ParadisEO-PEO - Lesson 1

Creating a simple ParadisEO-PEO Evolutionary Algorithm

Introduction

One of the first steps in designing an evolutionary algorithm using the ParadisEO-PEO framework consists in having a clear overview of the implemented algorithm. A brief pseudo-code description is offered bellow - the entire source code for the ParadisEO-PEO evolutionary algorithm is defined in the **peoEA.h** header file. The main elements to be considered when building an evolutionary algorithm are the transformation operators, i.e. crossover and mutation, the evaluation function, the continuation criterion and the selection and replacement strategy.

```
do
{
    select( population, offsprings );       // select the offsprings from the current population
    transform( offsprings );                // crossover and mutation operators are applied on the selected
                                            offsprings
    evaluate( offsprings );                 // evaluation step of the resulting offsprings
    replace( population, offsprings );      // replace the individuals in the current population whith
                                            individuals from the offspring population, according to a
                                            specified replacement strategy
}   while ( eaCheckpointContinue( population )
    );
```

The **peoEA** class offers an elementary evolutionary algorithm implementation. The **peoEA** class has the underlying structure for including parallel evaluation and parallel transformation operators, migration operators etc. Although there is no restriction on using the algorithms provided by the EO framework, no parallelism is provided - the EO implementation is exclusively sequential.

Requirements

You should have already installed the ParadisEO-PEO package - this requires several additional packages which should be already included in the provided archive. The installation script has to be launched in order to configure and compile all the required components. At the end of the installation phase you should end up having a directory tree resembling the following:

```
...
paradiseo-mo
paradiseo-moeo
paradiseo-peo
      doc
      tutorial
          Lesson1
          Lesson2
          examples
               tsp
          ...
      src
      ...
...
```

The source-code for this tutorial may be found in the **paradiseo-peo/tutorial/Lesson1** directory, in the main.cpp file. We strongly encourage creating a backup copy of the file if you consider modifying the source code. For a complete reference on the TSP-related classes and definitions please refer to the files under the **paradiseo-peo/tutorial/examples/tsp**. After the installation phase you should end up having an **tspExample**

executable file in the **paradiseo-peo/tutorial/Lesson1** directory. We will discuss testing and launching aspects later in the tutorial.

You are supposed to be familiar with working in C/C++ (with an extensive use of templates) and you should have at least an introductory background in working with the EO framework.

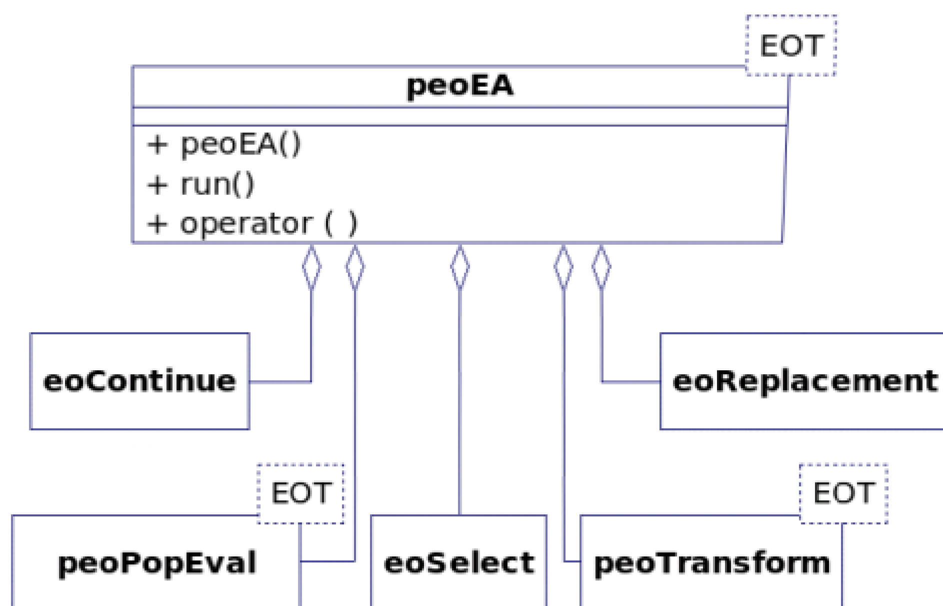
NOTE: All the presented examples have as case study the Traveling Salesman Problem (TSP). All the presented tutorials rely on a common shared source code defining transformation operators, evaluation functions, etc. for the TSP problem. For a complete understanding of the presented tutorials please take your time for consulting and for studying the additional underlying defined classes.

