MOSES - Quick Start

Predrag Janicic, Revised by Linas Vepstas

3 November 2008, revised 14 December 2011

Abstract

Meta-optimizing semantic evolutionary search (MOSES) is a new approach to program evolution, based on representation-building and probabilistic modeling. MOSES has been successfully applied to solve hard problems in domains such as computational biology, sentiment evaluation, and agent control. Results tend to be more accurate, and require less objective function evaluations, than other program evolution systems, such as genetic programming or evolutionary programming . Best of all, the result of running MOSES is not a large nested structure or numerical vector, but a compact and comprehensible program written in a simple Lisp-like mini-language.

Contents

1 Introduction

Meta-optimizing semantic evolutionary search (MOSES) is a new approach to program evolution, based on representation-building and probabilistic modeling. MOSES has been successfully applied to solve hard problems in domains such as computational biology, sentiment evaluation, and agent control. Results tend to be more accurate, and require less objective function evaluations, in comparison to other program evolution systems. Best of all, the result of running MOSES is not a large nested structure or numerical vector, but a compact and comprehensible program written in a simple Lisp-like mini-language. For more information see `http://metacog.org/doc.html`.

The list of publications on MOSES is given in the References section. Moshe Look's PhD thesis [6] should be the first material to be read by someone interested in using or modifying MOSES.

MOSES is implemented in C++ and it heavily uses templates. So, one interested in modifying MOSES must be familiar with C++ and, at least to some extent, to C++ templates.

2 Copyright Notice

3 Installation

To compile MOSES, you need

- a recent gcc (4.x or later);
- the boost libraries (`http://www.boost.org`);
- the CMake package (`http://www.cmake.org/HTML/Index.html`);

For compiling MOSES, create a directory `build` (from the root folder of the MOSES distribution), go under it and run "`cmake ..`". This will create the needed build files. Then, make the project using "`make`" (again from the directory `build`). Generated executables will be in the folder `build/moses/learning/moses/main`.

4 Overview

MOSES performs supervised learning, and thus requires either a scoring function or training data to be specified as input. As output, it generates a Combo program that, when executed, approximates the scoring function. MOSES uses general concepts from evolutionary search, in that it maintains a population of programs, and then explores the neighborhood of modified, "mutated" programs, evaluating their fitness. After some number of iterations, the fittest program found is output.

More precisely, MOSES maintains a population of demes. Each deme is a program with many adjustable, tuneable parameters. These adjustable parameters are colloquially referred to as knobs. Thus, finding the fittest program requires not only selecting a deme, but also determining the best setting for the knobs.

The MOSES algorithm proceeds by selecting a deme and performing random mutations on it, by inserting new knobs in various random places. The best-possible knob settings for the mutated deme are found by using using existing, well-known optimization algorithms such as hill-climbing, simulated annealing or estimation of distribution algorithms (EDA) such as Bayesian optimization (BOA/hBOA). The fitness of the resulting program(s) can be compared to the fittest exemplar of other demes. If the new program is found to be more fit, it is used to start a new deme, (while discarding the old, dominated deme) and the process then repeats.

All program evolution algorithms tend to produce bloated, convoluted, redundant programs ("spaghetti code"). To avoid this, MOSES performs reduction at each stage, to bring the program into normal form. The specific normalization used is based on Holman's "elegant normal form", which mixes alternate layers of linear and non-linear operators. The resulting form is far more compact than, say, for example, boolean disjunctive normal form. Normalization eliminates redundant terms, and tends to make the resulting code both more human-readable, and faster to execute.

MOSES is able to out-perform standard genetic programming systems for two important reasons. One is that the EDA algorithms, by finding the dependencies in a Bayesian network, in fact find how different parts of a program are related. This quickly rules out pointless mutations that change one part of a program without making corresponding changes in other, related parts of the program. The other reason is that by performing reduction to normal form, programs become smaller, more compact, faster to execute, and more human readable. Besides avoiding spaghetti code, normalization removes redundancies in programs, thus allowing smaller populations of less complex programs, speeding convergence.

