ADATE: Automatic Design of Algorithms Through Evolution

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

Motivation

- Loops and recursion are hard for GP
- Crossover is a very low-level program transformation operator
- Unlike GP, exhaustive search, from simple to complex programs
- Implicitely, it assumes Occam's Razor: simpler programs are more likely to be correct

Representation Language

- ML-ADATE: A functional language based on ML
- Why a functional language?: no global variables, effects of subexpresions are local, and changes to them remain local
- Usually functional programs are smaller than imperative ones (and the system looks for simple programs)

ADATE Lenguage

- Subset of ML
- Type definitions: tuples and lists
- Definitions of **Functions** (and **variables)** (in **let** sentences)
- **Case** sentences (conditionals)
- It allows for recursion

Case Definitions (boolean)

- If (A<B) then C else D

```
Case (A<B) of
    False => D
  | True  => C
```

Case Definition (boolean expressions)

```
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
```

Case Definitions (for types, like lists)

- If (A is the empty list) then B else C

```
Case A of
    Nil => B
    | A1::AS1 => C
```

- (A1::AS1 is a list made of an element A1 and a sublist AS1. A=[1,2,3], A1=1, AS1=[2,3])
- For data types, the branches in the case **must** correspond to the type definition
- A list can either be:
  - the empty list (NIL)
  - or a list made of head (A1) and rest (AS1)

Case Definition (list types)

```
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
```

A list can be either:

- The empty list NIL
- A construction of an element and a list: cons(element, list)

Function (subroutine) Definitions (local)

```
let
    fun g(x) = 3*x
    in
    g(5)
end
```

Function Definitions

```
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
```

Specifications in ADATE

- A set of types
- Primitive functions / terminals
- Type of the program *f* to be inferred
- A set of inputs $\{I_1, I_2, \ldots, I_n\}$
  - Well chosen, incremental difficulty and special cases
- A fitness function (output evaluation *oe*) that evaluates programs, taking into account the input/output pairs
  - $\{(I_1,f(I_1)), (I_2,f(I_2)), \ldots, (I_n,f(I_n))\}$

Partially Correct Programs

- They can return:
  - The correct answer
  - Don’t know (?)
  - The wrong answer
  - Maximum number of calls reached

The Output Evaluation Function (*oe*)

- Let P the candidate solution (program) to be evaluated
- Input to *oe*:
  - list of $[(I_1, P(I_1)), \ldots, (I_n, P(I_n))]$
- Output from *oe*:
  - Number of correct (N_c), wrong (N_w), and don't know answers
  - List of grades / fitness $[g_1, g_2, \ldots, g_k]$: list of real values that measure the quality of the P's outputs.

Input/Output Pairs and Grades

- In some cases an I/O specification is adequate:
  - Reverse list: ([1,2,3], [3,2,1]), ([2,1], [1,2])
- In other cases, a graded value is better
  - Pacman: [g_1 = number of points, g_2 = time the Pacman survived]
  - TSP: [g_1 = length of the path]
  - Shortest path for robot navigation, etc.

Example of Specification: Sort. I/O pairs

- ([], [])
- ([0], [0])
- ([0,1], [0,1])
- ([1,0], [0,1])
- ([0,1,2], [0,1,2])

- ([0,2,1], [0,1,2])
- ([1,0,2], [0,1,2])
- ([1,2,0], [0,1,2])
- ([2,0,1], [0,1,2])
- ([2,1,0], [0,1,2])

Specification of Sort. Datatype

- datatype list = nil | cons of int * list
- That is, a list of integers can either be:
  - An empty list (nil)
  - A construction of an integer and another list (like [1], [1,3], ...)