Problem Definition and Representation

As we are not directly concerned with the *Traveling Salesman Problem*, and to some extent out of scope, no in depth details are offered for the TSP. The problem requires finding the shortest path connecting a given set of cities, while visiting each of the specified cities only once and returning to the startpoint city. The problem is known to be NP-complete, i.e. no polynomial time algorithm exists for solving the problem in exact manner.

The construction of a ParadisEO-PEO evolutionary algorithm requires following a few simple steps - please take your time to study the signature of the **peoEA** constructor:

```
peoEA(
        eoContinue< EOT >& __cont,

        peoPopEval< EOT >& __pop_eval,

        eoSelect< EOT >& __select,

        peoTransform< EOT >& __trans,

        eoReplacement< EOT >& __replace

);
```



A few remarks have to be made: while most of the parameters are passed as EO-specific types, the evaluation and the transformation objects have to be derived from the ParadisEO-PEO **peoPopEval** and **peoTransform**

classes. Derived classes like the **peoParaPopEval** and **peoParaSGATransform** classes allow for parallel evaluation of the population and parallel transformation operators, respectively. Wrappers are provided thus allowing to make use of the EO classes.

In the followings, the main required elements for building an evolutionary algorithm are enumerated. For complete details regarding the implementation aspects of each of the components, please refer to the common shared source code. Each of the bellow referred header files may be found in the **pardiseo-peo/tutorial/examples/tsp** directory.

1. **representation**
   - the first decision to be taken concerns the representation of the individuals. You may create your own representation or you may use/derive one of the predefined classes of the EO framework.

For our case study, the TSP, each city is defined as a **Node** in the **node.h** header file - in fact an unsigned value defined as **typedef unsigned Node**. Moreover, each individual (of the evolutionary algorithm) is represented as a **Route** object, a vector of **Node** objects, in the **route.h** header file - **typedef eoVector< int, Node > Route**. The definition of the **Route**
object implies two elements: (1) a route is a vector of nodes, and (2) the fitness is an integer value (please refer to the *eoVector* definition in the EO framework).

In addition you should also take a look in the **route_init.h** header file which includes the **RouteInit** class, defined for initializing in random manner **Route** objects.

2. **evaluation function**
   - having a representation model, an evaluation object has to be defined, implementing a specific fitness function. The designed class has to be derived (directly or indirectly) from the **peoPopEval** class - you have the choice of using **peoSeqPopEval** or **peoParaPopEval**
   for sequential and parallel evaluation, respectively. These classes act as wrappers requiring the specification of an EO evaluation object derived from the eoEvalFunc class - please refer to their respective documentation.

The fitness function for our TSP case study is implemented in the **route_eval.h** header file. The class is derived from the eoEvalFunc EO class, being defined as **class RouteEval : public eoEvalFunc< Route >**.

3. **transformation operators**
   - in order to assure the evolution of the initial population, transformation operators have to be defined. Depending on your problem, you may specify quadruple operators (two input individuals, two output resulting individuals), i.e. crossover operators, binary operators (one input individual and one output resulting individual), i.e. mutation operators, or combination of both types. As for the evaluation function, the signature of the **peoEA** constructor requires specifying a **peoTransform** derived object as transformation operator.

The transform operators, crossover and mutation, for the herein presented example are defined in the **order_xover.h** and the **city_swap.h** header files, respectively.

4. **continuation criterion**
   - the evolutionary algorithm evolves in an iterative manner; a continuation criterion has to be specified. One of the most common and simplest options considers a maximum number of generations. It is your choice whether to use a predefined EO class for specifying the continuation criterion or using a custom defined class. In the later case you have to make sure that your class derives the eoContinue class.

5. **selection strategy**
- at each iteration a set of individuals are selected for applying the transform operators, in order to obtain the offspring population. As the specified parameter has to be derived from the eoSelect it is your option of whether using the EO provided selection strategies or implementing your own, as long as it inherits the eoSelect class.

For our example we chose to use the eoRankingSelect strategy, provided in the EO framework.

6. **replacement strategy**
- once the offspring population is obtained, the offsprings have to be integrated back into the initial population, according to a given strategy. For custom defined strategies you have to inherit the eoReplacement EO class. We chose to use an eoPlusReplacement as strategy (please review the EO documentation for details on the different strategies available).

