libea v0.1.0 - README

Introduction

libea is a small template based library for writing evolutionary algorithms. It offers base classes for implementing genomes and also some genetic operators.

Building libea

libea uses GNU Make as build system. To compile the library open the ***libea*** directory in a terminal and type in the following command:

```
# make
```

There are no dependencies. If you want to compile the test suite type in:

```
# make test
```

Kindly note that this requires the CppUnit[1] framework.

To generate the code documentation you have to enter the command below:

```
# make doc
```

You need Doxygen[2] for this step.

A quick overview

All individuals are inherited from the same base class: *AGenome*. This class provides mainly methods to read and write genes.

Usually you don't have to implement a genome class on your own. If you want to store primitive datatypes like *int* or *float* you can use the *PrimitiveGenome* template. Use this class also for binary genomes.

In ***libea*** it's possible to use user-defined classes for genes. Please consider that this leads to more overhead than using primitive datatypes in general. User-defined gene classes are inherited from *AGene*. Instead of the *PrimitiveGenome* template you have to use the *Genome* template when you decide to use user-defined genomes.

libea offers three templates for implementing operators: *ASelection*, *ACrossover* and *AMutation*. There's also a specialization available for binary mutations: *ABinaryMutation*.

[1] CppUnit
[2] Doxygen

Examples

Creating primitive genomes

This example shows the basic concepts of ***libea***.

Assume our individuals consist of binary values. In this case we use the *PrimitiveGenome* base class.

```
#include <libea/PrimitiveGenome.h>

using namespace ea;

static double fitness(const PrimitiveGenome<bool>* g)
{
    double f = 0.0;

    for(uint32_t i = 0; i < g->size(); i++)
    {
        if(g->at(i))
        {
            f++;
        }
    }

    return f;
}

int main(int argc, char* argv[])
{
    PrimitiveGenome<bool> g(10, &fitness);
}
```

As you can see here a genome has also an associated fitness function. Fitness functions have always one parameter (the genome) and return a *double*.

To set genes you can use the set() method of the genome:

```
for(int i = 0; i < 10; i++)
{
    g.set(i, i % 3);
}
```

A genome can also be represented as string or hash value:

```
cout << g.to_string() << endl;
cout << g.hash() << endl;
```

This is the way you can receive the fitness of an individual:

```
cout << g.fitness() << endl;
```

libea provides a base class which can be used as standardized way to generate genomes: *AFactory*.

A factory requires an associated random number generator and provides a method named *random()* to generate genomes:

```
#include <PrimitiveGenome.h>
#include <AFactory.h>
#include <TR1UniformDistribution.h>

using namespace ea;

class TestFactory
    : public AFactory<shared_ptr<PrimitiveGenome<bool>>>
{
    public:
        using AFactory<shared_ptr<PrimitiveGenome<bool>>>::random;

        const uint32_t GENOME_LENGTH = 20;

        TestFactory(shared_ptr<ARandomNumberGenerator> rnd_generator)
            : AFactory<shared_ptr<PrimitiveGenome<bool>>>(rnd_generator) {}

    protected:
        void random_impl(const uint32_t count,
                         IOutputAdapter<shared_ptr<PrimitiveGenome<bool>>> &output)
        {
            int32_t numbers[GENOME_LENGTH];

            for(uint32_t i = 0; i < count; i++)
            {
                auto genome = make_shared<PrimitiveGenome<bool>>(GENOME_LENGTH, &fitness);

                generator->get_int32_seq(0, 1, numbers, GENOME_LENGTH);

                for(uint32_t m = 0; m < GENOME_LENGTH; ++m)
                {
                    genome->set(m, numbers[m]);
                }

                output.append(genome);
            }
```

```
        }
};

int main(int argc, char *argv[])
{
    auto g = make_shared<TR1UniformDistribution<mt19937_64>>();
    vector<shared_ptr<PrimitiveGenome<bool>>> population;
    TestFactory f(g);

    f.random(10, population);

    for_each(begin(population),
             end(population),
             [] (shared_ptr<PrimitiveGenome<bool>> genome)
             {
                cout << "genome: "
                     << genome->to_string()
                     << ", fitness: "
                     << genome->fitness() << endl;
             });
}
```

libea makes heavy usage of shared pointers to save resources. It's a good approach to generate shared pointers in your factory class because these kind of pointers are used in other functions we're going to see later in this tutorial.

Creating user-defined genomes

If you want to use user-defined genomes please use the *Genome* template. Your gene class has to be inherited from the *AGene* template.

Solving the popular traveling salesman problem you might want to use a genome where each gene is a city. In this case you could write your own *City* class:

```
#include <Genome.h>

using namespace ea;

class City : public AGene<City>
{
    public:
        City(const string name, const int32_t x, const int32_t y)
            : _name(name), _x(x), _y(y) {}

        void set_name(const string name)
```

```
{
    _name = name;
    modified();
}

string get_name() const
{
    return _name;
}

void set_x(const int32_t x)
{
    _x = x;
    modified();
}

int32_t get_x() const
{
    return _x;
}

void set_y(const int32_t y)
{
    _y = y;
    modified();
}

int32_t get_y() const
{
    return _y;
}

size_t hash() override
{
    SDBMHash hasher;

    hasher << _x << _y;

    return hasher.hash();
}

bool less_than(City* gene) override
{
    return sqrt(pow(_x, 2) + pow(_y, 2)) < sqrt(pow(gene->_x, 2) + pow(gene->_y, 2))
}
```

```
        City* clone() const override
        {
            return new City(_name, _x, _y);
        }

        string to_string() const override
        {
            return _name;
        }

    private:
        string _name;
        int32_t _x;
        int32_t _y;
};
```

Looking at this code you might see one important aspect: after modifying a property of the *City* class we call the *modified()* method. This is quite important because the *Genome* class has an internal cache. The *modified()* method clears this cache.

