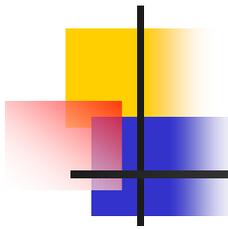
PIPE, ECJP, EDP





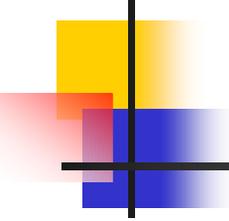
Inductive Functional Programming

- R. Olsson. **Inductive Functional Programming Using Incremental Program Transformation.** *Artificial Intelligence Journal.* 74:1. 1995
- ADATE: Automatic Design of Algorithms Through Evolution
- La PG no suele evolucionar recursividad o bucles
- El cruce es un mal operador de transformación de programas



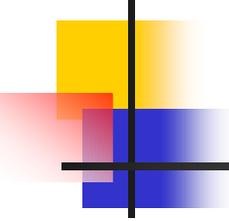
Especificaciones ADATE

- Un conjunto de **tipos**
- Funciones primitivas
- Tipo del resultado
- Un conjunto de entradas. Entradas de dificultad incremental y con casos especiales
- Una función (fitness) que evalúa la corrección del algoritmo con las entradas/salidas



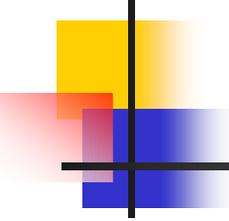
Búsqueda en ADATE

- Función heurística (fitness):
 - Corrección
 - Complejidad sintáctica (tamaño)
 - Complejidad en tiempo
- Iterative deepening (IDA*)



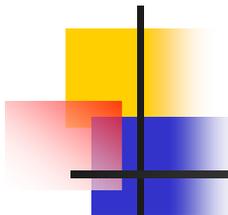
Lenguaje en ADATE

- Subconjunto de ML (lenguaje funcional)
- Definición de tipos
- Definición de funciones
- Let
- Case



Búsqueda en ADATE

- Se parte del programa vacío
- Se progresa explorando todos los programas de menor a mayor tamaño (navaja de Occam)
- Heurísticas en generación de expresiones:
 - En sentencias case, se debe activar más de una rama
 - En llamadas recursivas debe haber un parámetro que se vaya decrementando y un caso base
- Se generan nuevos programas mediante transformaciones compuestas (secuencias de trans. atómicas)
- Iterative Deepening (IDA*)



Transformaciones atómicas

- Reemplazo (R)
- Reemplazo que no empeora el programa (REQ)
- Abstracción: invención de nuevas funciones (ABSTR)
- Distribución case (CASE-DIST)
- “Embedding”: cambiar el tipo de la función para hacerla más general (EMB)

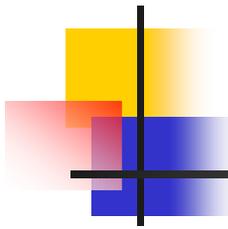
Ejemplo de reemplazo

```
fun sort Xs = case Xs of nil => Xs | X1::Xs1 => ?
```

```
fun sort Xs =  
  case Xs of nil => Xs  
  | X1::Xs1 => case Xs1 of nil => Xs | X2::Xs2 => ?
```

```
fun sort Xs =  
  case Xs of nil => Xs  
  | X1::Xs1 =>  
  case Xs1 of nil => Xs  
  | X2::Xs2 => case X2<X1 of true => ? | false => Xs
```

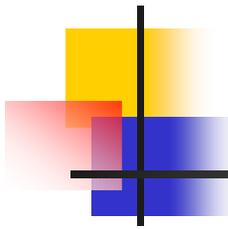
```
fun sort Xs =  
  case Xs of nil => Xs  
  | X1::Xs1 =>  
  case sort Xs1 of nil => Xs  
  | X2::Xs2 => case X2<X1 of true => ? | false => Xs
```



Ejemplo de abstracción

```
case sort Xs1 of nil => Xs
| X2::Xs2 => case X2<X1 of true => ? | false => Xs
```

```
fun sort Xs =
  case Xs of nil => Xs
  | X1::Xs1 =>
    let fun g V1 =
          case V1 of nil => Xs
          | X2::Xs2 => case X2<X1 of true => ? | false => Xs
        in
          g(sort Xs1)
        end
```



Distribución case

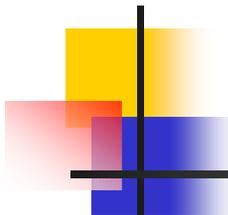
$h(A_1, \dots, A_i, \text{case } E \text{ of } Match_1 \Rightarrow E_1 \mid \dots \mid Match_n \Rightarrow E_n, A_{i+1}, \dots, A_m)$