Specification of Sort. Primitives

- Funs_to_use = ["false", "true", "<", "nil", "cons"]
  - cons (a, (b c)) = (a, b, c)

- That is, very primitive functions indeed. Sort was built from scratch

Specification of Sort. Output Evaluation Function (*oe*)

- It just counts the number of correct and wrong outputs predicted, from the I/O set
- No grades are used (but they could be used, by measuring the degree of disorder in the output list or how far is an element from its final position)
- Ex: ([3,2,1], [1,2,3]), but P([3,2,1]) = [**2**,**1**,3]. $g_1 = 1 + 1 = 2$

Components for the Heuristic Functions

- Output evaluation function (oe) ("fitness"):
  - **N_c, N_w, [$grade_1$, ..., $grade_k$]**
- **S**: Syntactic complexity on the space of syntactically correct programs (*N* is the total number of nodes and m_i is the number of possible symbols at node *i*):

$$\sum_{i=1}^{\#N} \log_2 m_i$$

- **T**: Time Complexity:
  - Number of recursive calls and "calls" to *lets* for all inputs

ADATE Heuristic Functions pe_i

- Absolute fitness values are not assigned to programs. Instead, they are compared pairwise
- *Pe*$_i$, to minimize in lexicographic order (if draws in the first component, compare the second, and so on)

i	Value returned by $\mathtt{pe}_i$
1	$-N_c$ :: Grades @ [N_w,S,T]
2	$-N_c$ :: Grades @ [N_w,T,S]
3	[N_w,$-N_c$] @ Grades @ [S,T]

Atomic Transformations

- **R** (<u>Replacement</u>): Replacement changes part of the individual with new expressions. This is the only transformation that changes the semantics of the program
- **REQ** (<u>Replacement without making the individuals fitness worse</u>): Does the same as Replacement but now the new individual is guaranteed to have an equal or better fitness (several R are made, and the best of the non-worsening Rs is chosen)

Atomic Transformations

- **ABSTR** (Abstraction): takes an expression in the individual and puts that expression in a function in a *let...in* block and replaces the expression with a call to that function.
- **CASE-DIST** (Case distribution): takes a case expression inside a function call and moves the function call into each of the case code blocks.
- **EMB** (Embedding): changes the return type of functions in *let ... in* blocks, in order to make it more general

Example of Replacement

```
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
```

Expression Synthesis for Replacement

- They are generated (enumerated) from small to large, using case sentences, and the primitives

Restrictions in Expression Synthesis in Recursive Calls

- Let $g(A_1, A_2)$ be a recursive call within $g(V_1, V_2)$
- Then some A_i has to be smaller than V_i

```
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
```

Xs1 smaller than Xs = X1::Xs1

- It does not guarantee termination, and not all possible forms of recursivity are included
- But the aim is to reduce the number of synthesized expressions anyway

Restrictions in Expression Synthesis in Case Sentences

- More than one branch must be activated, otherwise the case sentence is removed
- The number of branches in the case expression depends on the type of the variable. If A is a list:

```
Case A of
    Nil => B
    | A1::AS1 => C
```

Abstraction (Function Definition)

$$H(E_1, E_2, \ldots, E_n) \Longrightarrow$$

$$\texttt{let fun g}(V_1, V_2, \ldots, V_n) = H(V_1, V_2, \ldots, V_n) \texttt{ in g}(E_1, E_2, \ldots, E_n) \texttt{ end},$$

Example of Abstraction

```
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
```

```
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
```

Distribution Case

$h(A_1, \ldots, A_i,$ `case` E `of` $Match_1$ `=>` E_1 | … | $Match_n$ `=>` $E_n, A_{i+1}, \ldots, A_m)$

