Evolutionary Software Improvement for Instruction Set Meta-evolution

Michael Orlov and Moshe Sipper

Department of Computer Science
Ben-Gurion University
Israel

August 19, 2008

Evolutionary Software Improvement

- In Evolutionary Computation, only relatively small-scale solutions are viable
- Our goal: evolving large-scale solutions
- Method: *evolutionary software improvement*
- First step: Instruction set (meta-)evolution in the genetic-programming system Megavac
- Result: ESI process is feasible for evolutionary improvement of large software systems

Megavac

- Spatially structured, steady-state evolutionary platform
- Individuals are represented as cyclic linear programs (stack-based)
- Similar in concept to Avida (with emphasis is on EC and not ALife)
- Main components:
  - Genomes container (each in *wait* or *active* state)
  - Instruction scheduler (runs instructions of *active* genomes)
  - Connection topology (e.g., toroidal)
  - Selection method (e.g., tournament)
  - Mutator (variable-length mutations)
  - Reproducer (e.g., best-neighbor into worst-self)
  - Environment that provides inputs to genomes in *wait* state, and rewards genomes that send back correct outputs

Evolutionary Software Improvement: Requirements

1. There must be some aspect of the system that can be changed to improve some of the system's characteristics
   - not necessarily a specific component — can be some behavioral aspect
2. The chosen sub-system's function is representable as an evolvable program
   - requires definition of sufficiently expressive primitives
3. The chosen component or aspect has to be *amenable* to evolutionary improvement
   - the functionality should be sufficiently algorithmic in nature
   - comparison of evolved functionalities should be reasonably fast (this does not restrict the size of the system as a whole!)

Evolutionary Software Improvement: Process

1. Analyze the software system, and locate the aspect / component that can be expressed algorithmically
   - has to be sufficiently independent to be expressible with a reasonable-size program
   - must possess sufficient behavioral freedom to justify the evolutionary approach
   - substitution and evaluation of an altered component must be sufficiently brief
2. Define a fitness measure quantifying the performance of the component
   - must express objective software improvement goals: efficiency, quality, parsimony pressure, . . .
3. Analyze the chosen component, and define the language for expressing evolving individuals
   - primitives must allow the necessary freedom of expressed individuals
   - the existing component must be naturally expressible in the language — may be used to seed the initial population

Evolutionary Software Improvement: Process (contd.)

If the software improvement process evolves a better aspect or component, the system as a whole is improved using evolutionary computation.

If such a system is a state of the art software, the result may be human-competitive!

Megavac Evolutionary Improvement

Let's go over the ESI requirements and apply the ESI process.

- Key aspect to improve: instruction set (it's a first step for ESI...)
- Representable as evolvable program: simple bit-vector will do
- Amenability to evolutionary improvement: due to linear GP, Megavac is very fast
- Fitness measure: the area below the maximal fitness curve
    - nice property — independent from problems on which Megavac is run

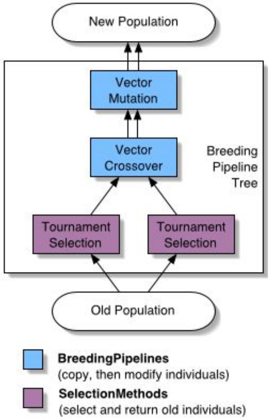
We develop problem-fitting instruction sets in meta-circular fashion.

Experimental Setup

- ECJ framework (by Luke and Panait) is used to evolve bit vectors
- Simple genetic algorithm is used
- Each bit vector of size 33 represents a subset of the complete Megavac instruction set
- ECJ:
  - population size — 40
  - 40 generations
  - single-point crossover $p_{\text{cross}} = 0.8$, single bit mutation $p_{\text{mut}} = 0.05$
  - tournament selection of size 2
- Megavac:
  - 1000 generations
  - 32 instruction execution rounds in each generation
  - population size 128, torus 4-neighbor topology
  - tournament selection includes all neighbors
  - variable-length mutation, data / control stack sizes of 4, 3 general-purpose registers

Experimental Setup Comments

ECJ setup is actually quite simple, this is a diagram straight from ECJ Tutorial:



Experimental Setup Comments

- Fitness of a bit vector is the area below max-fitness curve of one Megavac execution
  - this is the sum of per-generational maximal fitnesses
  - other possibilities: area below average-fitness curve, sum of fitness exponents (to emphasize higher Megavac fitness), ...
- Meta-evolution proceeds reasonably fast due to linear GP in Megavac
  - single run: 22 minutes on 2.6 GHz dual-core AMD Opteron
  - this is for evaluating 40 Megavac instances for 40 generations!
- ECJ easily supports parallelization
  - the architecture is scalable

Experiment: A multi-input problem

- Megavac facilitates *concurrent layered learning* via *composite environments*
- We define a composite environment with the following problems:
  - `Echo` — reward of 3.0 for returning the same input
  - `SubTwo` — reward of 15.0 for returning the difference of *two* inputs
  - `SubTwo` is impossible to evolve without `Echo`
  - evolving `SubTwo` requires > 10000 generations with the complete instruction set
- Five ECJ runs (meta-runs) in total
- Typical best-of-run result:
  - only 16 out of 33 instructions are enabled
  - optimal `Echo` individual: generation 52, `wait-read-send`
  - optimal `SubTwo` individual: generation 257, `wait-read-push-rswap(2)-sub-send`
  - multi-input, but one read per each send — surprising result!

