libea 0.2 - A short introduction

Sebastian Fedrau

November, 2014

1 Introduction

LIBEA is a template based library written in C++14. The purpose of this software is to provide an extensible and reliable framework for writing evolutionary algorithms.

2 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
```

The test suite can be compiled with the command below (kindly note that this requires the CppUnit[1] framework).

```
# make test
```

The source code documentation can be generated with the make utility too. You need Doxygen[2] for this step.

```
# make doc
```

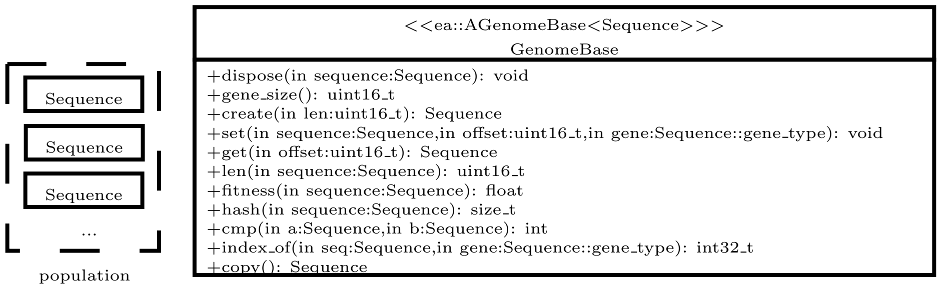
3 The framework

3.1 A quick overview

In LIBEA individuals are represented in *sequences*. Theoretically any data type large enough to store the genotype of an individual could be used as sequence type. A group of sequences is called *population*.

To modify a sequence or evaluate its fitness a corresponding *genome base class* inherited from *ea::AGenomeBase* is required. Figure 1 illustrates this concept.

Figure 1: sequences and corresponding genome base class



Sequence
Sequence
Sequence
...
population
<<ea::AGenomeBase<Sequence>>>
GenomeBase
+dispose(in sequence:Sequence): void
+gene_size(): uint16_t
+create(in len:uint16_t): Sequence
+set(in sequence:Sequence,in offset:uint16_t,in gene:Sequence::gene_type): void
+get(in offset:uint16_t): Sequence
+len(in sequence:Sequence): uint16_t
+fitness(in sequence:Sequence): float
+hash(in sequence:Sequence): size_t
+cmp(in a:Sequence,in b:Sequence): int
+index_of(in seq:Sequence,in gene:Sequence::gene_type): int32_t
+copy(): Sequence

LIBEA offers the genome base class templates *ea::PGenomeBase* and *ea::CPGenomeBase* with the corresponding sequence types *ea::Sequence* and *ea::CSequence*. Any data

[1]`http://freedesktop.org/wiki/Software/cppunit/`
[2]`http://www.stack.nl/~dimitri/doxygen/`

type with a valid copy constructor is supported as gene type. There are already type aliases for many primitive data types available in LIBEA (see table 1).

GENOME BASE CLASS	CORRESPONDING SEQUENCE TYPE	GENE TYPE
ea::PGenomeBase<T>	ea::Sequence<T>	T
ea::Int32PenomeBase	ea::Sequence<int32_t>	int32_t
ea::UInt32PGenomeBase	ea::Sequence<uint32_t>	uint32_t
ea::BinaryPGenomeBase	ea::Sequence<bool>	bool
ea::DoublePGenomeBase	ea::Sequence<double>	double
ea::StringPGenomeBase	ea::Sequence<std::string>	std::string
ea::CPGenomeBase<T>	ea::CSequence<T>	T
ea::Int32CPenomeBase	ea::CSequence<int32_t>	int32_t
ea::UInt32CPGenomeBase	ea::CSequence<uint32_t>	uint32_t
ea::BinaryCPGenomeBase	ea::CSequence<bool>	bool
ea::DoubleCPGenomeBase	ea::CSequence<double>	double
ea::StringCPGenomeBase	ea::CSequence<std::string>	std::string

Table 1: genome base classes and corresponding sequence types offered by LIBEA

The two offered templates are optimized for different use-cases. As shown in listing 1 the *ea::Sequence* structure has only two fields which are used to store the length of a sequence and its genes. Hence this kind of sequence is preferable if the application's memory footprint is critical.

In comparison the *ea::CSequence* structure has three additional fields. They are used to cache evaluated fitness values and the hash of an individual (see listing 2). In general this sequence type is a better choice because the performance benefits might out weight the drawbacks of memory consumption.

The fields of both structures should **never be accessed directly**. LIBEA provides the functions *ea::sequence_len()* and *ea::sequence_get()* for reading genes from a sequence.

Listing 1: Sequence template declaration