The programs that MOSES generates are "generic", in the sense that MOSES works with structured trees, represented in Combo. Such trees can represent propositional formula, procedural or functional programs, etc. The core MOSES solver is written in C++, and takes the form of a library. There are many example programs illustrating how to use this library.

5 Terminology

Here is some of the basic vocabulary used in MOSES:

Scoring function. During the learning phase, candidate programs being explored are scored by a *scoring function*. Ths is specific to the given problem. For instance, a given problem can be to learn a disjunctive propositional formulae over the given set of propositional variables, e.g., $\#1 \vee \#2 \vee \ldots \vee \#n$. The learning mechanism does not know this formula and instead it only has an access to the corresponding scoring function. A scoring function for this problem, for a given formula f (over the variables $\#1, \#2, \ldots, \#n$) would return the number of rows in the truth table for f such that the output truth value is equal to the truth value for $\#1 \vee \#2 \vee \ldots \vee \#n$. In this case, the scoring function could return values from 0 to 2^n. These values can be normalized or transformed in some other way. Usually, the perfect score is 0, while the worse scores are negative.

Exemplar. An *examplar* is a specific program; typically, the fittest one found.

Representation. The *representation-building* component of MOSES constructs a parameterized *representation* of a particular region of program space, centered around a single program (the *exemplar*), or around a family of closely related programs. The representation should be thought of as a program with a number of adjustable parameters or *knobs* that alter the program behaviour. During the optimization phase, the space of all possible parameter or knob settings will be explored, to locate the best possible settings, *i.e.* to find the fittest program.

Knobs. A representation built (around an exemplar) is given in terms of *knobs*. A *knob* is a single dimension of variation relative to an exemplar program tree. It may be discrete or continuous. For example, given the program tree fragment $or(0 < (*(\#1, 0.5)), \#2)$, a continuous knob might be used to vary the numerical constant. So, setting this knob to 0.7 would transform the tree fragment to $or(0 < (*(\#1, 0.7)), \#2)$. A discrete knob with $arity() == 3$ might be used to transform the boolean input $\#2$. So, setting this knob to 1 might transform the tree to $or(0 < (*(\#1, 0.7)), not(\#2))$, and setting it to 2 might remove it from the tree (while setting it to 0 would return to the original tree).

Program. A particular program is a representation together with a list of particular knob settings. These values can be bits, integer values, or real-number values, depending on the knob type.

Deme. A *deme* is a population of programs derived from one single representation. Thus, a deme can be thought of as a population of knob settings. During the optimization phase, an optimizer algorithm, such as hill-climbing, simulated annealing, or the Bayesian optimizaation algorithm is used to work with the population, locating the best possible knob settings for the given representation.

Metapopulation. MOSES maintains a collection of demes, playing each off the others. This set of demes is refered to as the *metapopulation*. Pairs of demes are in competition; fitter demes are used to replace less fit demes.

Domination. One deme is considered to *dominate* another if it is better in every way. During learning, demes typically do not dominate one-another: some are beter at some parts of the scoring function, while others are better at other aspects. Thus, both are kept around and further evolved. Demes that are completely dominated are (usually) discarded.

6 Overall Algorithm

The basic MOSES algorithm can be described as follows:

1. Construct an initial set of knobs (via representation-building) based on some prior (e.g., based on an empty program) and use it to generate an initial random sampling of programs. Add this deme to the metapopulation.
2. Select a deme from the metapopulation and iteratively update its sample, as follows:
   (a) Select some promising programs from the deme's existing sample to use for modeling, according to the scoring function. Ties in the scoring function are broken by preferring smaller programs.
   (b) Considering the promising programs as collections of knob settings, generate new collections of knob settings by applying some (competent) optimization algorithm.
   (c) Convert the new collections of knob settings into their corresponding programs, evaluate their scores, and integrate them into the deme's sample, replacing less promising programs.
3. For each of the new programs that meet the criteria for creating a new deme, if any:
   (a) Construct a new set of knobs (via representation-building) to define a region centered around the program (the deme's exemplar), and use it to generate a new random sampling of programs, producing a new deme.
   (b) Integrate the new deme into the metapopulation, possibly displacing less promising demes.
4. Repeat from step 2.

