libea 0.2 - A brief introduction

Sebastian Fedrau

February, 2015

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

To build LIBEA scons[1] is required. The library can be compiled and installed with the following commands:

```
# scons libea
# sudo scons install
```

After the installation you may want to build the test suite. Please note that this step requires the CppUnit[2] framework:

```
# scons test-suite
```

If you want to generate the source code documentation please ensure Doxygen[3] is installed on your system and type in the following command:

```
# scons doc
```

This documentation can be build with

```
# scons pdf
```

If you want to uninstall libea type in

```
# sudo scons install -c
```

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 is a valid 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.

LIBEA offers the genome base class templates *ea::PGenomeBase* and *ea::CPGenomeBase* with the corresponding sequence types *ea::Sequence* and *ea::CSequence*. Any data type with a valid copy constructor is a supported gene type. There are already type aliases for many primitive data types available in LIBEA (see table 1).

The two offered base class 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.

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

Figure 1: sequences and corresponding genome base class

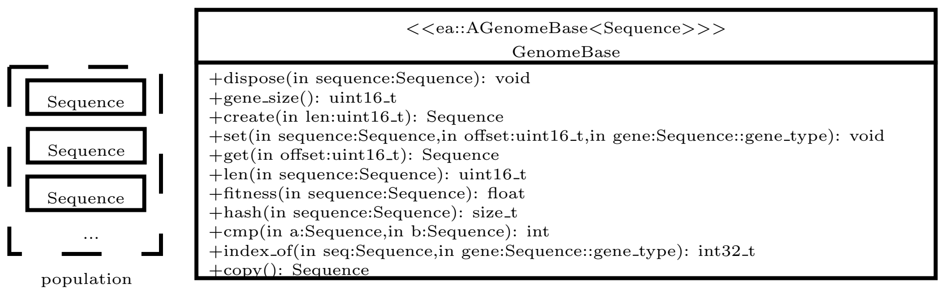

Sequence
Sequence
Sequence
...
population
<<ea::AGenomeBase<Sequence>>>
GenomeBase
+dispose(in sequence:Sequence): void
+gene_size(): uint16_t
+create(in len:uint16_t): Sequence
+set(in sequence:Sequence,in offset:uint16_t,in gene:Sequence::gene_type): void
+get(in offset:uint16_t): Sequence
+len(in 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

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

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

In comparison the *ea::CSequence* structure has three additional fields. They are used to cache the evaluated fitness value and the hash of an individual (see listing 2). In general this sequence type is a better choice because the performance benefit might out weight the drawback 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 the 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(ea::sequence_len_t 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  {
```

```cpp
  // print length, hash & fitness:
  std::cout << "len: " << base.len(seq) << ", hash: " << base.hash(seq) << ", fitness: " << base.fitness(seq) << std::endl;

  // print sequence:
  std::cout << "sequence: ";

  for(ea::sequence_len_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(ea::sequence_len_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:
```

```
  ea::dispose(base, { a, b });

  return 0;
}
```

3.4 Factories & output adapters

LIBEA offers the *ea::AFactory* base class as an option to create 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. 74-77). At the current stage LIBEA only provides an adapter for STL containers like *std::vector*, but new adapters can be added easily by implementing the *ea::IOutputAdapter* interface. To create an adapter use the *ea::make_output_adapter()* function.

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(ea::sequence_len_t 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()
```

```
    {
      int32_t genes[10];

      _g.get_int32_seq(1, 100, genes, 10);

      Sequence* seq = base.create(10);

      for(ea::sequence_len_t i = 0; i < 10; ++i)
      {
        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(ea::sequence_len_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:
  auto adapter = ea::make_output_adapter(population);

  // 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::OutputAdapter* 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 for all STL containers providing random access iterators with the *ea::make_input_adapter()* function.

Listing 5: IIndexSelection

```
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 the tournament selection operator (ll. 72-82).