Classes inherited from *AGene* have to override a few methods. One of these methods is the *hash()* method. If you're going to write your own gene class the *SDBMHash* class might be quite useful for you. Looking at the documentation/source code of this class you'll notice also that it's very easy to write your own hash classes.

Crossover operators

As described before ***libea*** provides various operators. To combine genomes and create new children we use crossover operators.

Crossover operators are inherited from the *ACrossover* template. Let's have a look at an example:

```
auto g = make_shared<TR1UniformDistribution<mt19937_64>>();

vector<shared_ptr<PrimitiveGenome<bool>>> population;
vector<shared_ptr<PrimitiveGenome<bool>>> children;

TestFactory f(g);

f.random(10, population);

for_each(begin(population), end(population), [] (shared_ptr<PrimitiveGenome<bool>> genome)
{
```

```
    cout << "genome: "
         << genome->to_string()
         << ", fitness: "
         << genome->fitness()
         << endl;
});

OnePointCrossover<PrimitiveGenome<bool>> c(g); // OnePointCrossover.h

c.crossover(begin(population), end(population), children);

cout << endl;

for_each(begin(children), end(children), [] (shared_ptr<PrimitiveGenome<bool>> genome)
{
    cout << "genome: "
         << genome->to_string()
         << ", fitness: "
         << genome->fitness()
         << endl;
});
```

In this example we use again a binary genome and the *TestFactory* class we've implemented before. After generating a population we create an *OnePointCrossover* instance and combine all genomes of the source population. The generated children are written to the *children* vector.

It's not difficult to create your own crossover operator. Here's a template:

```
#include <ACrossover.h>

template<typename TGenome>
class MyCrossover : public ACrossover<TGenome>
{
    public:
        using ACrossover<TGenome>::crossover;

        MyCrossover(std::shared_ptr<ARandomNumberGenerator> rnd_generator)
            : ACrossover<TGenome>(rnd_generator) {}

        virtual ~MyCrossover() {};

    protected:
        uint32_t crossover_impl(const std::shared_ptr<TGenome> &a,
                                const std::shared_ptr<TGenome> &b,
                                IOutputAdapter<std::shared_ptr<TGenome>> &output) override
```

```
        {
            std::shared_ptr<TGenome> child;

            // TODO: generate child

            output.append(child);

            return 1;
        }
};
```

You only have to override the *crossover_impl()* method which receives two genomes. Generate some child genomes and append them to the given adapter. The return value is the number of generated children.

If you're going to copy genes from one genome to another ***never*** use the *set()* method of the destination genome because this may lead to a memory leak. Please use the *copy_to()* method instead. It frees resources and clone genes correctly. Here's an example how to copy genes properly:

```
template<typename TGenome>
void copy(const TGenome& src, TGenome& dst)
{
    assert(src->size() == dst->size());

    for(uint32_t i = 0; i < src->size(); i++)
    {
        dst->copy_to(i, src->at(i));
    }
}
```

Mutation

Without mutation evolution is quite boring. Therefore ***libea*** provides some mutation operators.

Once again we use a binary genome for this example:

```
auto g = make_shared<TR1UniformDistribution<mt19937_64>>();

vector<shared_ptr<PrimitiveGenome<bool>>> population;

TestFactory f(g);

f.random(10, population);
```

```
for_each(begin(population), end(population), [] (shared_ptr<PrimitiveGenome<bool>> genome)
{
    cout << "genome: "
         << genome->to_string()
         << ", fitness: "
         << genome->fitness()
         << endl;
});

SingleBitStringMutation m(g); // SingleBitStringMutation.h

m.mutate(begin(population), end(population));

cout << endl;

for_each(begin(population), end(population), [] (shared_ptr<PrimitiveGenome<bool>> genome)
{
    cout << "genome: "
         << genome->to_string()
         << ", fitness: "
         << genome->fitness()
         << endl;
});
```

The genomes are mutated in-place, but it's also possible to mutate a single genome and copy the result to a different location:

```
auto g = make_shared<TR1UniformDistribution<mt19937_64>>();

auto g0 = make_shared<PrimitiveGenome<bool>>(10, &fitness);
auto g1 = make_shared<PrimitiveGenome<bool>>(10, &fitness);

g0->parse_string("0010111011");

InverseBitStringMutation m(g); // InverseBitStringMutation.h

m.mutate(g0, g1);

cout << "g0: " << g0->to_string() << endl;
cout << "g1: " << g1->to_string() << endl;
```

Selection

Last but not least there are selection operators in ***libea***. Here's an example where we select 10 genomes from a larger population:

```
auto g = make_shared<TR1UniformDistribution<mt19937_64>>();

vector<shared_ptr<PrimitiveGenome<bool>>> population;
vector<shared_ptr<PrimitiveGenome<bool>>> children;

TestFactory f(g);

f.random(100, population);

FitnessProportionalSelection<PrimitiveGenome<bool>> sel(g);
// FitnessProportionalSelection.h

sel.select(population, 10, children);

for_each(begin(children), end(children), [] (shared_ptr<PrimitiveGenome<bool>> genome)
{
    cout << "genome: "
         << genome->to_string()
         << ", fitness: "
         << genome->fitness()
         << endl;
});
```

Putting it all together

As you have seen in this tutorial ***libea*** doesn't provide any ready-to-use evolutionary algorithms but a basic framework which may help you writing your own algorithms. There are some standard operators available which can be used for different kind of genomes, too.

This library has been written just for fun and I hope you might find it useful. To have a more detailed look at ***libea*** you should generate the code documentation as described before in this tutorial.