**From Paradiseo**

# Paradiseo: MoLesson3

# Paradiseo-MO Lesson 3: Implementing a Simulated Annealing

In this lesson, the source code is in *tutorial/Lesson3* and the executable in *build/tutorial/Lesson3*.

According to the color and the style used, an object is:

- *object* <=> a generic template (abstract).
- **object** <=> an implementation of a generic template class (ready-to-use class).
- *object* <=> a template that the user need to implement for his problem.
- **object** <=> an implementation (of a blue or red template) that the user have designed for his problem.

## Introduction

Before beginning this lesson, be sure that the ParadisEO-MO package has been compiled (see the README file in the Paradiseo-MO directory if you do not know how compile the Paradiseo-MO package).

Furthermore, it can be interesting to have opened the documentation (*file doc/html/index.html*).

**NB:** for the Unix user, type also the command *make install* to add the *benchs* directory in the right localization.

Open a terminal and from the ParadisEO-MO directory, go where the executable is (*build/tutorial/Lesson3*).

A simulated annealing has been already designed and compiled. It works on the Traveling Salesman Problem (TSP). You can test it with an instance that is in the "benchs" directory. For example, you can type :

*./simulted_annealing ../examples/tsp/benchs/berlin52.tsp*

Here, an example of what you can see on your screen after executing the command:

>> Loading [../examples/tsp/benchs/berlin52.tsp]

[From] -29414 52 1 20 40 48 9 27 13 22 5 28 24 29 21 26 44 38 33 37 45 31 42 18 12 3 14 36 30 6 51 32

17 11 0 34 4 10 43 50 16 2 23 35 19 46 49 39 25 15 41 8 7 47

[To] -8329 52 1 6 41 20 16 2 17 30 22 29 19 49 28 15 43 45 36 37 39 38 33 34 35 48 0 21 31 44 18 40 7 8 9 42 32 50 10 51 13 12 46 25 26 27 11 24 3 5 14 4 23 47

You have launched a simulated annealing on the berlin52 TSP instance.

# A Simulated Annealing for the TSP

## Format of the 'simulated_annealing.cpp' file:

Now open the 'simulated_annealing.cpp' file (in *tutorial/Lesson3*). You can see 3 main parts in the file:

- *comments* (from line 1 to line 10): it contains information about the file. **NB: if you have have any problems, use the address given in these commentaries.**

- *include* (from line 12 to line 21): it indicates which files are needed to be included in the program. The file names beginning by "mo" correspond to files which are the 'src' directory of the ParadisEO-MO package and the others to files specifically designed for making the hill climbing for the TSP (these files are in the directory 'tutorial/examples/tsp/src').

- *main* (from line 23 to the end) : it contains the simulated annealing main code.

**NB : during this lesson, if you want to use an object that is not indicated in the include part, you need to include the corresponding file before using it.**

### moSA class:

In order to better understand this lesson, open the documentation generated by doxygen (file *doc/html/index.html*). On the top of the opened window, click on "Classes", then on "Class Hierarchy" and to finish on "moSA<M>". You have now the documentation of the **moSA** class which describes a simulated_annealing in the ParadisEO-MO package.

In the **moSA** class, there is only one constructor, click on it in the documentation (click on "moSA" under "Public Member Functions").

You have now the description of the constructor and its 6 parameters. Four parameters are paradiseo-mo objects (beginning by 'mo') and the sixth is a paradiseo-eo object (beginning by 'eo').

You have certainly noticed a <M> that appears everywhere... it correspond to the notion of move (*moMove*). A **moSA** is a generic class (a template) that can be used on what you want according to the fact you give a description for <M>.

In this example, <M> corresponds to **TwoOpt**. It is a well know operator for the TSP. But how has been designed the **TwoOpt** type ??? It depends on how a TSP solution is represented.

## Route and TwoOpt:

In our example, a TSP solution is a **Route**, more precisely it is a vector of unsigned and it can have a fitness represented by a float (described in 'route.h' in *tutorial/examples/tsp/*).

**TwoOpt** describe a move that can be applied to a TSP solution, in our case a **Route**... that is why in the 'two_opt.h' file, you can see '... public moMove<Route> ...'. So 'two_opt.h' describes a **TwoOpt** move and 'two_opt.cpp' explains how applies a **TwoOpt** move to a route. In our case, a **TwoOpt** object has only two parameters that corresponds to the index of 2 cities. Apply of **TwoOpt** move on a TSP solution (a **Route** in our example) corresponding to swap all the cities between these two indexes.

Only when the solution representation is chosen, the move can be designed. And only when the move is designed, we can design the objects needed by the moSA constructor.

## Constructor parameters:

As you have already noticed, 4 parameters of the constructor are based on a <M> type, so they are the classes that have to be designed according to the move representation.

- *moRandMove*: it only corresponds to the manner to give initial values to the move parameters. In our case, the object **TwoOptRand** is a *moRandMove* for the **TwoOpt** move ('...public moRandMove<TwoOpt>' in 'two_opt_rand.h). Its application is very simple: the first and the indexes are randomly generated (see 'two_opt_rand.cpp').

- *moIncrEval*: this object corresponds to the evaluation of a solution without knowing it (???). More precisely, we have a evaluated solution and a move; and we want to know the fitness of the solution gained when the move is applied. The interest is to find the fitness without completely evaluating the new **Route** (as it is made in an *eoEvalFunc*). In our case, a **TwopOptIncrEval** is a *moIncrEval* for the **TwoOpt** move ('...public moIncrMove<TwoOpt>' in 'two_opt_incr_eval.h'). Its application allows to know the fitness of a new solution without apply the move.