Note that representation-building and optimization algorithm that parametrize the above algorithm are vital components of the system and they crucially influence its performance. Representation building is specific for each domain

(e.g., learning propositional formulae), while optimization algorithm is general. Currently, in MOSES, there is support for representation building for several problem domains (e.g., propositional formulae, actions[1]) and several optimization algorithms.

7 Source Files and Folders

In the MOSES distribution, there are the following folders (under `moses/learning/moses`) with the source files:

moses This is the folder with the core support for MOSES — support for representation building, some scoring functions, optimization algorithms, *etc.*

main This is the folder with files giving examples of executables demonstrating different features of the MOSES system: representation building, reducing expressions, and, of course, applications of MOSES itself (for instance, there are examples for the "ant problem", for "parity formulae", *etc.*)

eda support for estimation of distribution algorithms with lower level support for optimization algorithms.

8 Key Types, Structures, and Classes

This section briefly discusses technical details about the key elements of MOSES.

8.1 Structured Expressions

For representing structured expressions (programs, propositional formulae, *etc*) MOSES relies on the library ComboReduct.

In ComboReduct, structured expressions are represented by trees of the type `combo_tree` as follows:

```
typedef Util::tree<vertex> combo_tree;
```

The file `comboreduct/combo/vertex.h` defines `vertex` as shown blow. It is done this way so that it can capture different sorts of nodes, for different, but still fixed, problem domains.

```
 typedef boost::variant<builtin, wildcard, argument,
contint, action, builtinaction, perception, definiteobject, indefiniteobject,
message, procedurecall, anntype, actionsymbol> vertex;
```

For a detailed explanation, review the docs provided in the distribution of the ComboReduct library.

[1] By "actions" we mean mini programming languages describing actions of a agents such as artificial and [6]. Available actions typically cover atomic instructions like "step forward", "rotate left", "rotate right", "step forward", branching instruction such as „if-then-else", and loop instruction such as „while".

8.2 Metapopulation

The metapopulation is a set of scored combo trees. Several types of scores are used, ranging from single floats, to composite scores that include a complexity measure, to 'behavioural' scores, consisting of vectors of floats. These are all defined in the file `learning/moses/moses/types.h`

A 'scored combo tree' is then just a pair that associates a score with a tree:

```
typedef taggeditem<combo::combotree, compositebehavioralscore>
bscoredcombotree;
```

The metapopulation is then a set of scored combo trees, defined in `learning/moses/moses/metapopulation` (more precisely, it is a class, inheriting from the set):

```
typedef std::set<bscoredcombotree, bscoredcombotreegreater>
bscoredcombotreeset;

struct metapopulation : public bscoredcombotreeset {...};
```

The metapopulation will store expressions (as scored trees) that were encountered during the learning process (not all of them; the weak ones, which are dominated by existing ones, are usually skipped as non-promising).

As an example, one can iterate through the metapopulation and print all its elements with their scores and complexities in the following way:

```
for (constiterator it=begin(); it!=end(); ++it)
cout << gettree({*}it) << \textquotedbl{} \textquotedbl{} << getscore({*}it)
<< \textquotedbl{} \textquotedbl{} << getcomplexity({*}it) << endl;
```

The metapopulation is updated in iterations. In each iteration, one of its elements is selected as an exemplar. The exemplar is then used for building a new deme (that will further extend the metapopulation).