Listing 6: selection 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
14     {
15       float avg = 0;
16
17       for(ea::sequence_len_t i = 0; i < ea::sequence_len(seq); ++i)
18       {
19         avg += ea::sequence_get(seq, i);
20       }
```

```

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

static ea::Int32PGenomeBase<Fitness> base;

static void print_genome(Sequence* seq)
{
  for(ea::sequence_len_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(ea::sequence_len_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(ea::sequence_len_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<std::size_t> children;
```

```cpp
  auto output = ea::make_output_adapter(children);

  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).

Listing 7: ACrossover

```cpp
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.

In the example two parent individuals are created (ll. 41-42). Then two children are generated using a cycle crossover operator (ll. 51-55).

Listing 8: crossover 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;
```

```cpp

// fitness function:
class Fitness
{
  public:
    float operator()(const Sequence* const &seq) const
    {
      float avg = 0;

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

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

  std::size_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:
  ea::dispose(base, { a, 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
```

```cpp
{
  public:
    float operator()(const Sequence* const &seq) const
    {
      float count = 0;

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

      return count;
    }
};

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

static void print_genome(Sequence* seq)
{
  for(ea::sequence_len_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(ea::sequence_len_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:
  ea::dispose(base, { a, b });

  return 0;
}
```

3.8 Processing operator pipelines

The *ea::Pipeline* namespace provides a framework to connect and process multiple genetic operators.

LIBEA processes operators by connecting elements to a pipeline. Pipeline elements read sequences from a source and write the processed data to a sink. Each element has to implement the *ea::Pipeline::IElelement* interface. LIBEA offers a corresponding pipeline element for all operator types.

A functor implementing the *ea::Pipeline::ITerminator* is executed after the last pipeline element has been processed to decide if the process should be repeated (see figure 2).

SelectionElement

The *ea::Pipeline::SelectionElement* pipeline element reads sequences from a source population and writes selected elements to the sink. Elements are selected by an underlying *ea::IIndexSelection* operator.

Creating a *ea::Pipeline::SelectionElemenet* instance two template parameters have to be defined: the desired selection operator and a functor inherited from *ea::Pipeline::ASelectionSize*. The functor is required to determine the number of sequences to select. LIBEA already offers two functors:

- ***ea::libea::SourceDivisor***: divides the size of the source population by template parameter N
- ***ea::libea::FixedSelectionSize***: returns the template parameter N

