



**Parallel Cooperative
Optimization Research
Group**

Reusable design of single solution-based metaheuristics

S. Cahon, E-G. Talbi, N. Melab



**Laboratoire d'Informatique
Fondamentale de Lille**





The main steps to build a Solution-based metaheuristic

- Always

- The neighborhood/the moves and some operators to manages it/them
- The way to initialize the starting condition

- Specialist

- Neighborhood selection strategy,
- Cooling schedule,
- Tabu list, ...
- The continuation criterion



Framework and tutorial application

- A framework for the design of Local Searches (Hill Climbing, Simulated Annealing and Tabu Search)
 - Moving Objects (MO)
- Tutorial application
 - The Traveling Salesman Problem (TSP)



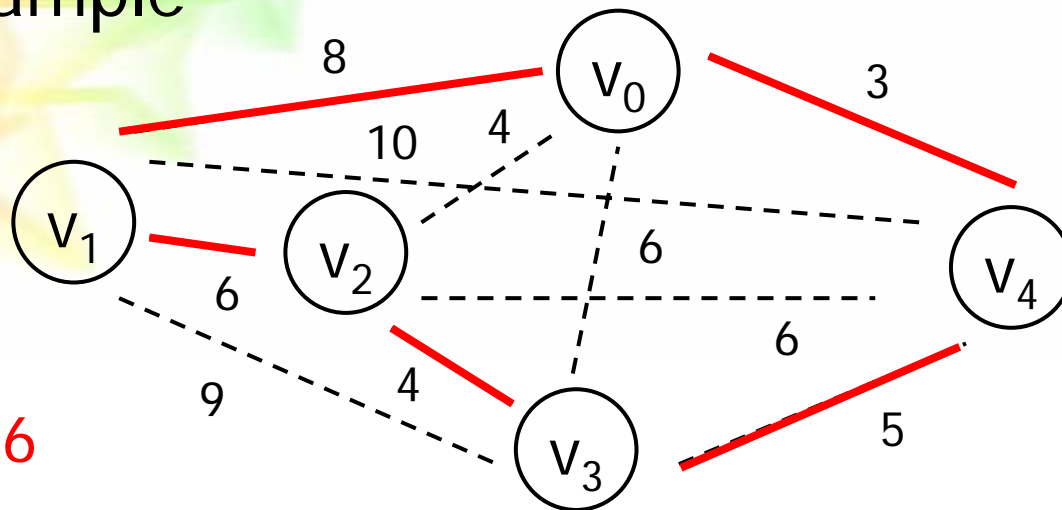
Moving Objects (MO)

- A framework that extends the EO library in order to design solution-based meta-heuristics
- Design/development team
 - S. Cahon, N. Melab and E-G Talbi (OPAC-LIFL)

<http://paradiseo.gforge.inria.fr>

Application to the Traveling Salesman Problem (TSP)

- "Given a collection of N cities and the distance between each pair of them, the **TSP** aims at finding the shortest route visiting all of the cities"
- Symmetric TSP
- Example



Length: 26



Common features

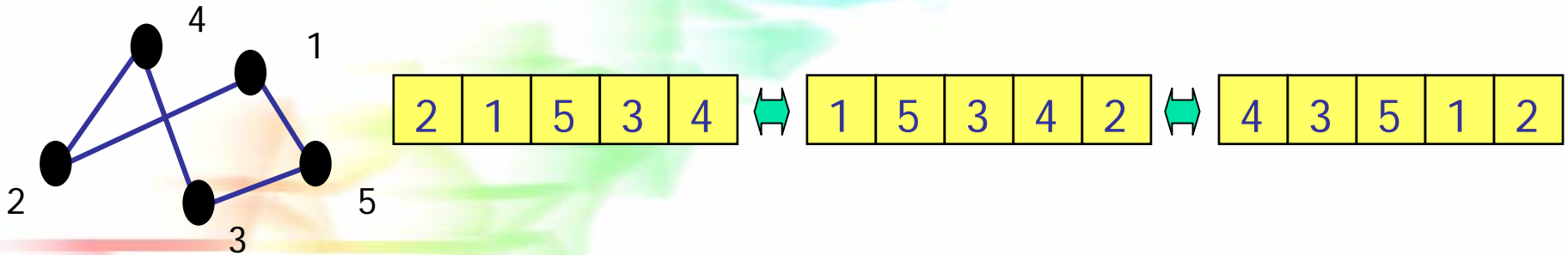


Designing the neighborhood

- It is strongly dependant of the representation of the solution !
- The nature of the landscape depends of it
- Issues
 - Moves that imply minor changes are preferred
 - Fitness delta should be easily and efficiently computed
 - Analysis of complexity (i.e. size of the neighborhood)