- *moSolContinue* : this object corresponds to the stop criterion during the neighborhood inspection. In our example, cont is a template class instanciated with a **TwoOpt** move ('...moGenSolContinue <Route> cont (1000)...' in tabu_search.cpp). This template allows the inspection of the current solution neighborhood until a maximum number of steps (in our example 1000). But you can also use **moFitSolContinue** (to reach a fitness threshold), **moNoFitImprSolContinue** (number of iterations without fitness improvement) or **moSteadyFitSolContinue** (after a minimum number of iterations is reached, number of iterations without improvement).

- ***double _init_temp*** : this parameter allows to initialize the temperature of the cooling system.

- *moCoolingSchedule* : this object corresponds to the cooling system used in the simulated annealing algorithm. In our example, cool_sched is a template class that implements an easy cooling system ('...moExponentialCoolingSchedule cool_sched (0.1, 0.98)' in simulated_annealing.cpp). The template class **moExponentialCoolingSchedule** decreases the temperature according to a factor (new_temperature=old_temperature*factor) until a threshold is reached.

- *eoEvalFunc* : as indicated earlier, it is not an object of paradiseo-mo but it is an object of paradiseo-eo. But the using is the same. An *eoEvalFunc* allows to evaluate a <EOT>; in our case, a <EOT> is a **Route**. So an object **RouteEval** is a *eoEvalFunc* for a **Route** ('...public eoEvalFunc<Route>' in route_eval.h). When a **Route** is given to a **RouveEval** object, the **Route** fitness is computed. Its application in our example corresponds to the sum of the distance between each consecutive city of the **Route**, it is a negative value (see 'route_eval.cpp').

## Instantiate our simulated annealing:

Now, we have all the parameters ready, it is also important to have an initial solution (a **Route** in our case) to launch the simulated annealing on it. That is the role of line 34 to 37 :

*Route route ;*

*RouteInit init ;*

*init (route) ;*

The **RouteInit** object is a eoInit object used with the Route type ('...public eoInit<Route>' in route_init.h). It is an interesting object in order to initialise solution (see paradiseo-eo documentation for eoInit).

The initial solution has to be evaluated, so we used our evaluation function (lines 39 and 40):

*RouteEval full_eval ;*

*full_eval (route) ;*

The remaining objects are prepared (lines 42 to 56):

*TwoOptRand two_opt_rand ;*

*TwoOptIncrEval two_opt_incr_eval ;*

*TwoOpt move ;*

*moExponentialCoolingSchedule cool_sched (0.1, 0.98) ;*

*moGenSolContinue <Route> cont (1000) ;*

The **moSA** is finally instantiated (line 60):

*moSA <TwoOpt> simulated_annealing (two_opt_rand, two_opt_incr_eval, cont, 1000, cool_sched, full_eval) ;*

It can be launched on our initial solution (line 61):

*simulated_annealing (route) ;*

## Modify the current simulated annealing:

An example of easy modification that can be possible for this application is to change the temperature threshold. In simulated_annealing.cpp change the value of the first parameter of the **moExponentialCoolingSchedule** (modify 0.1 to 0.01).

Save your modification ant type 'make' at the prompt of the Lesson3 directory (*build/tutorial/Lesson3*).

If you do not have any errors, you can test the new program as you have made earlier :

*./simulated_annealing ../examples/tsp/benchs/berlin52.tsp*

You can also try to use another implementation of the ***CoolingSchedule*** template: the **LinearCoolingSchedule**. In this implementation, the temperature decreases according to a quantity (new_temperature=old_temperature-factor) until a threshold is reached.

In the 'simulated_annealing.cpp' file, change the file from this:

*moExponentialCoolingSchedule cool_sched (0.1, 0.98) ; // Exponential Cooling Schedule*

*//moLinearCoolingSchedule cool_sched (0.1, 0.5) ; // Linear Cooling Schedule*

to this :

*//moExponentialCoolingSchedule cool_sched (0.1, 0.98) ; // Exponential Cooling Schedule*

*moLinearCoolingSchedule cool_sched (0.1, 0.5) ; // Linear Cooling Schedule*

In order to use the **moLinearCoolingSchedule**.

Save your modification ant type 'make' at the prompt of the Lesson3 directory (*/build/tutorial/Lesson3*).

If you do not have any errors, you can test the new program as you have made earlier :

*./simulated_annealing ../examples/tsp/benchs/berlin52.tsp*

It works ??? yes, **you are unbelievable**.

We are going to arrive at the end of this lesson... What have we learned ?

# Epilogue

In order to design a simulated annealing with the ParadisEO-MO packages, you need to chose how a solution of our problem is represented. Then you need to create a object which is a move (*moMove*) that works with our solution representation ('...public eoMove<mySolutionRepresentation>').

When the move representation is designed, you need to instantiate a **moSA** object. First, choose which constructor you want to use and watch carefully the description of each parameters. For each of them, thanks to the documentation, search if it already exists a generic implementation that has a behaviour you would like to use in order to not design something that already exists (example of the **moExponentialCoolingSchedule** and **moLinearCoolingSchedule** objects). If no interesting generic implementation can be used in your case, design your own according to the type of object you want to use (example of the **moRandMove** and **moIncrEval** objects).

Finally, create an initial solution, evaluate it, prepare all the moSA parameters, instantiate the **moSA** and launch it on your initial solution.

Good Luck.

# What is next ???

You want to try another solution based heuristic, try the **lesson 1** to learn to design a *hill climbing* algorithm, the **lesson 2** to learn to design a *tabu search* or maybe the **lesson 4** to learn to design an *iterated local search*.

---