```
template<typename TGene>
struct Sequence
{
  /*! Datatype of stored genes. */
  typedef TGene gene_type;

  /*! Number of stored genes. */
  uint16_t len;
  /*! Dynamic array holding genes. */
  TGene* genes;
};
```

Listing 2: CSequence template declaration

```
template<typename TGene>
struct CSequence : Sequence<TGene>
{
  /*! Flags. */
  uint8_t flags;
  /*! Cached fitness value. */
  float fitness;
  /*! Cached hash value. */
  size_t hash;
};
```

3.2 Core features

3.3 Working with sequences

Listing 3 demonstrates some basic functions of LIBEA. An individual represented in a sequence of integers is created (*ll. 44-50*) and copied (*l. 62*). The sequences are compared before and after modifying the second individual (*ll. 67-75*).

To access the genes and evaluate the fitness a genome base class is declared (*l. 22*). The template parameters specify gene type and fitness function (*ll. 5-19*).

Listing 3: core functions

```
 1 #include <iostream>
 2 #include <libea.hpp>
 3
 4 // fitness function:
 5 class Fitness
 6 {
 7   public:
 8     float operator()(const ea::Sequence<int32_t>* const &seq) const
 9     {
10       float avg = 0;
11
12       for(auto i = 0; i < ea::sequence_len(seq); i++)
13       {
14         avg += ea::sequence_get(seq, i);
15       }
16
17       return avg / ea::sequence_len(seq);
18     }
19 };
20
21 // base class for modifying/accessing sequences:
22 static ea::PGenomeBase<int32_t, Fitness> base;
23
24 // print a sequence:
25 static void print_genome(ea::Sequence<int32_t>* seq)
26 {
27   // print length, hash & fitness:
28   std::cout << "len:␣" << base.len(seq) << ",␣hash:␣" << base.hash(seq) << ",␣fitness:␣" << base.fitness(seq) << std::endl;
29
30   // print sequence:
31   std::cout << "sequence:␣";
32
33   for(uint32_t i = 0; i < 10; i++)
```

```
  {
    std::cout << base.get(seq, i) << "␣";
  }

  std::cout << std::endl;
}

int main(int argc, char *argv[])
{
  // create new sequence:
  ea::Sequence<int32_t>* a = base.create(10);

  // set genes:
  for(uint32_t i = 0; i < 10; i++)
  {
    base.set(a, i, i + 100);
  }

  std::cout << "genome␣a:" << std::endl;
  print_genome(a);

  // show index of genes:
  for (uint32_t g = 100; g <= 110; g++)
  {
    std::cout << "index␣of␣" << g << ":␣" << base.index_of(a, g) << std::endl;
  }

  // copy sequence:
  ea::Sequence<int32_t>* b = base.copy(a);
  std::cout << "genome␣b:" << std::endl;
  print_genome(b);

  // compare sequences:
  std::cout << "compare:␣" << base.cmp(a, b) << std::endl;

  // change single gene:
  base.set(b, 5, 42);
  std::cout << "genome␣b:" << std::endl;
  print_genome(b);

  // compare sequences:
  std::cout << "compare:␣" << base.cmp(a, b) << std::endl;

  // free memory:
  base.dispose(a);
  base.dispose(b);

  return 0;
}
```

3.4 Factories & output adapters

LIBEA offers the *ea::AFactory* base class as a unified way for creating populations. Listing 4 shows how to create sequences of ten random integers using a custom factory (*ll. 30-51*). The values are generated with a random number generator (*ll. 35-46*).

The individuals are written to an *ea::OutputAdapter* (*ll. 74-78*). At the current stage LIBEA only provides an adapter for STL vectors, but new adapters can be added easily by implementing the *ea::IOutputAdapter* interface.

Listing 4: custom factory

```
1  #include <stdint.h>
2  #include <iostream>
3  #include <vector>
4  #include <libea.hpp>
5
6  // store genomes in sequences of int32_t, this typedef makes
       the code more readable:
7  typedef ea::Sequence<int32_t> Sequence;
8
9  // fitness function:
10 class Fitness
11 {
12   public:
13     float operator()(const Sequence* const &seq) const
14     {
15       float avg = 0;
16
17       for(auto i = 0; i < ea::sequence_len(seq); i++)
18       {
19         avg += ea::sequence_get(seq, i);
20       }
21
22       return avg / ea::sequence_len(seq);
23     }
24 };
25
26 // base class for modifying/accessing sequences:
27 static ea::PGenomeBase<int32_t, Fitness> base;
28
29 // factory class:
30 class Factory : public ea::AFactory<Sequence*>
31 {
32   public:
33     Sequence* create_sequence()
34     {
35       int32_t genes[10];
36
37       _g.get_int32_seq(1, 100, genes, 10);
38
39       Sequence* seq = base.create(10);
40
41       for(uint32_t i = 0; i < 10; i++)
42       {
```