A simple example for constructing a peoEA object

The source code for this example may be found in the main.cpp file, under the **paradiseo-peo/tutorial/Lesson1** directory. Please make sure you are under the right directory. At this point you have two options: (a) you can just follow the example without touching the main.cpp or, (b) you can start from scratch, following the presented steps, in which case you are required to make a backup copy of the main.cpp file and replace the original file with an empty one.

1. **include the necessary header files**
- as we will be using Route objects, we have to include the files which define the Route type, the initializing functor and the evaluation functions. Furthermore, in order to make use of transform operators, we require having the headers which define the crossover and the mutation operators. All these files may be found in the shared directory that we mentioned in the beginning. At this point you should have something like the following:

```
#include "route.h"
#include "route_init.h"
#include "route_eval.h"

#include "order_xover.h"
#include "city_swap.h"
```

In addition we require having the paradiseo header file, in order to use the ParadisEO-PEO features, and a header specific for our problem, dealing with processing command-line parameters - the param.h header file. The complete picture at this point with all the required header files is as follows:

```
#include "route.h"
#include "route_init.h"
#include "route_eval.h"

#include "order_xover.h"
#include "city_swap.h"

#include "param.h"

#include <paradiseo>
```

NOTE: the paradiseo header file is in fact a "super-header" - it includes all the esential ParadisEO-PEO header files. It is at at your choice if you want use the paradiseo header file or to explicitly include different header files, like the peoEA.h header file, for example.

2. **define problem specific parameters**
- in our case we have to specify how many individuals we want to have in our population, the number of generations for the evolutionary algorithm to iterate and the probabilities associated with the crossover and mutation operators.

```
#include "route.h"
#include "route_init.h"
#include "route_eval.h"
```

```
#include "order_xover.h"
#include "city_swap.h"

#include "param.h"

#include <paradiseo>


#define POP_SIZE 10
#define NUM_GEN 100
#define CROSS_RATE 1.0
#define MUT_RATE 0.01
```

3. **construct the skeleton of a simple ParadisEO-PEO program** - the main function including the code for initializing the ParadisEO-PEO environment, for loading problem data and for shutting down the ParadisEO-PEO environment. From this point on we will make abstraction of the previous part referring only to the main function.

```
...

int main( int __argc, char** __argv ) {

        // initializing the ParadisEO-PEO environment
        peo :: init( __argc, __argv );

        // processing the command line specified parameters
        loadParameters( __argc, __argv );

        // EVOLUTIONARY ALGORITHM TO BE DEFINED

        peo :: run( );
        peo :: finalize( );
        // shutting down the ParadisEO-PEO environment

        return 0;
}
```

4. **initialization functors, evaluation function and transform operators** - basically we only need to create instances for each of the enumerated objects, to be passed later as parameters for higher-level components of the evolutionary algorithm.

```
// random init object - creates random Route objects
RouteInit route_init;
// evaluator object - offers a fitness value for a specified Route object
RouteEval full_eval;

// crossover operator - creates two offsprings out of two specified parents
OrderXover crossover;
// mutation operator - randomly mutates one gene for a specified individual
CitySwap mutation;
```

5. **construct the components of the evolutionary algorithm** - each of the components that has to be passed as parameter to the **peoEA**
constructor has to be defined along with the associated parameters. Except for the requirement to provide the appropriate objects (for example, a **peoPopEval** derived object must be specified for the evaluation functor), there is no strict path to follow. It is your option what elements to mix, depending on your problem - we aimed for simplicity in our example.

- an initial population has to be specified; the constructor accepts the specification of an initializing object. Further, an evaluation object is required - the **peoEA** constructor requires a **peoPopEval** derived class.

```
// initial population for the algorithm having POP_SIZE individuals
eoPop< Route > population( POP_SIZE, route_init );
// evaluator object - to be applied at each iteration on the entire population
peoSeqPopEval< Route > eaPopEval( full_eval );
```

- the evolutionary algorithm continues to iterate till a continuation criterion is not met. For our case we consider a fixed number of generations. The continuation criterion has to be specified as a checkpoint object, thus requiring the creation of an *eoCheckPoint* object in addition.

```
// continuation criterion - the algorithm will iterate for NUM_GEN generations
eoGenContinue< Route > eaCont( NUM_GEN );
// checkpoint object - verify at each iteration if the continuation criterion is met
eoCheckPoint< Route > eaCheckpointContinue( eaCont );
```