8.3 Deme

The metapopulation consists of expressions (stored as scored trees). These expressions do not refer to some exemplar. On the other hand, during the learning process, new demes are generated. A deme is centered around an exemplar from the metapopulation. However, elements of the deme are not represented as (scored) trees but, instead, are represented relative to the exemplar. So, the deme has the type (declared in `eda/instance_set.h`):

```
eda::instanceset<treescore>
```

where `instance_set` is a set of scored instances, and `instance` is declared in `eda/eda.h` is a vector of packed knob settings:

```
typedef vector<packedt> instance;
```

The basic learning and optimization processes work over these instances, and not over trees. So, a suitable way for representing expressions/trees as sequences of bit/integer/real values (relative to the exemplar) enable abstract and clean optimization steps, uniform for different domains. This representation is based on representation building relative to a given exemplar.

8.4 Representation

A structure describing a representation of a deme (relative to its exemplar) is declared in `moses/representation.h`. Its constructor uses as arguments a simplification rule, an exemplar, a type tree, a pseudo-number generator and, optionally, just for actions (like for the ant problem), sets of available operators, perceptions, and actions).

The constructor of this structure, builds knobs with respect to the given exemplar (by the method `build_knobs`).

This structure stores the exemplar like a tree (more precisely `vtree`). This structure has a method for using a given instance to transform the exemplar (`transform`) providing a new expression tree.

The structure also has methods for clearing the current version of the exemplar (setting all knobs to default values — zeros) — `clear_exemplar`, for getting the exemplar — `get_clean_exemplar`, and for getting the reduced, simplified version of the exemplar `get_clean_exemplar`.

9 Key Methods

This section briefly discusses key MOSES's methods.

9.1 Representation Building

Representation building is one of the key aspects of the MOSES approach. It is implemented separately for different problem domains (propositional formulae, action, etc). Support for several problem domains is given in the file `moses/build_knobs.h/cc`. Representation building starts with an exemplar and add to it new nodes and corresponding knobs (see [6]). Knobs can have different settings. If all knobs are set to 0, then we the original exemplar is obtained.

There are several types of knobs, described in `moses/knobs.h`. Some of them are suitable for propositional formulae (so some subexpression can be *present*, *absent*, or *negated*. In simple knobs, a subexpression can be just *present* or *absent*. In action knobs, a node can have different settings, corresponding to atomic or compound actions, sampled as "perms" in the method `build_knobs::sample_action_perms`.

The key method is `void build_knobs::build_action(pre_it it)` — it substantially determines the representation building for one domain and substantially influence the learning process.

9.2 Optimization

The role of optimization is to score elements of the deme, and to process them and to generate new promising instances. The optimization process works over sequences of numbers (i.e., over "instances"). It is invoked in the main procedure, in each iteration of expanding (the method `metapopulation::expand`, implemented in the file `moses/moses.h`) the metapopulation. The result is a set of instances that are transformed to trees and then added to the metapopulation (if they are not dominated by existing elements).

Currently, there are two optimization methods implemented (implemented in the file `moses/optimization.h`):

univariate optimization based on Bayesian optimization (see [6]);

iterative hillclimbing based on a simple greedy iterative process, looking for a better than exemplar instance at distance $1, 2, 3, \ldots$;

In addition, there is support for "sliced iterative hillclimbing", similar as the one as above, but using a time slicing, so it can be used in some wider context, without leaving other subtasks idle for too long.

Optimization algorithms also have to take care of number of evaluations (number of calls to scoring functions) used. Basically, this number controls the resources given to the algorithm.

9.3 Scoring

While the representation building is specific for each problem domain, the scoring is specific for each specific problem. For instance, in learning propositional formulae, the same representation building algorithm is used, but different scoring functions will be used for each specific task (for instance, for learning disjunction or conjunction over the given set of propositional variables).

There is a

score function returning `int`, generally with 0 as a perfect score, and negative numbers as worse scores;

behavioral score function returning a vector of specific values, that is used for comparing expressions on different dimensions and for discarding elements dominated by other elements.

Technically, these functions are provided as operators within structures.

The scoring functions are used for instantiating higher-level functions used uniformly for different problem domains and different problems.

