Symbiotic/Competitive Coevolutionary Genetic Programming: A Guide to the SCM Implementation

John A. Doucette (jdoucett@cs.dal.ca)

January 20, 2010

1 What is...

1.1 This Guide

This document is intended to make the job of implementing a problem for the Symbiotic/Competitive Coevolutionary Model of Genetic Programming (SCM GP) code base easier. It describes the process required to implement a new problem and existing problem types that may be easily extended.

1.2 SCM GP

SCM GP is an implementation of the symbiotic/competitive coevolutionary bid-based genetic programming algorithm known as SBB. The algorithm is presented in full in [1]. Key requirements of the algorithm are

1. The problem of interest must be phrased as a mapping from a set of real-valued inputs to some finite set of outputs. The set of possible outputs is known as the "action" space, the terminology being derived from the use of the model in control problems.

2. The problem of interest should have a large number of possible data points available for study. The SBB algorithm's run time is asymptotically independent of the number of point available in total, and requires a large variety of points for best results.

If your problem does not meet these criteria, you should address the issue. A common failure of these conditions arises when the action space is non-finite, as in real-valued control problems. In these cases it is usually sufficient to descretize the action space manually. For example, a problem allowing a real-valued rotation could be limited to a subset of possible turns, perhaps allowing relative rotations from the set $\pm\{\pi/2, \pi/3, \pi/4\}$ instead.

1.3 The Code Base

The code base for SCM GP is written in gnu C++, and designed for use and compilation on *nix systems with gnu and make available. It is comprised of 15 source code files and a make file, summarized in table 1.

2 Adding a Problem

2.1 Make an Environment

The first step in implementing a new problem will be to make a new environment class. This class describes a specific type of problem space. Fortunately, many of the implementation details can be taken care of by abstracted superclasses.

2.1.1 Pick a Superclass

There are three superclasses to pick from in env.hpp: env, implicitEnv, and explicitEnv.

implicitEnv problems are those where the environment is implicit. Rather than specifying a set of data points to train or test on, you will only need to specify a method of generating data points. In domains like puzzle solving, this may be used to specify a class of puzzles, and allow random sampling from that space to be done on the fly. It is useful when the space of possible problems is large.

explicitEnv problems require a list of points to be provided for each run of the program. This gives the user more direct control over which points are used, but also limits the total number of points available, which may limit the maximum efficacy of the algorithm.

For problems which fall somewhere in the middle, or which require unique fitness measures to be used, env provide a less abstracted superclass. Implementing a problem as a descendant of env will be more work, but is often easier than trying to fit a strange problem into one of the other environment types. Sections 2.1.2-2.1.4 below detail the creation of a descendant of env. The other classes are somewhat easier to figure out, and so are not included.

2.1.2 Create a Class

Create a new class in the env.hpp file that inherits from your superclass. Create an problem specific data fields. Create method prototypes for all inherited methods and problem specific methods. See sections 2.1.3 and 2.1.4 below for more details on the method constraints.

2.1.3 Implement Problem Specific Constructor, Destructor, and Misc. Methods

Constraints:

File Name	Purpose
env.cpp/env.hpp	Implements an abstracted environment class with descendant classes. Most modifications will be made here.
learner.cpp/learner.hpp	Implements a learner class This class should _not_ be modified as part of a new problem implementation.*
main.cpp	Implements a driver program using the various objects. Small modifications may be required.
misc.cpp/misc.hpp	Misc. functions and constants used by other parts of the package. Should not need to be modified.
point.cpp/point.hpp	Implements a point class. Should _not_ be modified.*
scm.cpp/scm.hpp	Implements an SCM class. Almost all modifications not made to the env.* files will be made here.
shapes.cpp/shapes.hpp	Implements some simple geometric constructs used in some sample problems. Probably does not need to be modified.
team.cpp/team.hpp	Implements a team class. Should _not_ be modified. *
makefile	The make file.

Table 1: Breakdown of files in SCM GP. Entries denoted with a * should not be modified.

Your classes constructor must include an argument "int mstep", and be of the form:

myclass(...., int mstep, ...) : env(mstep) {

2.1.4 Implement the Abstracted Methods

The abstracted methods will be called by various components of the SCM GP system, so you should be careful when implementing them. They are:

```
double act(point *p, long int action, vector <double> &state, long step_number, bool &s
```

- The method should modify the state vector according to the action. If more actions are permitted, set end to false, otherwise set it to true. The return value should be the fraction of maximum reward obtained by taking this action. For example, in classification, actions correspond to labels, so act simply checks if p->label == action, and return 1 or 0 accordingly. Stop is immediately set to true. In control problems though, we might allow the agent to continue providing actions until step_number exceeds some threshold (often mstep).

```
double test(team* t, string prefix, boolean last)
```

-The method should evaluate t across some set of problem exemplars and return an aggregate measure of their performance. If last is true, it may print some data, or do a longer than normal evaluation. Prefix may be printed as an identifier of the particular evaluation method being used at the moment.

```
point * initUniformPoint(long gtime, set<long> usedIds)
```

-The method should return a new point sampled at random from the environment. gtime is the current time in the run (measured in terms of specific function types of function calls, not seconds). usedIds may be used to avoid creating duplicate points if necessary.

2.2 Make a Model

Now that the environment is implemented, a model needed. A model is a class that describes how to parse problem specific parameters, constructs and stores a problem specific environment object, and drives the learning process.

2.2.1 Pick a Superclass

Like with env, we can pick from scm, scmImplicit, or scmExplicit. The instructions below detail only how to inherit from scm.

2.2.2 Create a Class

Make a new class in the scm.hpp header file. It should have a data field of whatever the environment you created was called. See below for constraints and requirements on methods.

2.2.3 Implement Problem Specific Constructor, Destructor, and Misc. Methods

Constructor must have the form

myclass(map <string, string> &args) : scm(args)

-args will map the names of arguments in a parameter file to the corresponding values. The constructor should extract any problem specific parameters and pass them to it's environment variable during initialization. The constructor must set the _numActions field of the superclass to match the size of the problem specific action space. The constructor must insert pointers to PROFILE_SIZE new randomly sampled points into the superclass's _profilePoints vector.

The deconstructor is responsible for freeing the points in _profilePoints, even though they belong to the superclass.

2.2.4 Implement the Abstracted Methods

The methods are:

void initPoints()

-inserts pointers to _Psize-_Pgap new random points into the superclass's _P vector. Note that this can be done using the env's initUniformPoint method. Points should be created with a gtime of -1.

void genPoints(long gtime)

- When this method is called, _P in the superclass will have less than _Psize members. This method should add new random points until _Psize members are present. Points should be created with a gtime of gtime.

void stats(long gtime, long level)

-The method should print data to cout (optional).

void test(long level)

-The method should call the test function of env for each of the elements in the superclass's _M vector (_M contains pointers to teams). It should print a result indicating which team was the best and what it's aggregate score on the test was to cout.

2.3 Modify the Driver

The driver.cpp file reads in the parameters, determines the type of model to build (based on the "envType" parameter), and creates it. Modify this file to recognize your new problem type.

References

[1] Pitir Paper