- selection strategy - we are required to specify a selection strategy for extracting individuals out of the parent population; in addition the number of individuals to be selected has to be specified.

```
// selection strategy - applied at each iteration for selecting parent individuals
eoRankingSelect< Route > selectionStrategy;
// selection object - POP_SIZE individuals are selected at each iteration
```

```
eoSelectNumber< Route > eaSelect( selectionStrategy, POP_SIZE );
```

- transformation operators - we have to integrate the crossover and the mutation functors into an object which may be passed as a parameter when creating the **peoEA** object. The constructor of **peoEA** requires a **peoTransform** derived object. Associated probabilities have to be specified also.

```
// transform operator - includes the crossover and the mutation operators with a specified associated rate
eoSGATransform< Route > transform( crossover, CROSS_RATE, mutation, MUT_RATE );
// ParadisEO transform operator (peo prefix) - wraps an e EO transform object
peoSeqTransform< Route > eaTransform( transform );
```

- replacement strategy - required for defining the way for integrating the resulting offsprings into the initial population. At your option whether you would like to chose one of the predefined replacement strategies that come with the EO framework or if you want to define your own.

```
// replacement strategy - for replacing the initial population with offspring individuals
eoPlusReplacement< Route > eaReplace;
```

6. **evolutionary algorithm**

- having defined all the previous components, we are ready for instanciating an evolutionary algorithm. In the end we have to associate a population with the algorithm, which will serve as the initial population, to be iteratively evolved.

```
peoEA< Route > eaAlg( eaCheckpointContinue, eaPopEval, eaSelect, eaTransform, eaReplace );

// specifying the initial population for the algorithm, to be iteratively evolved
eaAlg( population );
```

If you have not missed any of the enumerated points, your program should be like the following:

```
#include "route.h"
#include "route_init.h"
#include "route_eval.h"

#include "order_xover.h"
#include "city_swap.h"

#include "param.h"

#include <paradiseo>


#define POP_SIZE 10
#define NUM_GEN 100
#define CROSS_RATE 1.0
#define MUT_RATE 0.01


int main( int __argc, char** __argv ) {

        // initializing the ParadisEO-PEO environment
        peo :: init( __argc, __argv );

        // processing the command line specified parameters
        loadParameters( __argc, __argv );

        // init, eval operators, EA operators ---------------------------------------------

        // random init object - creates random Route objects
        RouteInit route_init;
        // evaluator object - offers a fitness value for a specified Route object
        RouteEval full_eval;

        // crossover operator - creates two offsprings out of two specified parents
        OrderXover crossover;
        // mutation operator - randomly mutates one gene for a specified individual
        CitySwap mutation;
        // ----------------------------------------------

        // evolutionary algorithm components ---------------------------------------------

        // initial population for the algorithm having POP_SIZE individuals
        eoPop< Route > population( POP_SIZE, route_init );
        // evaluator object - to be applied at each iteration on the entire population
        peoSeqPopEval< Route > eaPopEval( full_eval );

        // continuation criterion - the algorithm will iterate for NUM_GEN generations
        eoGenContinue< Route > eaCont( NUM_GEN );
        // checkpoint object - verify at each iteration if the continuation criterion is met
        eoCheckPoint< Route > eaCheckpointContinue( eaCont );

        // selection strategy - applied at each iteration for selecting parent individuals
        eoRankingSelect< Route > selectionStrategy;
        // selection object - POP_SIZE individuals are selected at each iteration
        eoSelectNumber< Route > eaSelect( selectionStrategy, POP_SIZE );

        // transform operator - includes the crossover and the
        // mutation operators with a specified associated rate
        eoSGATransform< Route > transform( crossover, CROSS_RATE, mutation, MUT_RATE );
        // ParadisEO transform operator (peo prefix) - wraps an e EO transform object
        peoSeqTransform< Route > eaTransform( transform );

        // replacement strategy - for replacing the initial
        // population with offspring individuals
        eoPlusReplacement< Route > eaReplace;
        // ----------------------------------------------
```

```
        // ParadisEO-PEO evolutionary algorithm -------------------------------------------

        peoEA< Route > eaAlg( eaCheckpointContinue, eaPopEval, eaSelect, eaTransform, eaReplace );

        // specifying the initial population for the algorithm, to be iteratively evolved
        eaAlg( population );
        // ---------------------------------------------


        peo :: run( );
        peo :: finalize( );
        // shutting down the ParadisEO-PEO environment

        return 0;
}
```

Compilation and Execution

First, please make sure that you followed all the previous steps in defining the evolutionary algorithm. Your file should be called **main.cpp**
- please make sure you do not rename the file (we will be using a pre-built makefile, thus you are required not to change the file names). Please make sure you are in the **paradiseo-peo/tutorial/Lesson1** directory - you should open a console and you should change your current directory to the one of Lesson1.