Application to the TSP

- Reminding the chosen coding
→ Ordered sequence of visited vertices



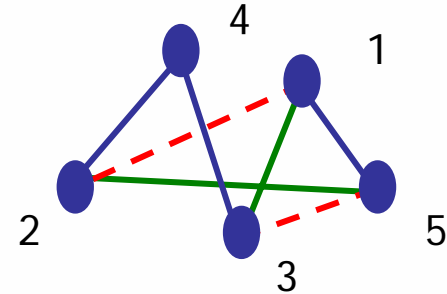
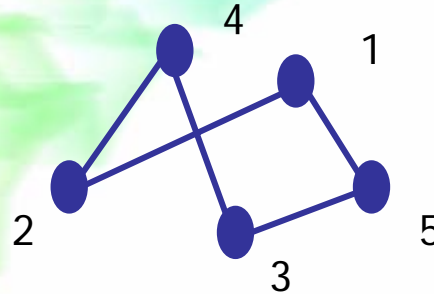
- Some relevant moves
→ **Two-opt**, **City-swap**, LK, etc

Two-opt

- Two points within the string are selected and the segment between them is inverted. This operator put in two new edges in the tour

2	1	5	3	4
---	---	---	---	---

2	5	1	3	4
---	---	---	---	---



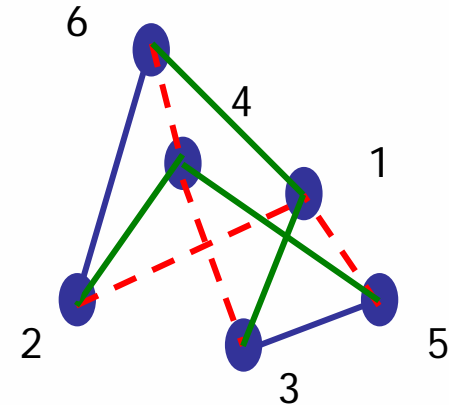
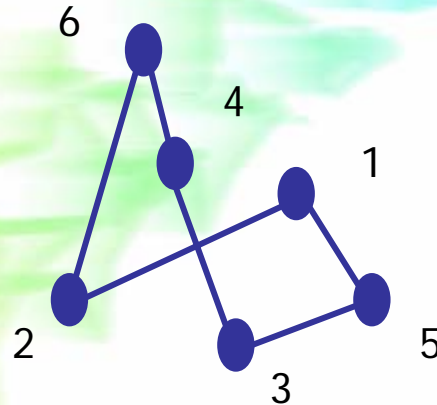
$$\Delta = -d(2,1) - d(5,3) + d(2,5) + d(1,3)$$

City-swap

- The values contained in two positions are exchanged, thus introducing four new edges in the string

2	1	5	3	4	6
---	---	---	---	---	---

2	4	5	3	1	6
---	---	---	---	---	---



$$\begin{aligned} \text{Delta} = & - d(2,1) - d(1,5) - d(3, 4) - d(4, 6) \\ & + d(2,4) + d(4,5) + d(3,1) + d(1,6) \end{aligned}$$

It is known less efficient than City-swap for many instances !



Strategies for the starting solution (to improve)

- Two main uses cases in practice
 - A randomly generated initial solution,
 - A solution built by another metaheuristic (greedy algorithm, genetic algorithm, ...)

The background features a series of overlapping, semi-transparent geometric shapes, primarily triangles and polygons, arranged in a diagonal path from the bottom-left towards the top-right. The colors of these shapes include shades of red, orange, yellow, green, blue, and purple. A black crosshair, consisting of a vertical and a horizontal line, is positioned on the left side of the slide, intersecting the text.