$\Downarrow$

`case` E `of`

$Match_1$ `=>` $h(A_1, \ldots, A_i, E_1, A_{i+1}, \ldots, A_m)$

$\vdots$

| $Match_n$ `=>` $h(A_1, \ldots, A_i, E_n, A_{i+1}, \ldots, A_m)$

Compound Transformation. Coupling Rules

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

22 Compound Transformations (forms)

R
EMB R
REQ ABSTR REQ R
REQ ABSTR REQ REQ R
REQ ABSTR EMB REQ R
REQ ABSTR EMB REQ REQ R
REQ ABSTR EMB R
REQ ABSTR R
REQ R
CASE-DIST ABSTR REQ R
CASE-DIST ABSTR REQ REQ R
CASE-DIST ABSTR EMB REQ R
CASE-DIST ABSTR EMB REQ REQ R
CASE-DIST ABSTR EMB R
CASE-DIST ABSTR R
CASE-DIST R
ABSTR REQ R
ABSTR REQ REQ R
ABSTR EMB REQ R
ABSTR EMB REQ REQ R
ABSTR EMB R
ABSTR R

Search in ADATE

- Basically, exhaustive, no randomization, but uses some heuristics in expression synthesis and program generation
- It starts with the empty program ?
- Then program space is explored from small to large programs (Occam's Razor)
- New programs are generated by means of forms (compound transformations)
- Search = two nested iterative deepening processes

Search in ADATE. Iterative Deepening.

- $Work_i$ = number of individuals to be generated at iteration *i*
- $Work_0$ = 10000
- Every iteration, *Work* is increased exponentially:
  - $Work_{i+1} = 10000 * a^i$
  - a = 3 from theoretical and practical considerations

Search in ADATE. Primary Iteration

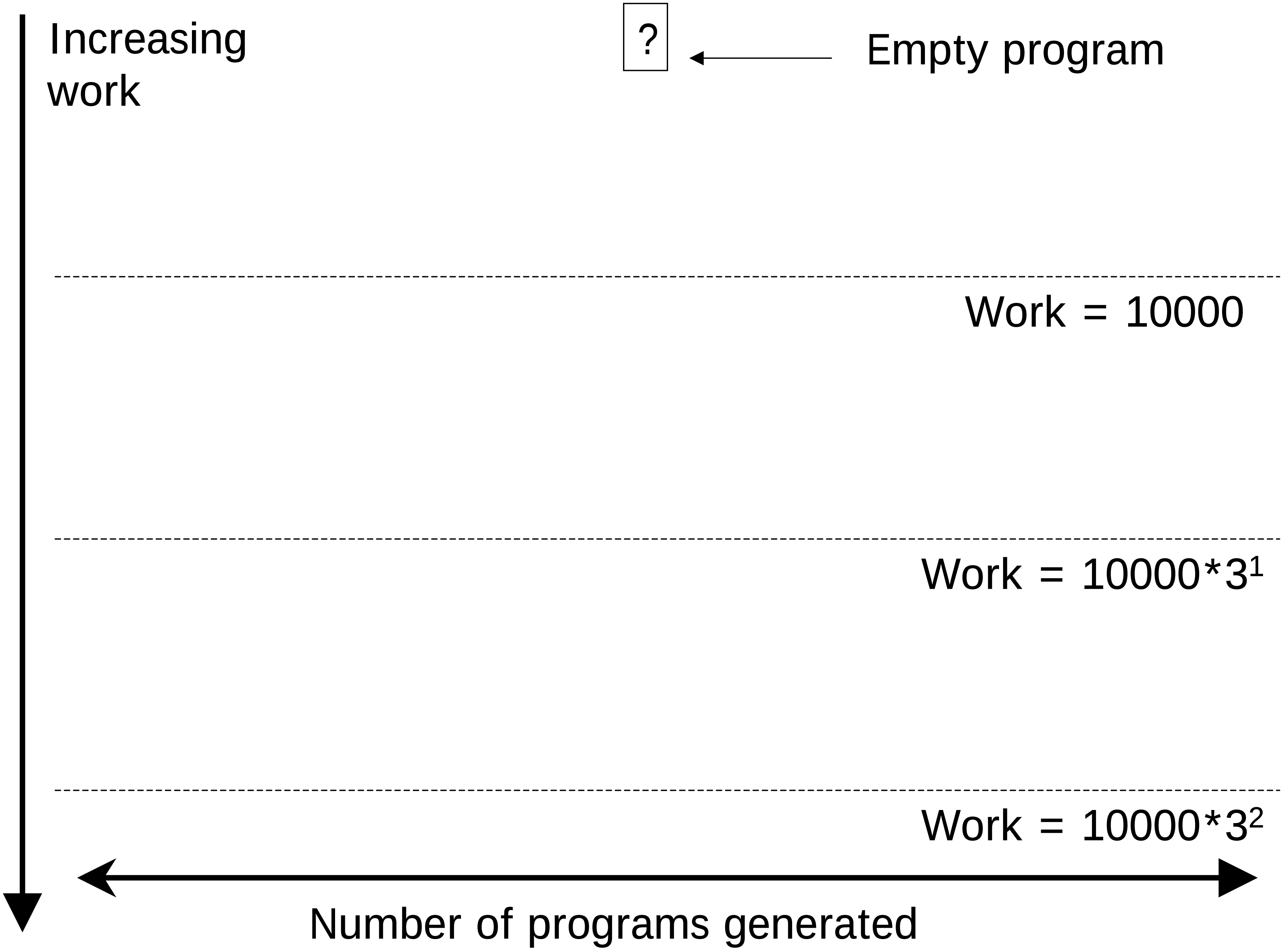
- Iteration 0: generate 10000 programs
- Iteration 1: generate 30000 programs
- Iteration 2: generate 90000 programs
- Etc.