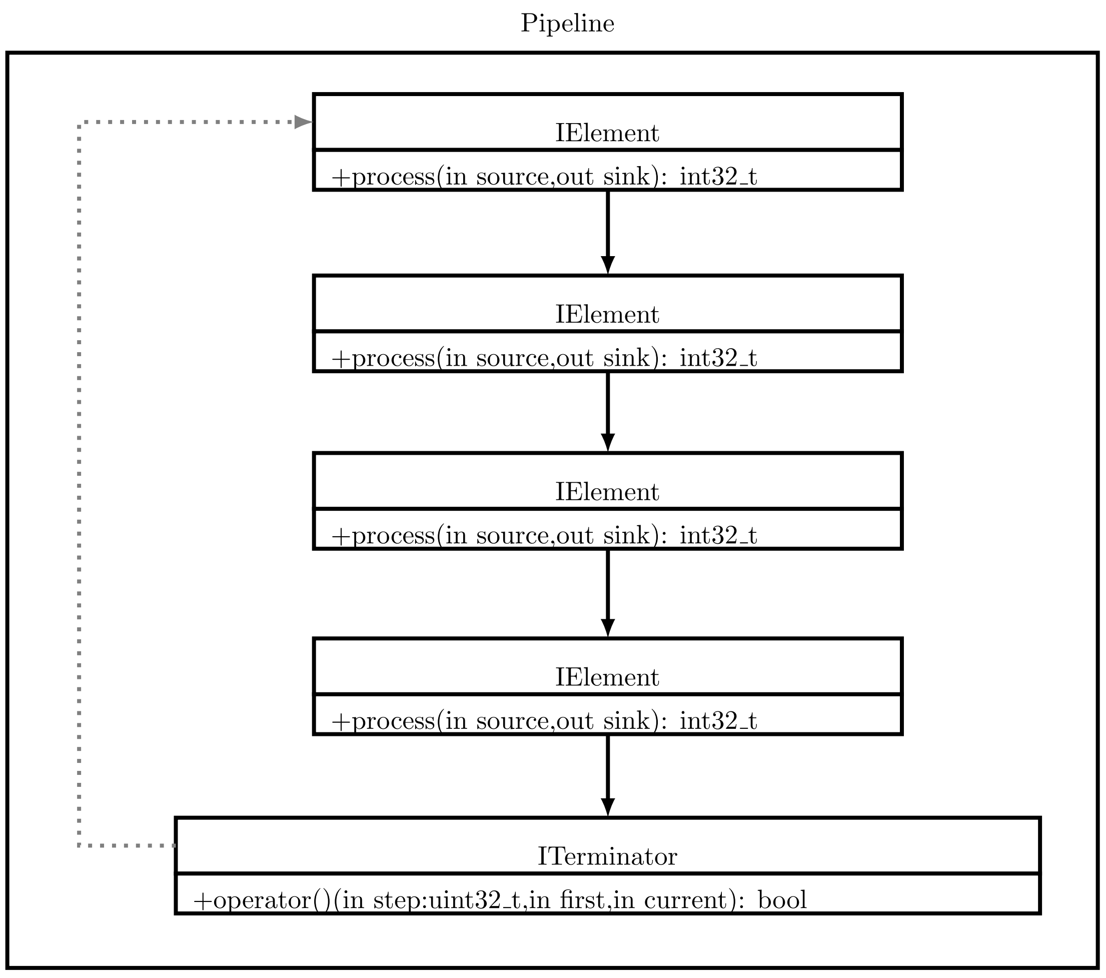
CrossoverElement

The *ea::Pipeline::CrossoverElement* pipeline element combines all sequences from a source population and produces children using the underlying *ea::ACrossover* operator. Generated children are written to the sink.

MutationElement

The *ea::Pipeline::MutationElement* reads sequences from a source population and writes them to the sink. When writing the sequences are mutated with a definable probability P using the underlying *ea::AMutation* operator.

Figure 2: connecting pipeline elements


Pipeline
IElement
+process(in source,out sink): int32_t
IElement
+process(in source,out sink): int32_t
IElement
+process(in source,out sink): int32_t
IElement
+process(in source,out sink): int32_t
ITerminator
+operator()(in step:uint32_t,in first,in current): bool

ITerminator

Terminators have to implement the *ea::Pipeline::ITerminator* interface (see listing 11). The *()* operator returns *true* if the process should be terminated and has the following parameters:

- ***step***: a counter incremented at the beginning of each iteration
- ***first***: an input adapter providing access to the *initial* population
- ***current***: an input adapter providing access to the *current* population

LIBEA offers the *ea::Pipeline::ForLoopInidicator* to repeat the pipeline process using *step* as loop counter.

Listing 11: ITerminator

```
    template<typename TGenomeBase>
    class ITerminator
    {
      public:
        virtual ~ITerminator() {}

        virtual bool operator()(const uint32_t step,
                    IInputAdapter<typename TGenomeBase::sequence_type>& first,
                    IInputAdapter<typename TGenomeBase::sequence_type>& current) = 0;
    };
```

Travelling saleman problem

Listing 12 connects four genetic operators to find solutions for the popular travelling salesman problem.

A city is identified by an *uint8_t* (l. 11). A route is a sequence of cities (l. 12). Each city has a location which can be accessed by a *std::map* (ll. 14-31). The fitness function returns the sum of the distances between the cities of a route (ll. 34-51).

The initial population consists of 100 random routes created with a factory class (ll. 57-82). The mean and median fitness values are displayed (ll. 94-102) before processing the pipeline.

The operators are connected by their corresponding pipeline elements. At the beginning of each iteration sequences are selected by tournament selection. Then the selected sequences are used to create new children using edge recombination. After selecting 50 child sequences by double tournament selection the result is mutated using single swap mutation. The process is repeated 100 times (ll. 107-126).