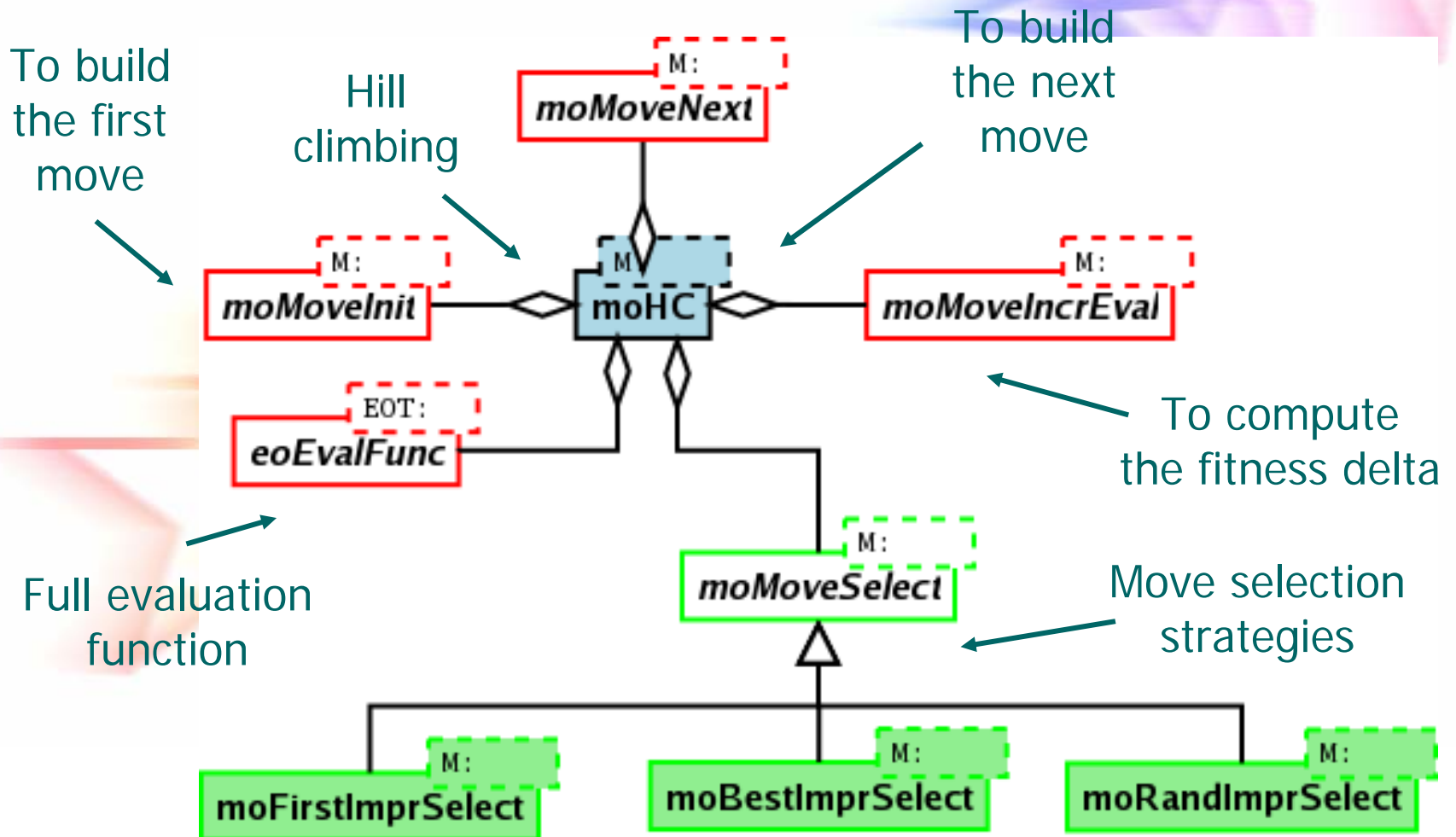
Hill Climbing



How to build a Hill Climbing ?

- Design a move operator, its features
- Design/implement the operator to build the first move (and implicitly the first neighboring candidate)
- Design/implement the operator to update a given move to its successor
- Design/implement the incremental evaluation function
- Chose the neighbour selection strategy
- No continuation criterion (stopping as a local optima is reached)

Core classes

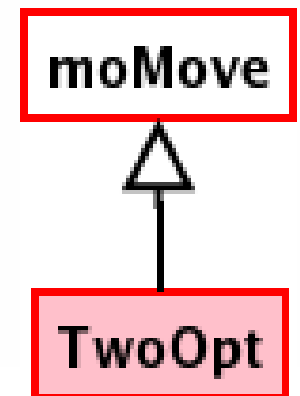


The Two-opt move

- A Two opt move is a couple of positions in the sequence of visited nodes

```
class TwoOpt : public moMove <Route>, public std :: pair <unsigned, unsigned> {  
public :  
    void operator () (Route & __route);  
};
```

To be implemented

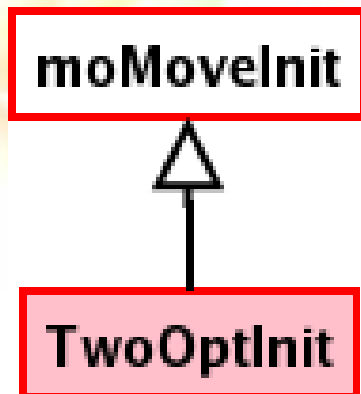


The Two-opt move initializer

- It initializes both positions to zero !

```
class TwoOptInit : public moMoveInit <TwoOpt> {  
public :  
    void operator () (TwoOpt & __move, const Route & __route) ;  
};
```

To be implemented





The Two opt move updater

- It increments the second position if possible. Else, it increments the first position, and reinitializes the second position

```
class TwoOptNext : public moMoveNext <TwoOpt> {  
public :  
    bool operator () (TwoOpt & __move, const Route & __route) ;  
};
```

To be implemented

Succeeding in building
the next move

The two-opt move incremental evaluation function

- It computes the new length from the costs of the added/removed edges

```
class TwoOptIncrEval : public moMoveIncrEval <TwoOpt> {  
  
public :  
  
    int operator () (const TwoOpt & __move, const Route & __route) ;  
};
```