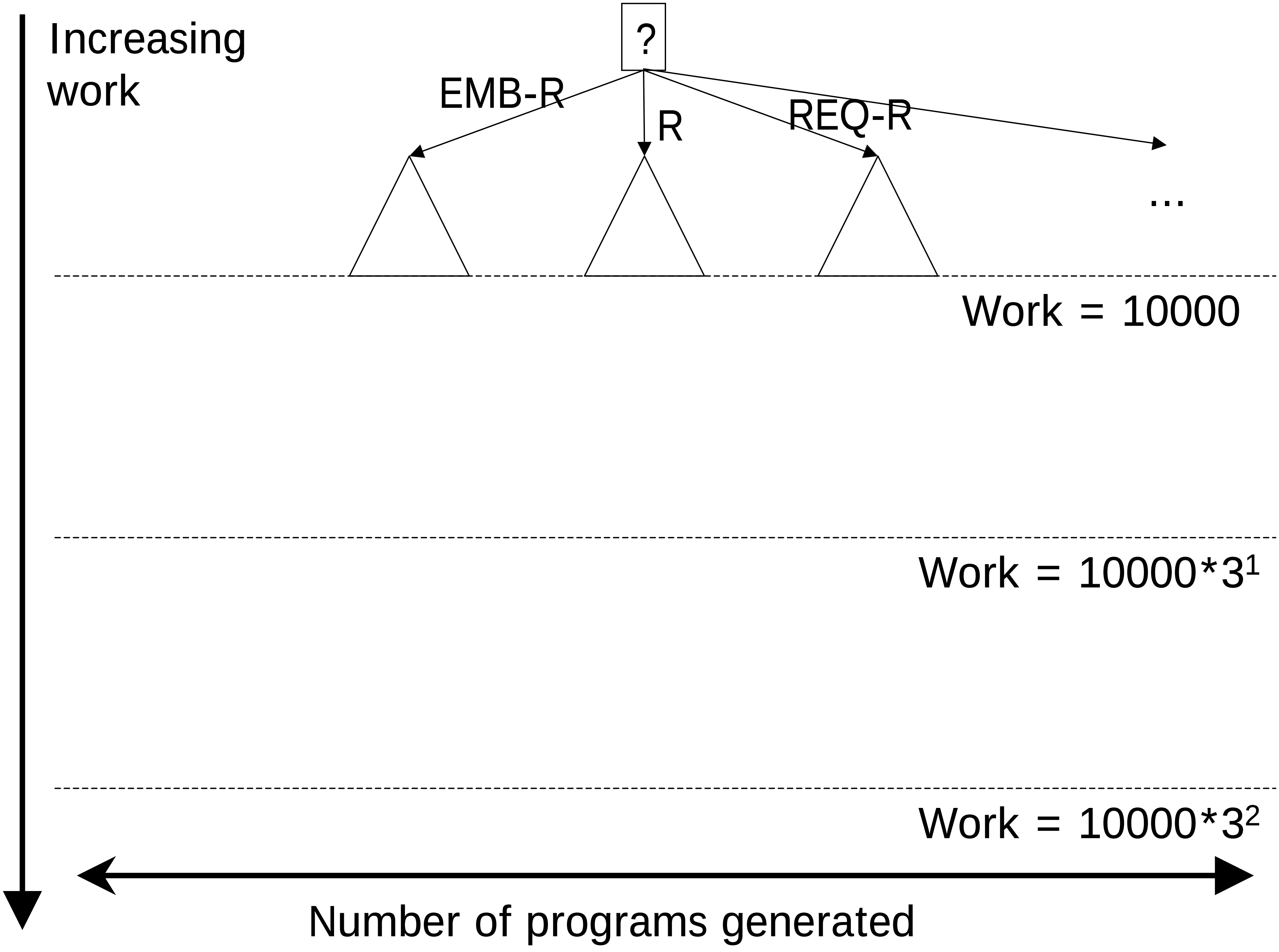
Iterative Deepening. Secondary Iteration