```cpp
        base.set(seq, i, genes[i]);
      }

      return seq;
    }

  private:
    static ea::AnsiRandomNumberGenerator _g;
};

ea::AnsiRandomNumberGenerator Factory::_g;

// debug function:
static void print_genome(Sequence* seq)
{
  std::cout << "hash: " << base.hash(seq) << ", fitness: " << base.fitness(seq) << ", sequence: ";

  for(uint32_t i = 0; i < 10; i++)
  {
    std::cout << base.get(seq, i) << " ";
  }

  std::cout << std::endl;
}

int main(int argc, char *argv[])
{
  Factory f;
  std::vector<Sequence*> population;

  // create adapter from back_inserter:
  auto inserter = std::back_inserter(population);
  ea::STLVectorAdapter<Sequence*> adapter(inserter);

  // create 100 genomes:
  f.create_population(100, adapter);

  // print genomes & free memory:
  std::for_each(begin(population), end(population), [](Sequence* seq)
  {
    print_genome(seq);
  });

  ea::dispose(base, begin(population), end(population));

  return 0;
}
```

3.5 Selection operators

Selection operators have to implement the *ea::IIndexSelection* interface (see listing 5). Instead of copying the selected genomes to an *ea::OutpuAdapter* only the indexes are written. Access to the parent individuals is given through an adapter implementing the *ea::IInputAdapter* interface. Input adapters can be created easily for all STL containers providing random access iterators with the *ea::make_input_adapter()* function.

Listing 5: IIndexSelection

```cpp
template<typename TGenomeBase>
class IIndexSelection
{
  public:
    typedef typename TGenomeBase::sequence_type sequence_type;

    ~IIndexSelection() {}

    virtual void select(IInputAdapter<sequence_type>& input, const uint32_t count, IOutputAdapter<uint32_t>& output) = 0;
};
```

Listing 6 creates a population with ten random sequences (*ll. 43-58*). Then five individuals are selected using a tournament selection operator (*ll. 72-83*).

Listing 6: selection operators

```cpp
1  #include <stdint.h>
2  #include <iostream>
3  #include <vector>
4  #include <libea.hpp>
5
6  // store genomes in sequences of int32_t, this typedef makes the code more readable:
7  typedef ea::Sequence<int32_t> Sequence;
8
9  // fitness function:
10 class Fitness
11 {
12   public:
13     float operator()(const Sequence* const &seq) const
14     {
15       float avg = 0;
16
17       for(auto i = 0; i < ea::sequence_len(seq); i++)
18       {
19         avg += ea::sequence_get(seq, i);
20       }
21
22       return avg / ea::sequence_len(seq);
23     }
24 };
25
26 static ea::Int32PGenomeBase<Fitness> base;
```

```

static void print_genome(Sequence* seq)
{
  for(uint32_t i = 0; i < ea::sequence_len(seq); i++)
  {
    std::cout << base.get(seq, i) << " ";
  }

  std::cout << std::endl;
}

int main(int argc, char *argv[])
{
  ea::AnsiRandomNumberGenerator g;

  // create parent individuals:
  std::vector<Sequence*> population;

  for(uint32_t i = 0; i < 10; i++)
  {
    auto seq = base.create(10);

    int32_t genes[10];
    g.get_unique_int32_seq(0, 9, genes, 10);

    for(uint32_t j = 0; j < 10; j++)
    {
      base.set(seq, j, genes[j]);
    }

    population.push_back(seq);
  }

  // create input adapter:
  auto input = ea::make_input_adapter(population);

  // print parent individuals:
  std::for_each(begin(population), end(population), [](
      Sequence* seq)
  {
    print_genome(seq);
  });

  std::cout << std::endl;

  // select & print individuals:
  std::vector<uint32_t> children;
  auto inserter = std::back_inserter(children);
  ea::STLVectorAdapter<uint32_t> output(inserter);

  ea::TournamentSelection<ea::Int32PGenomeBase<Fitness>> sel;
  sel.select(input, 5, output);

  std::for_each(begin(children), end(children), [&population](uint32_t index)
  {
    std::cout << "selected child: " << index << " ==> ";
    print_genome(population[index]);
  });

  // cleanup:
  ea::dispose(base, begin(population), end(population));

  return 0;
}
```

3.6 Crossover operators

Crossover operators are inherited from *ea::ACrossover* and have to override the *crossover()* method (see listing 7). In the example two parent individuals are created (*ll. 41-42*). Then two children are generated using a cycle crossover operator (*ll. 51-56*).

Listing 7: ACrossover

```
template<typename TGenomeBase>
class ACrossover
{
  public:
    typedef typename TGenomeBase::sequence_type sequence_type;