New length

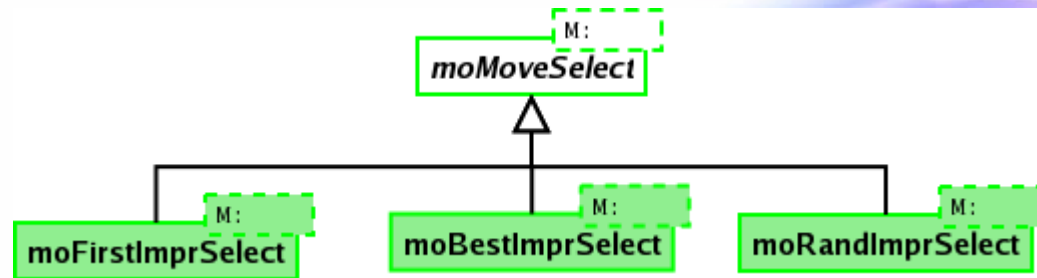
To be implemented

moMoveIncrEval



TwoOptIncrEval

Neighbour selection strategy



- Deterministic/full: choosing the **best neighbor** (i.e. that improves the most the cost function)
- Deterministic/partial: choosing **the first processed neighbour that is better** than the current solution,
- Stochastic/full: processing the whole neighborhood and applying **a random better one**



Implementation of a Hill Climbing

```
Route route; /* One solution */
RouteInit route_init; /* Its builds random routes */
route_init (route); /* Building a random starting solution */

RouteEval full_route_eval; /* Full route evaluator */
TwoOptInit two_opt_init; /* Initializing the first couple of edges to swap */
TwoOptNext two_opt_next; /* Updating a movement */
TwoOptIncrEval two_opt_incr_eval; /* Efficiently evaluating a given neighbor */

moBestImprSelect <TwoOpt> two_opt_move_select; /* Movement selection
strategy (elitist) */

/* Building the Hill Climbing from those components */
moHC <TwoOpt> hc (two_opt_init, two_opt_next, two_opt_incr_eval,
two_opt_move_select, full_route_eval);

/* It applies the HC to the solution */
hc (route);
```



Simulated Annealing

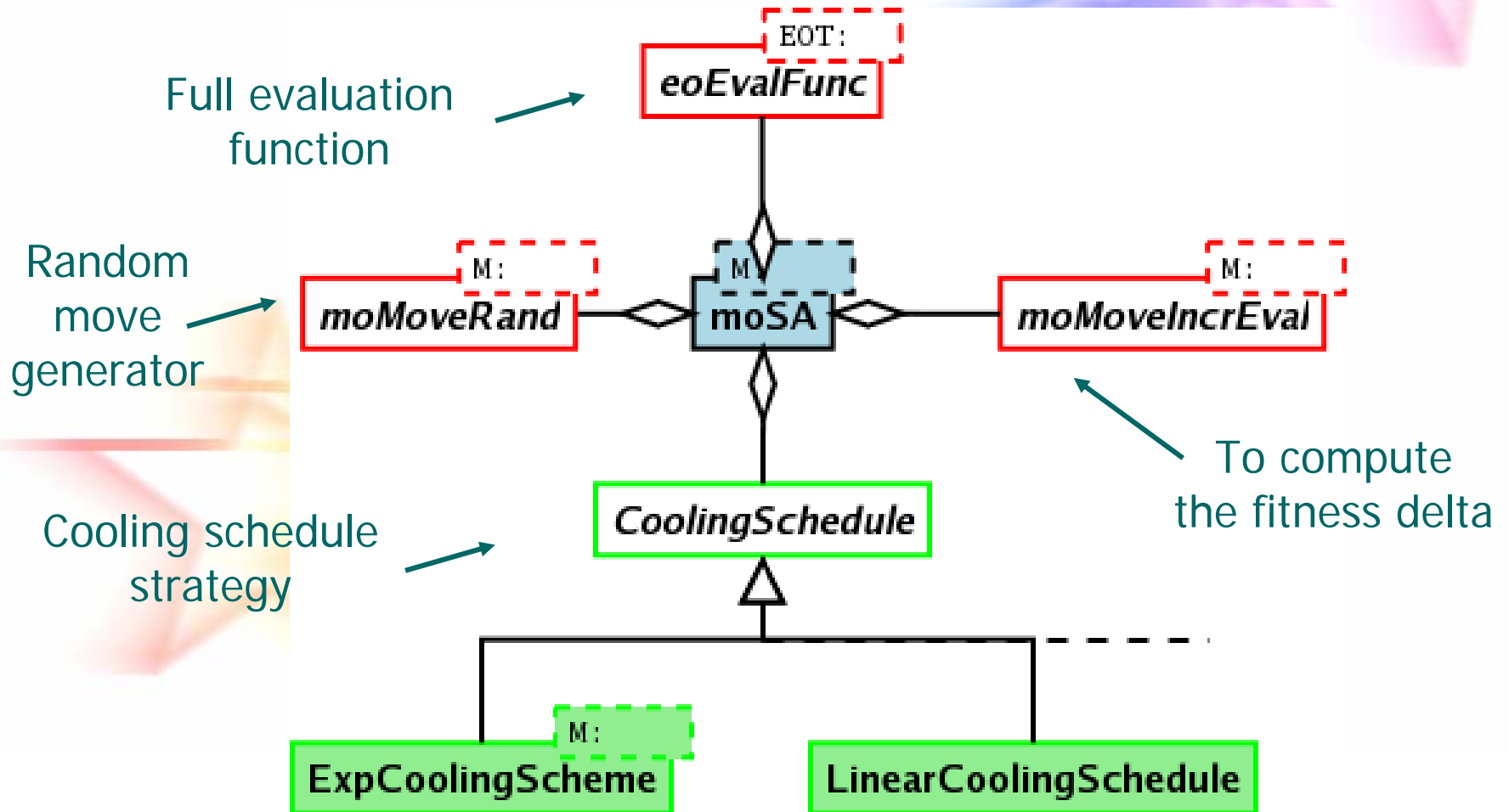
How to build a Simulated Annealing ?