Experiment: A multi-input problem (Results)

Table: Best-of-run instruction sets are shown. Average Megavac fitness is derived from dividing the meta-fitness by the number of Megavac generations, 1000. Average fitness of over 40.0 guarantees (a very good) ability to solve `SubTwo`.

Meta-fitness	Average	Instruction set
79104	79.1	`brge`, `brlez`, `brnz`, `call`, `dup`, `fitness`, `nop`, `pop`, `push`, `read`, `rswap`, `send`, `sub`, `swap`, `wait`, `zero` *(16 instructions)*
74278	74.3	`c2d`, `dup`, `erc`, `fitness`, `pop`, `push`, `read`, `rnd`, `rswap`, `send`, `store`, `sub`, `swap`, `wait`, `zero` *(15 instructions)*
82820	82.8	`add`, `brge`, `brlez`, `drop`, `dup`, `erc`, `jump`, `pop`, `read`, `send`, `sendn`, `sub`, `swap`, `wait` *(14 instructions)*
82742	82.7	`add`, `brge`, `call`, `drop`, `jump`, `load`, `nop`, `push`, `read`, `rswap`, `send`, `sendn`, `store`, `sub`, `wait`, `waitn` *(16 instructions)*
79348	79.3	`brge`, `brgez`, `brnz`, `c2d`, `d2c`, `erc`, `fitness`, `neg`, `nop`, `push`, `read`, `ret`, `rswap`, `send`, `store`, `sub`, `wait`, `waitn` *(18 instructions)*

Experiment: A multi-input problem (Results)

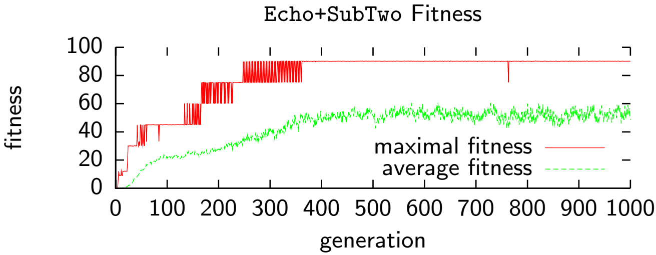
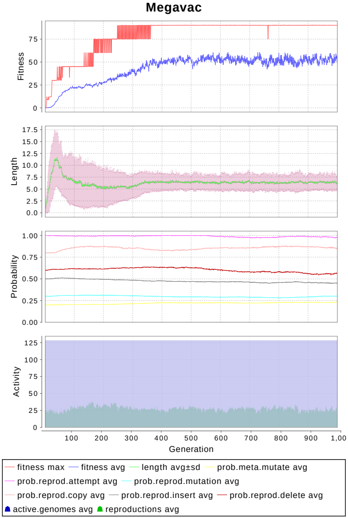


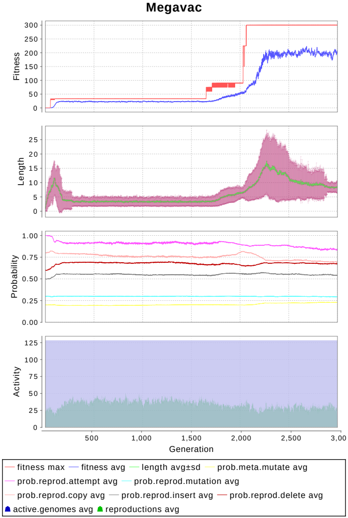
Figure: Fitness statistics after a typical run of the Megavac framework with the instruction set evolved to solve the `Echo+SubTwo` problem. Maximum and average fitness values per generation are shown in the plot.

Reduced Instruction Set Validation

- We extend the composite environment with another problem:
  - `SubSq` — reward of 75.0 for returning $x^2 - y$ for two inputs x and y
- As expected, Megavac does not evolve a solution to `SubSq` with the complete instruction set
  - not surprising, since even `SubTwo` needs > 10000 generations
- When the instructions set is restricted (the first best-of-run discussed previously), `SubSq` does evolve:
  - optimal individual `wait-read-swap-dup-push-mul-sub-send` appears at generation 2066
  - again, one read per each send

Typical Results for Both Environments





Conclusions

- The evolutionary software improvement process can be seen to weed out unnecessary, or at least less contributing, instructions, and improving the software system as a whole by reducing its complexity and tightening its code.
- We have shown the feasibility of evolutionary software improvement. Representing Megavac as a genetic program, and evolving it using traditional methods would not be possible. Instead, we located a critical component affecting Megavac's evolutionary performance—its instruction set, and evolved instruction subsets that drastically improved the performance.
- We view evolutionary software improvement primarily as a general technique for applying evolution to complex systems.

Discussion and Future Work

- Represent Megavac's reproduction process as an algorithm
  - GA is no more suitable, use genetic programming?
  - Requires careful definition of primitives, such as reproductive variation operators
  - Will an automatically evolved evolutionary algorithm find an interesting exploration / exploitation policy?
- Take an unrelated (non-evolutionary) system, and apply ESI
  - We want to show viability of evolutionary software improvement as a general technique

Thank You

Questions?