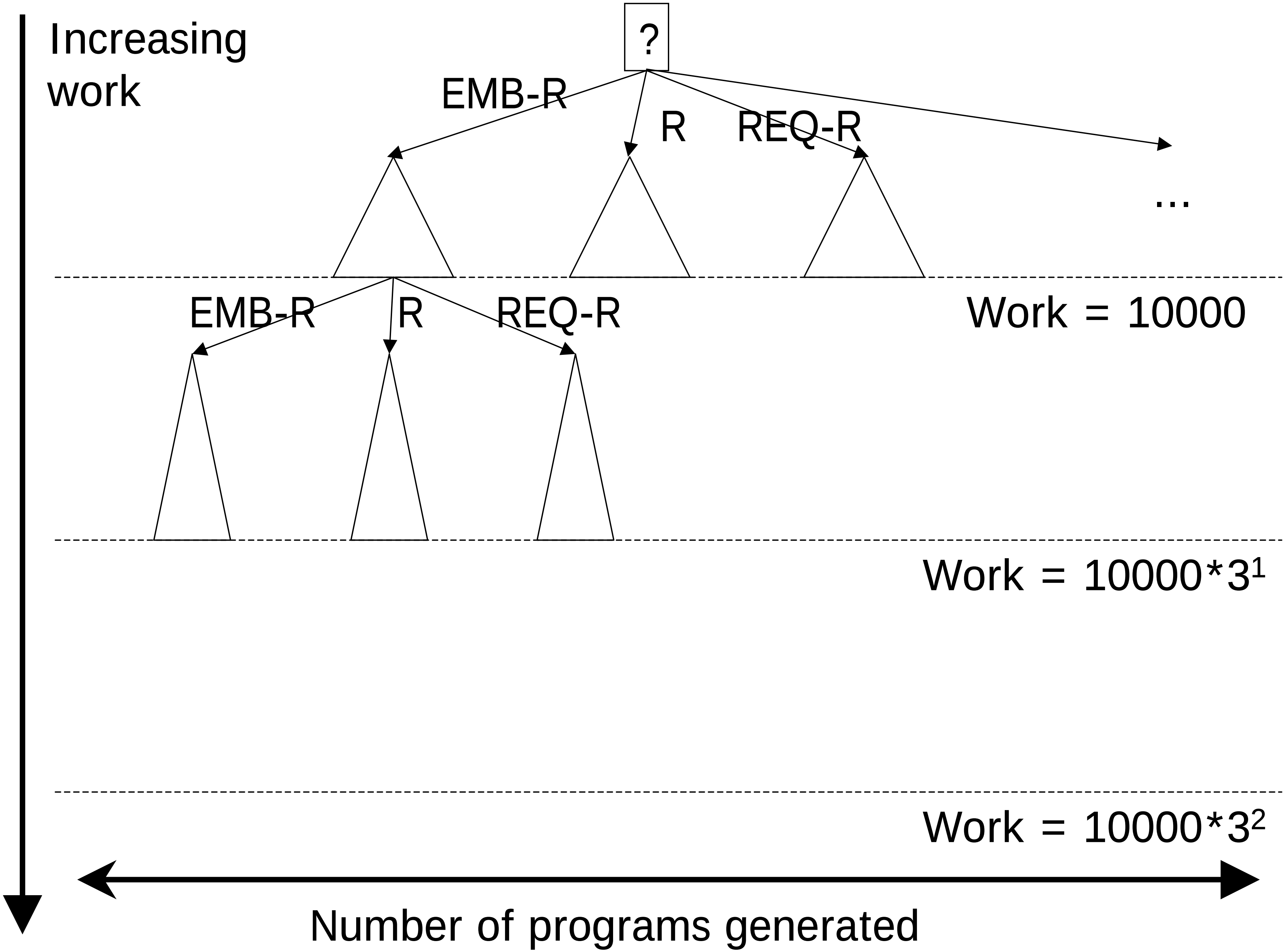
- $Work_i$, it is divided equally among all the forms (22 compound transformations)
- That is, for every form, $Work_i$ /22 programs should be produced