Could be reused from Hill Climbing

- Design a move operator, its features
- Design/implement the operator to build a random candidate move
- Design/implement the incremental evaluation function
- Chose the cooling schedule strategy

Independent of the tackled problem

Core classes

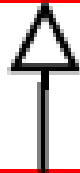


The Two-opt random move generator

- It randomly determines a couple of random positions !

```
class TwoOptRand : public moMoveRand<TwoOpt> {  
public :  
    void operator () (TwoOpt & __move, const Route & __route) ;  
};
```

moMoveRand

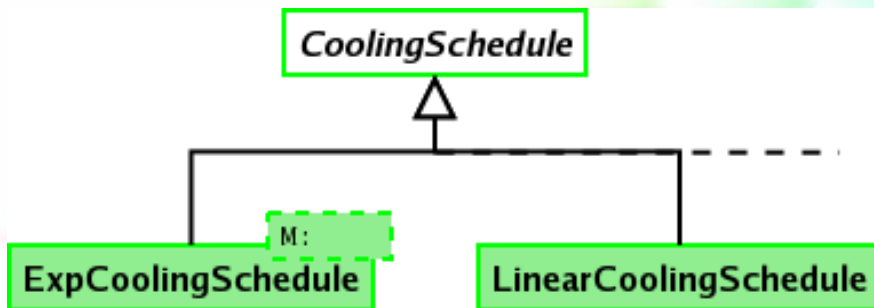


TwoOptRand

← To be implemented

Cooling Schedule

- Two (basic) strategies are already implemented: linear and exponential



- Linear

$$temp = temp - x$$

- Exponential

$$temp = temp * x$$

Implementation of Simulated Annealing

```
Route route; /* One solution */  
RouteInit route_init; /* It builds random routes */  
route_init (route); /* Building a random starting solution */
```

```
RouteEval full_route_eval; /* Full route evaluator */  
TwoOptRand two_opt_rand; /* It builds random candidate movements */  
TwoOptIncrEval two_opt_incr_eval; /* Efficiently evaluating a given neighbor */
```