case E **of**

$Match_1 \Rightarrow h(A_1, \dots, A_i, E_1, A_{i+1}, \dots, A_m)$

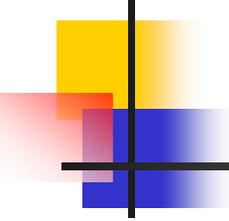
\vdots

$\mid Match_n \Rightarrow h(A_1, \dots, A_i, E_n, A_{i+1}, \dots, A_m)$



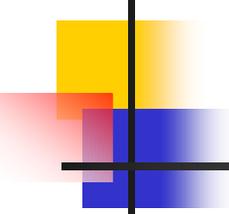
Transformaciones compuestas

1. $\text{REQ} \Rightarrow \text{R}$. The R is applied in the expression introduced by the REQ .
2. $\text{REQ} \Rightarrow \text{ABSTR}$. The ABSTR is such that the expression introduced by the REQ occurs in the $H(E_1, \dots, E_n)$ used by the ABSTR but not entirely in H .
3. $\text{ABSTR} \rightarrow \text{R}$. The R is applied in the the right hand side $H(V_1, \dots, V_n)$ of the **let**-definition introduced by the ABSTR .
4. (a) $\text{ABSTR} \rightarrow \text{REQ!}$ or (b) $\text{ABSTR} \rightarrow \text{REQ! REQ!}$. The $\text{REQ}(s)$ are applied in $H(V_1, \dots, V_n)$.
5. $\text{ABSTR} \Rightarrow \text{EMB!}$. The **let**-function introduced by the ABSTR is embedded.
6. $\text{CASE-DIST} \Rightarrow \text{ABSTR}$. The ABSTR is such that the root of $H(E_1, \dots, E_n)$ was marked by the CASE-DIST .
7. $\text{CASE-DIST} \Rightarrow \text{R}$. The R is such that the root of the expression Sub , which is replaced by the R , was marked by the CASE-DIST .
8. $\text{EMB} \rightarrow \text{R}$. The R is applied in the right hand side of the definition of the embedded function.



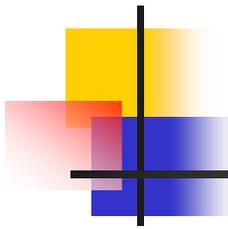
Problemas resueltos

- Simplifica polinomios: si hay coeficientes repetidos del mismo grado, los suma
- Intersección de rectángulos: devuelve la intersección de dos rectángulos, si la hay
- Generación de permutaciones: genera todas las permutaciones de una lista
- Container: mover cajas pequeñas dentro de container (<http://www-ia.hiof.no/~geirvatt/>)
- Otros: transposición de matrices, búsqueda de subcadenas, multiplica dos números binarios, encuentra caminos en grafos, ...



A DATE. Resultados

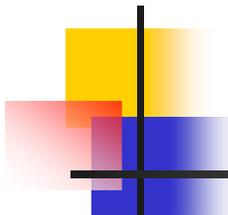
- Simplificación de polinomios
 - ["+", "=", "false", "true", "term", "nil", "cons"]
- Intersección de rectángulos
 - ["<", "point", "rect", "none", "some"]
- Inserción/borrado en árboles binarios
 - ["<", "bt_nil", "bt_cons", "false", "true"]
- Inversión, intersección, borrado en listas
 - ["false", "true", "=", "nil", "cons"]
- Generación de permutaciones
 - ["false", "true", "nil", "cons", "append"]
- Ordenación
 - ["false", "true", "<", "nil", "cons"]



A DATE. Resultados

<i>Problem</i>	<i>Run time in days:hours</i>
Polynomial simplification	0:7
Rectangle intersection	1:18
BST deletion	7:12
BST insertion	3:5
List reversal	0:10
List intersection	6:3
List delete min	8:8
Permutation generation	9:5
List sorting	1:12
List splitting	0:7



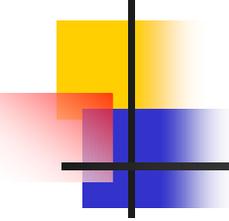


ADATE. Sort

```
program =
fun f (V2_4) =
  case V2_4 of
    nil => V2_4
  | cons( V2_aa, V2_ab ) =>
    let
      fun g2_d84e8 (V2_d84e9) =
        case V2_d84e9 of
          nil => cons( V2_aa, nil )
        | cons( V2_cbe81, V2_cbe82 ) =>
          case (V2_aa < V2_cbe81) of
            false => cons( V2_cbe81, g2_d84e8( V2_cbe82 ) )
          | true =>
            cons( V2_aa, V2_d84e9 )
        in
          g2_d84e8( f( V2_ab ) )
        end
    end
```

ADATE. Intersección de rectángulos

```
fun f
  ((
    ( V2_5 as rect(
      ( V2_6 as point( V2_7, V2_8 ) ),
      ( V2_9 as point( V2_a, V2_b ) )
    ) ),
    ( V2_c as rect(
      ( V2_d as point( V2_e, V2_f ) ),
      ( V2_10 as point( V2_11, V2_12 ) )
    ) )
  )) =
  case (V2_a < V2_e) of
  false => {
    case (V2_7 < V2_11) of
    false => none
    | true =>
    case (V2_8 < V2_12) of
    false => none
    | true =>
    case (V2_b < V2_f) of
    false => some(
      rect(
        point(
          case (V2_e < V2_7) of false => V2_e | true => V2_7,
          case (V2_8 < V2_f) of false => V2_8 | true => V2_f
        ),
        point(
          case (V2_a < V2_11) of false => V2_11 | true => V2_a,
          case (V2_b < V2_12) of false => V2_12 | true => V2_b
        )
      )
    )
    | true =>
    none
  }
}
```



Conclusiones

- Programación Automática: ir más allá del Aprendizaje Automático
- Abundancia de técnicas
- Resultados en el mundo real interesantes
- Importancia de la potencia computacional (ley de Moore)
- Necesidad de buenas ideas (optimal ordered problem solver, metalearning, ..., <http://www.idsia.ch/~juergen/>)