Iterative Deepening. Secondary Iteration

1. **Selection:** A program is picked from the population
2. **Generation**: Generate children of that program by performing one compound transformations of each form. No form can generate more than $Work_i$ /22 programs
3. **Insertion:** Check the children with the program evaluation functions to see if they are to be discarded or inserted into the population
4. Repeat step 2 and 3 for the forms until $Work_i$ programs have been produced. Then, go to 1


Increasing work
?
Empty program
Work = 10000
Work = 10000*3^1
Work = 10000*3^2
Number of programs generated


Increasing work
?
EMB-R
R
REQ-R
...
Work = 10000
Work = 10000*3^1
Work = 10000*3^2
Number of programs generated


Increasing work
?
EMB-R
R
REQ-R
...
EMB-R
R
REQ-R
Work = 10000
Work = 10000*3^1
Work = 10000*3^2
Number of programs generated

ADATE's Population

- The population is divided into:
  - Classes: programs with the same number of **case** sentences
  - Subclasses: programs with the same number of **let** sentences
  - Each subclass (c,l) contains three programs, the best one found so far according to pe_1, pe_2, and pe_3
  - (Recent versions include the time complexity as well as the syntactic one)
- The aim is to maintain diversity, avoid large programs eliminating small ones, and make sure that small programs are expanded first

ADATE Population Structure

Number of *case* sentences

p1, p2, p3 (0,2)	p1, p2, p3 (1,2)	p1, p2, p3 (2,2)	p1, p2, p3 (3,2)
p1, p2, p3 (0,1)	p1, p2, p3 (1,1)	p1, p2, p3 (2,1)	p1, p2, p3 (3,1)
p1, p2, p3 (0,0)	p1, p2, p3 (1,0)	p1, p2, p3 (2,0)	p1, p2, p3 (3,0)

Number of *let* sentences

Selecting the Next Program to be Expanded/Transformed

- A program is eligible for expansion, if it is better than all the programs (c,l)-simpler than itself. Better, according to at least one *pe_i*
- The program to be expanded will be the most (c,l)-simple, among all the eligible
- No program is ever expanded, if it contains more than 1.2 * case sentences than the best program found so far
- Note:
  - (c1,l1) < (c2,l2) if ((c1<c2) or ((c1=c2) and (l1<l2)))