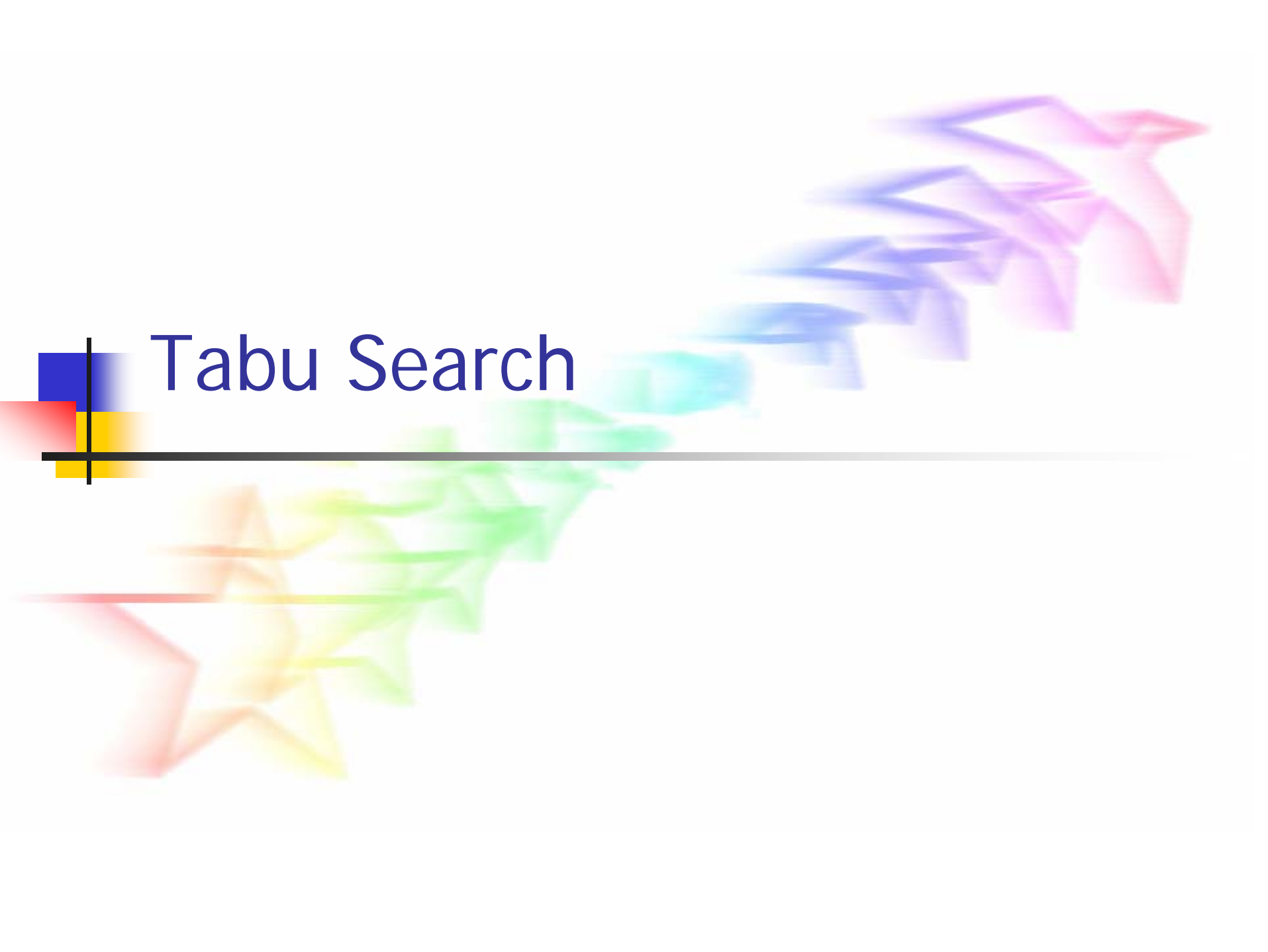
```
moExpCoolingSchedule cool_scheme (0.99, 1); /*Cooling schedule and associated  
parameters */
```

Factor and threshold

```
/* Building the Simulated Annealing from those components */  
moSA <TwoOpt> hc (two_opt_init, two_opt_incr_eval, 100, 100, cool_scheme,  
full_route_eval);
```

```
/* It applies the SA to the solution */  
sa (route);
```