Some low-level, problem specific, scoring functions are defined in the file (under `src/MosesEda`) `moses/scoring_functions.h`, while higher-level support is defined in the file `moses/scoring.h/cc`.

10 MOSES: Putting It All Together

With all components briefly described above, this section discusses how are they combined in a system MOSES.

The main moses method is trivial: it only expand the metapopulation in iterations until the given number of evaluations or a perfect solution is reached. This method is implemented in `moses/moses.h`, in several variations (some with additional arguments corresponding to available actions and perceptions, just for the action problem domain).

Typical usage of MOSES starts by providing scoring functions. For instance, for learning disjunction propositional formula one can use the following declaration (defined in `moses/scoring_functions.h`):

```
disjunction scorer;
```

and for solving the ant problem, one can use the following declaration (defined in `moses/scoring_functions.h`):

```
antscore scorer;
```

Also, the type of expression to be learnt has to be provided [2]. For instance, for the disjunctive formula, one should use:

```
typetree tt(id::lambdatype); tt.appendchildren(tt.begin(),id::booleantype,arity+1);
```

where `arity` carries the information of the number of propositional variables to be considered. For the ant problem, one would write:

```
typetree tt(id::lambdatype); tt.appendchildren(tt.begin(),id::actionresulttype,1);
```

Then the metapopulation has to be declared. It is instantiated via templates, saying which scoring function, which behavioral scoring function, and which optimization algorithm to use. As, arguments one has to provide the random generator, the initial exemplar, the type tree, simplification procedure, then the scorers and the optimization algorithm. This is an example for learning the disjunctive formula:

```
metapopulation<logicalscore,logicalbscore,univariateoptimization>
metapop(rng, vtree(id::logicaland),tt,logicalreduction(), logicalscore(scorer,arity,rng),
logicalbscore(scorer,arity,rng), univariateoptimization(rng));
```

and this is an example for the ant problem:

```
metapopulation<antscore,antbscore,univariateoptimization>
metapop(rng,vtree(id::sequentialand),tt,actionreduction(), scorer,
bscorer, univariateoptimization(rng));
```

[2] for a detail explanation of the type system used in ComboReduct see the doc provided with the distribution of ComboReduct

There is also a version of MOSES that uses the sliced interactive hillclimbing — `sliced_moses`. It supports the action domain, but can be simply modify to support other domains as well.

These, and several more examples, can be found in the folder `main`. The example programs provided often ask for arguments like the seed for the pseudo-random number generation or for the number of evaluations.

11 Final Remarks

MOSES is a rather big system and it cannot be documented in details in a few pages. However, the descriptions given above should be helpful when one first encounters MOSES and tries to use it and modify it.

References

[1] Moshe Looks, "Scalable Estimation-of-Distribution Program Evolution", Genetic and Evolutionary Computation COnference (GECCO), 2007.

[2] Moshe Looks, "On the Behavioral Diversity of Random Programs", Genetic and Evolutionary Computation COnference (GECCO), 2007.

[3] Moshe Looks, "Meta-Optimizing Semantic Evolutionary Search", Genetic and Evolutionary Computation COnference (GECCO), 2007.

[4] Moshe Looks, Ben Goertzel, Lucio de Souza Coelho, Mauricio Mudado, and Cassio Pennachin,"Clustering Gene Expression Data via Mining Ensembles of Classification Rules Evolved Using MOSES", Genetic and Evolutionary Computation COnference (GECCO), 2007.

[5] Moshe Looks, Ben Goertzel, Lucio de Souza Coelho, Mauricio Mudado, and Cassio Pennachin, "Understanding Microarray Data through Applying Competent Program Evolution", Genetic and Evolutionary Computation COnference (GECCO), 2007.

[6] Moshe Looks, "Competent Program Evolution" Doctoral Dissertation, Washington University in St. Louis, 2006.

[7] Moshe Looks, "Program Evolution for General Intelligence", Artificial General Intelligence Research Institute Workshop (AGIRI), 2006.