Program Selection

Cases

p1, p2, p3 (0,2)	p1, p2, p3 (1,2)	p1, p2, p3 (2,2)	p1, p2, p3 (3,2)
p1, p2, p3 (0,1)	**p1**, p2, p3 (1,1)	p1, p2, p3 (2,1)	p1, p2, p3 (3,1)
p1, p2, p3 (0,0)	p1, p2, p3 (1,0)	p1, p2, p3 (2,0)	p1, p2, p3 (3,0)

#

In red, programs simpler than the green one

Inserting a New Program into the Population

- A program is rejected if it is no better than all its ancestors, for at least one pe_i
- The program is inserted into its (c,l) subclass, and replaces the *ith* program, if it is better than it, according to the corresponding pe_i function

Solved Problems by ADATE

- **Simplifying a polynomial**:
  - $(x^4+3x^2) + (x^3+2x^2) = x^4+ x^3+5x^2$
- **Intersecting two rectangles**
- **Permutating a list**: generate all permutations of a list
- **Container:** move small boxes inside a container (http://www-ia.hiof.no/~geirvatt/)
- Other: Reversing a list, List delete min, Intersecting two lists, Sorting a list , Locating a substring, Binary search tree insertion, Transposing a matrix, Binary search tree deletion, Path finding in graphs

Primitives Used to Solve the Problems

- Sorting a List
  - ["false", "true", "<", "nil", "cons"]
- Simplifying a Polinomial
  - ["+", "=", "false", "true","term", "nil", "cons"]
- Intersection of two rectangles
  - ["<", "point", "rect", "none", "some"]
- Inserting/deleting in binary trees
  - ["<", "bt_nil", "bt_cons", "false", "true"]
- Reversing/Intersection/Deleting in lists
  - ["false", "true", "=", "nil", "cons"]
- Permutation Generation
  - ["false", "true", "nil'", "cons", "append"]

Results (200MHz PentiumPro)

Problem	*Run time in days:hours*
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

Sort Program

```
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
```

Sort Program (O(n^2))

```
f(x) =
case x of
    []     => x
    A:AS =>
        g(y) =
        case y of
            []     => [A]
            B:BS => if (A<B) then B:g(BS) else A:y
        in g(f(AS))
```

Intersection of Two Rectangles

```
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
      )
```

Program Space Size

- Sorting program:
  - 96 bits -> 2^96 programs
  - 2^20 = 1048576
- Intersection of Two Rectangles:
  - 239 bits ->2^239 programs
- Huge program spaces!

Conclusions ADATE

- Incremental program construction is possible
- Heuristic functions work well in such huge spaces
- ADATE designed for synthesis of algorithms, not for the synthesis of numerical functions (lots of GP work belongs to this class)

Conclusions

- Different approaches to Automatic Inductive Programming:
  - Synthesis-based (functional, logic):
  - Search-based (GP, PIPE, ADATE, OOPS)

Conclusions

- Synthesis-based:
  - Algorithms with conditionals and recursion
  - Mostly, structural tasks
  - Use input/output pairs but no performance measure
  - Require few training instances, and few computational effort

Conclusions

- Search-based:
  - Generality, all kinds of tasks but …
  - High computational effort
  - I/O pairs & performance measures
  - GP: can evolve all kind of structures (mathematical expressions, and even circuits and antennaes), but recursion is hard
  - PIPE: Very similar
  - ADATE: more algorithmically orientated, deals well with recursion, higher level operators

Conclusions

- Already some remarkable results
- Computer power keeps growing, so much more is to be expected
- Heuristically guided incremental generation of programs is possible
- Why not combining synthesis and search based techniques? (suggested by U. Schmidt)
- Focus on the fact that it is computer programs that are to be generated, study better the space of useful computer programs

Generality / Computing Effort Tradeoff