Init. Temp and number of
iter. At any step



Tabu Search

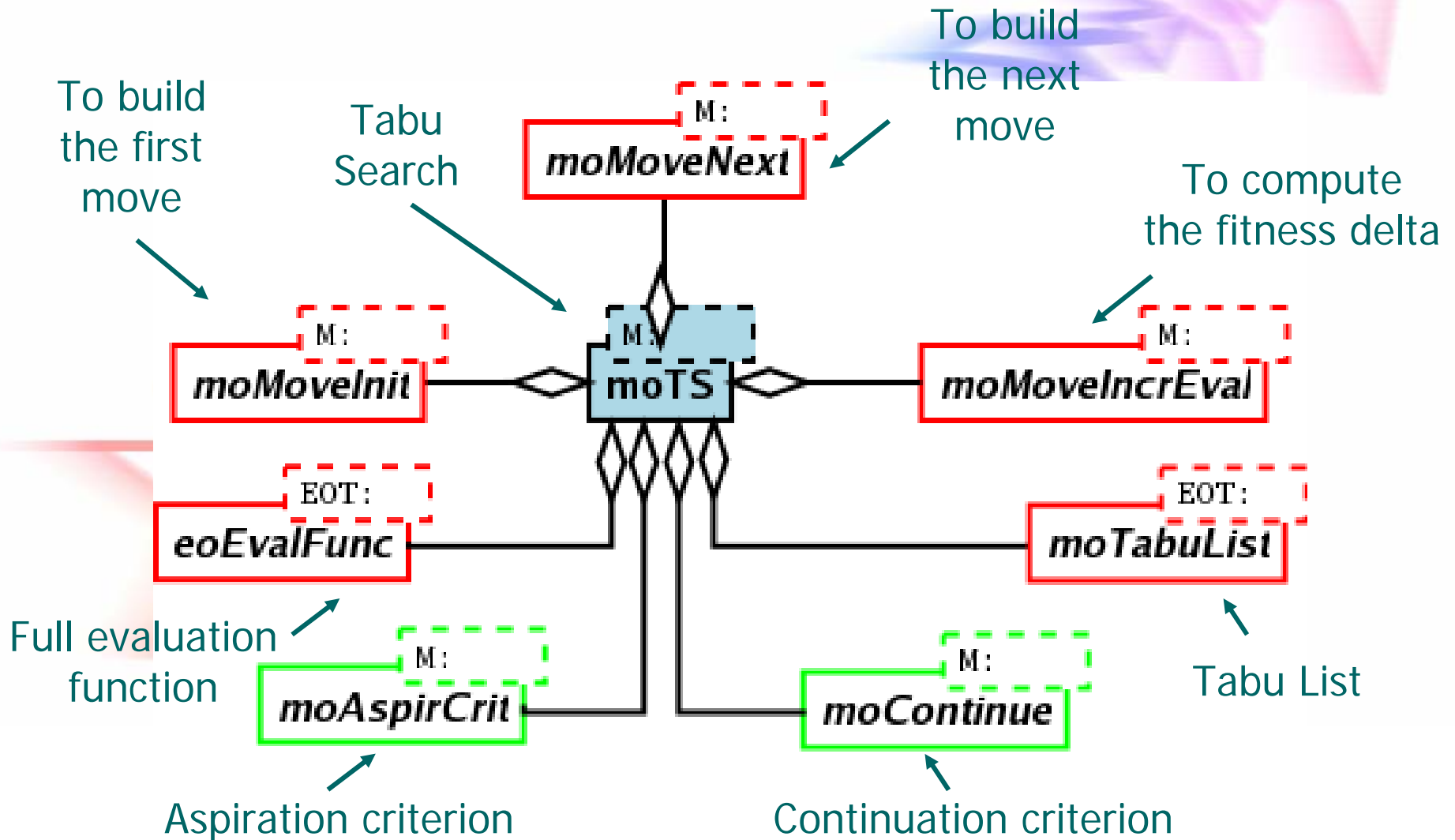


How to design a Tabu Search

- Design a move operator, its features
- Design/implement the operator to build the first move (and implicitly the first neighboring candidate)
- Design/implement the operator to update a given move to its successor
- Design/implement the incremental evaluation function
Could be reused from Hill Climbing
- Design/implement the Tabu List
- Choose the aspiration criterion
- Choose the continuation criterion

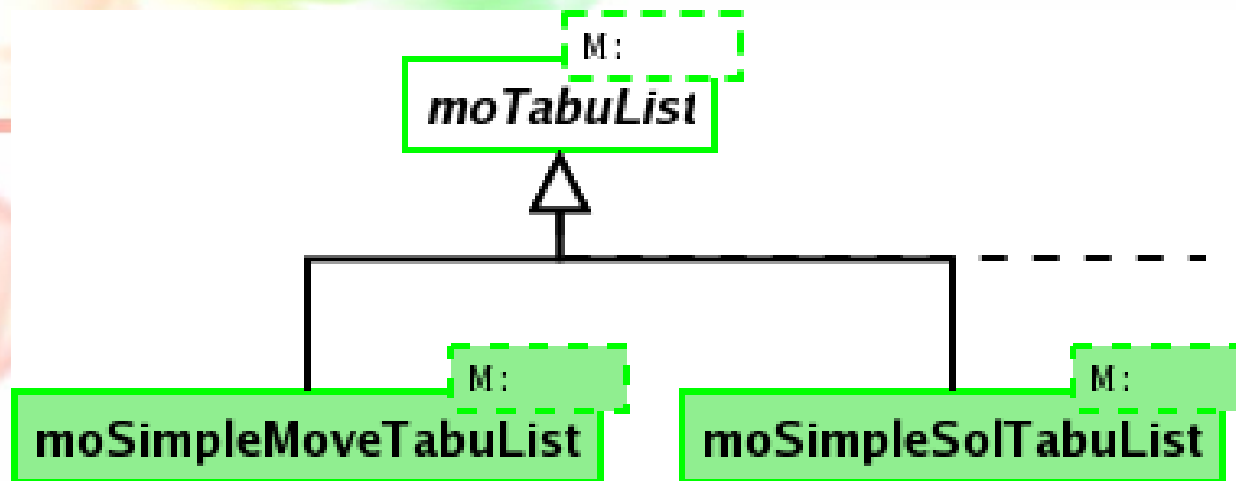
Independent
of the tackled
problem

Core classes



Designing/implementing the Tabu List

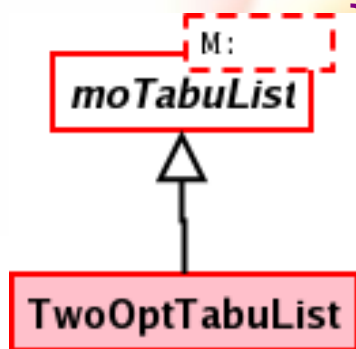
- Predefined structures.
 - Hash table of tabu solutions or tabu moves storing the tenure (short term memory)



