OOPS: Optimal Ordered Problem Solver

Jürgen Schmidhuber. 2004. **“Optimal Ordered Problem Solver”**. *Machine Learning Journal, 54*

Time is the Key

- One of the most precious resources in (intensive) search-based AIP is time.
- Specially if programs can run for some unknown period of time (loops, recursion)
- Two kinds of techniques that handle time:
  - Saving time and speedup (parallelism, RAT algorithm, ...)
  - Allocating time appropriately to candidate programs

The Right to Exist

- Most search-based AIP techniques allocate time to individuals in a coarse way
- Individuals (candidate programs) either exist or they don't
- If they exist, total time is shared equally by all of them (usually until termination or until timeout)
- Better programs should be given more time to run!
- Exception: the coroutine model [Maxwell, 94]

Bias in AIP

- Bias: (informally) current belief about what programs are preferrable
- For instance, an initial bias could be:
  - Prefer shorter algorithms to longer ones
  - Make no preferences among all algorithms with the same length
- If P is a program, |P| is P's length, and |Q| is the number of primitives:
  - $\text{Bias} = \text{Prob}(P) = (1/|Q|^{|P|})*(1/2^{|P|})$
- Bias should be updated when the algorithm gains experience (this is what PIPE does)

Near-Bias-Optimal Allocation of Time

- Allocate time to programs proportionally to their probability
- Time(P) <= Prob(P)* Total-time

Levin Search

- Time iterative deepening (total time increases exponentially):

1. Phase = 0
2. Test all programs P such that:
   1. $Time(P) <= Prob(P) * 2^{phase}$
3. Phase = phase + 1, go to 2

Optimal Ordered Problem Solver (OOPS)

- Program representation: linear coding (any language, but more appropriate for machine code, stack-based, etc.)
- Self-delimiting languages
- Let t_1, t_2, ..., t_n a sequence of tasks
- For instance, t_1 = compute factorial(1), t_2 = compute factorial(2), ...
- They are solved in sequence by OOPS

Optimal Ordered Problem Solver (OOPS)

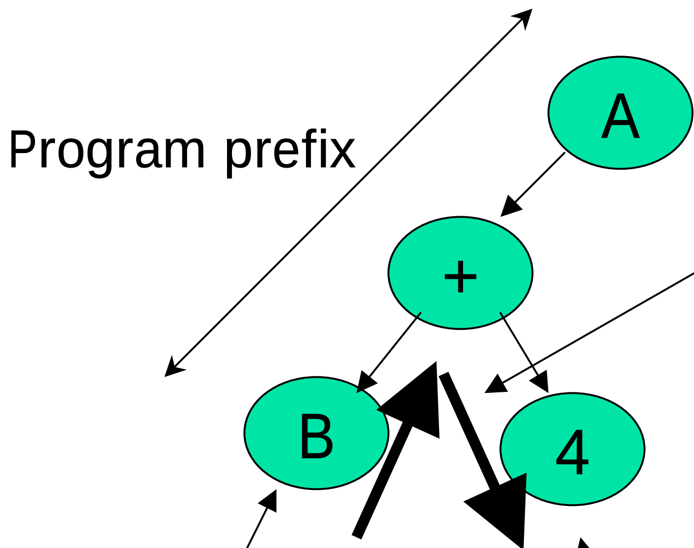
1. OOPS spends half of its time **extending** a previously found successful program for tasks t_1 to t_{n-1}, in order to find a solution to t_n (it takes advantage of previous experience)
2. OOPS spends half of its time triying to find a **new** program that solves t_1 to t_n

- The first branch is run on task t_n, the second branch runs all tasks t_1 to t_n in a time-sharing fashion
- Programs can call previous solutions as subroutines, and take code belonging to previous solutions and edit it

OOPS Algorithm

1. FOR n = 1 to number-of-tasks
   1. T := 2
   2. Spend T/2 extending successful code that solves t_1 to t_{n-1}, with the aim of solving t_n
   3. Spend T/2 testing fresh programs that solve all tasks t_1 to t_n
   4. T := 2*T; go to 2 until solution found

Growing New Programs in OOPS. Depth Search with Prob(P)*Time



Backtracking because
Time(“A+B”) >
Prob(“A+B”)*Total-Time

- Programs are grown 1 instruction when execution requires it
- Programs stop being executed when they exceed their allocated time, and then backtracking gives control to another program
- Program control can be transferred back (jumps, loops, …)

First program:

A+B

Second program:

A+4

OOPS Metalearning

- Programs in OOPS have instructions that can change the bias (the probability distribution that generates extensions to programs)
- Initially, when an instruction has to be grown, it is selected randomly
- But after bias shift, some instructions can become more likely than others, taking into account the program prefix so far (ej: Prob(4/"A+") >> Prob(B/"A+"))
- OOPS found a general solver for Towers-of-Hanoi, taking advantage of a general solver for creating 2^n1^n strings