Compilation: being in the **paradiseo-peo/tutorial/Lesson1 directory**, you have to type *make*. As a result the **main.cpp** file will be compiled and you should obtain an executable file called **tspExample**. If you have errors, please verify any of the followings:

- you are under the right directory - you can verify by typing the *pwd* command - you should have something like **.../paradiseo-peo/tutorial/Lesson1**
- you saved your modifications in a file called **main.cpp**, in the **paradiseo-peo/tutorial/Lesson1** directory
- there are no differences between the presented example and your file.

NOTE: in order to successfully compile your program you should already have installed an MPI distribution in your system.

Execution: the execution of a ParadisEO-PEO program requires having already created an environment for launching MPI programs. For *MPICH-2*, for example, this requires starting a ring of daemons. The implementation that we provided as an example is sequential and includes no parallelism - we will see in the end how to include also parallelism. Executing a parallel program requires specifying a mapping of resources, in order to assing different algorithms to different machines, define worker machines etc. This mapping is defined by an XML file called **schema.xml**, which, for our case, has the following structure:

```
<?xml version="1.0"?>

<schema>
        <group scheduler="0">
                <node name="0" num_workers="0">
                </node>

                <node name="1" num_workers="0">
                <runner>1</runner>
                </node>

                <node name="2" num_workers="1">
                </node>
                <node name="3" num_workers="1">
                </node>
        </group>
</schema>
```

Not going into details, the XML file presented above describes a mapping which includes four nodes, the first one having the role of scheduler, the second one being the node on which the evolutionary algorithm is actually executed and the third and the fourth ones being slave nodes. Overall the mapping says that we will be launching four processes, out of which only one will be executing the evolutionary algorithm. The other node entries in the XML file have no real functionality as we have no parallelism in our program - the entries were created for you convenience, in order to provide a smooth transition to creating a parallel program.

Launching the program may be different, depending on your MPI distribution - for MPICH-2, in a console, in the **paradiseo-peo/tutorial/Lesson1** directory you have to type the following command:

mpiexec -n 4 ./tspExample @lesson.param

NOTE: the "-n 4" indicates the number of processes to be launched. The last argument, "@lesson.param", indicates a file which specifies different application specific parameters (the mapping file to be used, for example, whether to use logging or not, etc).

The result of your execution should be similar to the following:

```
Loading '../data/eil101.tsp'.
NAME: eil101.
COMMENT: 101-city problem (Christofides/Eilon).
TYPE: TSP.
DIMENSION: 101.
EDGE_WEIGHT_TYPE: EUC_2D.
Loading '../data/eil101.tsp'.
NAME: eil101.
COMMENT: 101-city problem (Christofides/Eilon).
EOF.
TYPE: TSP.
DIMENSION: 101.
EDGE_WEIGHT_TYPE: EUC_2D.
EOF.
Loading '../data/eil101.tsp'.
NAME: eil101.
COMMENT: 101-city problem (Christofides/Eilon).
TYPE: TSP.
DIMENSION: 101.
EDGE_WEIGHT_TYPE: EUC_2D.
EOF.
Loading '../data/eil101.tsp'.
NAME: eil101.
COMMENT: 101-city problem (Christofides/Eilon).
TYPE: TSP.
DIMENSION: 101.
EDGE_WEIGHT_TYPE: EUC_2D.
EOF.
STOP in eoGenContinue: Reached maximum number of generations [100/100]
```

Introducing parallelism

Creating parallel programs with ParadisEO-PEO represents an easy task once you have the basic structure for your program. For experimentation, in the **main.cpp** file, replace the line

```
peoSeqPopEval< Route > eaPopEval( full_eval );
```

with

```
peoParaPopEval< Route > eaPopEval( full_eval );
```

The second line only tells that we would like to evaluate individuals in parallel - this is very interesting if you have a time consuming fitness evaluation function. If you take another look on the **schema.xml** XML file you will see the last two nodes being marked as slaves (the "num_workers" attribute) - these nodes will be used for computing the fitness of the individuals.

At this point you only have to recompile your program and launch it again - as we are not using a time consuming fitness fitness function, the effects might not be visible - you may increase the number of individuals to experiment.