    ~ACrossover() {}

    virtual uint32_t crossover(const sequence_type& a, const sequence_type& b, IOutputAdapter<sequence_type>& output) = 0;
};
```

Crossover operators create at least one child sequence from two parent individuals. The *crossover()* method returns the number of child sequences written to the given output adapter. Listing 8 shows an example.

Listing 8: crossover operators

```
1  #include <stdint.h>
2  #include <iostream>
3  #include <vector>
4  #include <libea.hpp>
5
6  // store genomes in sequences of int32_t, this typedef makes the code more readable:
7  typedef ea::Sequence<int32_t> Sequence;
8
9  // fitness function:
10 class Fitness
11 {
12   public:
13     float operator()(const Sequence* const &seq) const
```

```
    {
      float avg = 0;

      for(auto i = 0; i < ea::sequence_len(seq); i++)
      {
        avg += ea::sequence_get(seq, i);
      }

      return avg / ea::sequence_len(seq);
    }
};

static ea::Int32PGenomeBase<Fitness> base;

static void print_genome(Sequence* seq)
{
  for(uint32_t i = 0; i < ea::sequence_len(seq); i++)
  {
    std::cout << base.get(seq, i) << " ";
  }

  std::cout << std::endl;
}

int main(int argc, char *argv[])
{
  // create parent individuals:
  auto a = base.create(10);
  auto b = base.create(10);

  for(uint32_t i = 0; i < 10; i++)
  {
    base.set(a, i, i);
    base.set(b, i, 9 - i);
  }

  // create children:
  std::vector<Sequence*> children;
  auto inserter = std::back_inserter(children);
  ea::STLVectorAdapter<Sequence*> adapter(inserter);
  ea::CycleCrossover<ea::Int32PGenomeBase<Fitness>> crossover;

  uint32_t n = crossover.crossover(a, b, adapter);

  std::cout << "number of created children: " << n << std::
      endl;

  std::for_each(begin(children), end(children), [](Sequence*
      seq)
  {
    print_genome(seq);
  });

  // cleanup:
```

```
  base.dispose(a);
  base.dispose(b);
  ea::dispose(base, begin(children), end(children));

  return 0;
}
```

3.7 Mutation operators

Mutation operators are inherited from *ea::AMutation* and have to override the *create_child()* method (see listing 9).

Listing 9: AMutation

```
template<class TGenomeBase>
class AMutation
{
  public:
    typedef typename TGenomeBase::sequence_type sequence_type;

    virtual ~AMutation() {};

    virtual void mutate(sequence_type& sequence) = 0;

    sequence_type create_child(const sequence_type& sequence)
    {
      static TGenomeBase base;

      sequence_type m = base.copy(sequence);
      mutate(m);

      return m;
    }
};
```

In listing 10 a parent individual is created (*ll. 43-48*) and mutated in-place (*l. 55*). Afterwards an inverted child sequence is generated (*ll. 59-61*).

Listing 10: mutation operators

```
 1  #include <iostream>
 2  #include <libea.hpp>
 3
 4  // store genomes in sequences of int32_t, this typedef makes
        the code more readable:
 5  typedef ea::Sequence<bool> Sequence;
 6
 7  // fitness function:
 8  class Fitness
 9  {
10    public:
11      float operator()(const Sequence* const &seq) const
12      {
13        float count = 0;
14
```

```
      for(auto i = 0; i < ea::sequence_len(seq); i++)
      {
        if(ea::sequence_get(seq, i))
        {
          count++;
        }
      }

      return count;
    }
};

typedef ea::BinaryPGenomeBase<Fitness> Base;
static ea::BinaryPGenomeBase<Fitness> base;

static void print_genome(Sequence* seq)
{
  for(uint32_t i = 0; i < ea::sequence_len(seq); i++)
  {
    std::cout << (base.get(seq, i) ? 1 : 0);
  }

  std::cout << std::endl;
}

int main(int argc, char *argv[])
{
  // create parent individual:
  auto a = base.create(10);

  for(uint32_t i = 0; i < 10; i++)
  {
    base.set(a, i, i % 3);
  }

  print_genome(a);

  // inplace mutation:
  ea::BitStringMutation<Base> muta;

  muta.mutate(a);
  print_genome(a);

  // create child:
  ea::InverseBitStringMutation<Base> mutb;

  auto b = mutb.create_child(a);
  print_genome(b);

  // cleanup:
  base.dispose(a);
  base.dispose(b);

  return 0;
```

```
}
```

4 Summary

As you have seen in this tutorial LIBEA provides a basic framework which may help you writing your own algorithms. There are some standard operators available which can be used with different kind of genomes, too.

The 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.