After processing the pipeline mean and median fitness values of the generated child population are displayed (ll. 128-131).

Listing 12: pipeline operators

```
1  #include <libea.hpp>
2  #include <vector>
3  #include <map>
4  #include <iostream>
5
6  using namespace std;
7  using namespace ea;
8  using namespace ea::Pipeline;
9
10 // type declarations:
11 typedef uint8_t City;
12 typedef CSequence<City>* Route;
13
14 typedef struct
15 {
16   uint16_t x;
```

```
  uint16_t y;
} Point;

// city map:
static map<City, Point> Cities = {
  { 0, { 31, 9 } },
  { 1, { 17, 82 } },
  { 2, { 56, 47 } },
  { 3, { 70,  3 } },
  { 4, { 11,  7 } },
  { 5, { 93, 83 } },
  { 6, { 96, 22 } },
  { 7, { 26, 97 } },
  { 8, { 52, 58 } },
  { 9, { 68,  6 } } };

// fitness:
class Fitness
{
  public:
    float operator()(const Route &seq) const
    {
      float f = 0;

      for(ea::sequence_len_t i = 0; i < sequence_len(seq) - 1;
          ++i)
      {
        auto a = sequence_get(seq, i);
        auto b = sequence_get(seq, i + 1);

        f += sqrt(pow(Cities[b].x - Cities[a].x, 2) + pow(
            Cities[b].y - Cities[a].y, 2));
      }

      return f;
    }
};

// base class type declaration:
typedef CPGenomeBase<City, Fitness> Base;

// factory:
class RouteFactory : public ea::AFactory<Route>
{
  public:
    Route create_sequence()
    {
      int32_t genes[10];

      _g.get_unique_int32_seq(0, 9, genes, 10);

      auto seq = _base.create(10);

      for(ea::sequence_len_t i = 0; i < 10; ++i)
      {
        _base.set(seq, i, genes[i]);
      }

      return seq;
    }

  private:
    static AnsiRandomNumberGenerator _g;
    static Base _base;
};

AnsiRandomNumberGenerator RouteFactory::_g;
Base RouteFactory::_base;

// process pipeline:
int
main(int argc, char *argv[])
{
  Base base;
  RouteFactory f;
  vector<Route> population;
  auto rnd = make_shared<ea::AnsiRandomNumberGenerator>();

  // create 100 routes:
  auto a = make_output_adapter(population);

  f.create_population(100, a);

  // create and process pipeline:
  auto source = make_input_adapter(population);

  cout << "mean␣fitness␣(parent):␣" << mean<Base>(source) <<
      endl;
  cout << "median␣fitness␣(parent):␣" << median<Base>(source)
      << endl;

  vector<Route> children;
  auto cadapter = make_output_adapter(children);

  auto selection_a =
    SelectionElement<Base, SourceDivisor<Base>>
      (new TournamentSelection<Base, std::less<double>>());

  auto selection_b =
    SelectionElement<Base, FixedSelectionSize<Base, 50>>(
      new DoubleTournamentSelection<Base, std::less<double>>()
          );

  auto crossover =
    CrossoverElement<Base>
      (new EdgeRecombinationCrossover<Base>());

  auto mutation =
    MutationElement<Base>
      (new SingleSwapMutation<Base>(), rnd);

  auto terminator = ForLoopTerminator<Base>(100);
  ITerminator<Base>& terminator_ref = terminator;

  pipeline_process<Base>(source, cadapter, { &selection_a, &
      crossover, &selection_b, &mutation }, terminator_ref);

  source = make_input_adapter(children);

  cout << "mean fitness (parent): " << mean<Base>(source) <<
      endl;
  cout << "median fitness (parent): " << median<Base>(source)
      << endl;

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

  return 0;
}
```

4 Summary

As you have seen in this introduction LIBEA provides a basic framework to write evolutionary algorithms in a reliable way. There are some standard operators available which can be used with different kind of genomes.

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 document.