Designing/implementing the Tabu List

- Regards the two-opt in TSP, it is preferred to build another efficient and more adapted structure

```
class TwoOptTabuList : public moTabuList <TwoOpt> {  
public :  
    int operator () (const TwoOpt & __move, const Route & __route) ;  
};
```

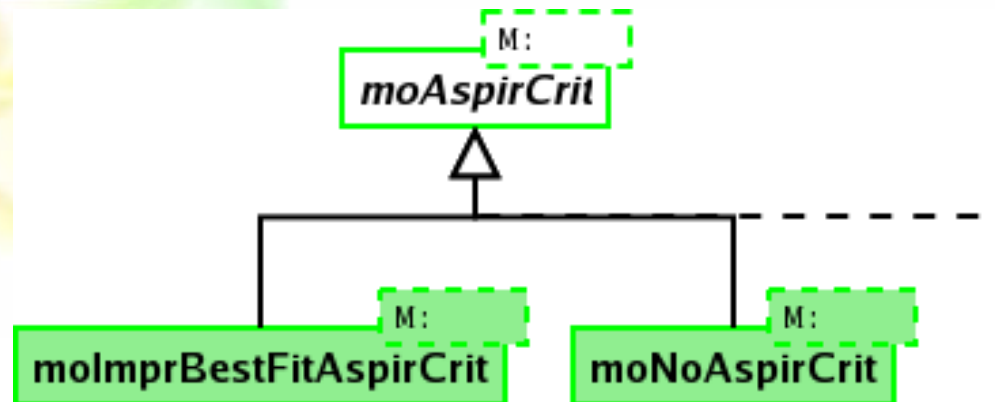


New length

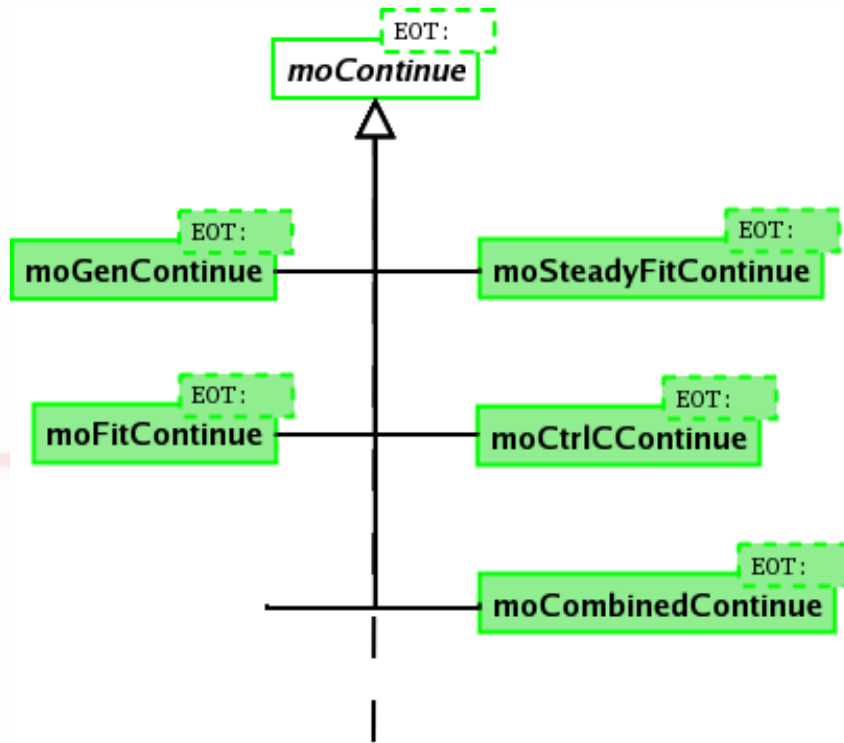
To be implemented

Choosing an aspiration criterion

- (Basic) implemented strategies
 - No aspiration criterion,
 - A tabu move builds a new solution that updates the best solution found during the search.



Choosing a continuation criterion



- Same strategies as those provided in the design of an Evolutionary Algorithm
 - An optimum is reached,
 - A given total number of iterations
 - A given number of gen. without improvement, etc



Implementing a Tabu Search

```
Route route; /* One solution */
RouteInit route_init; /* Its builds random routes */
route_init (route); /* Building a random starting solution */

RouteEval full_route_eval; /* Full route evaluator */
TwoOptInit two_opt_init; /* Initializing the first couple of edges to swap */
TwoOptNext two_opt_next; /* Updating a movement */
TwoOptIncrEval two_opt_incr_eval; /* Efficiently evaluating a given neighbor */

moNoAspirCrit <TwoOpt> two_opt_aspir_crit; /* Aspiration criterion */
TwoOptTabuList <TwoOpt> two_opt_tabu_list; /* Tabu List */

moGenContinue <TwoOpt> cont (1000); /* A fixed number of iter. */

/* Building the Tabu Search from those components */
moTS <TwoOpt> ts (two_opt_init, two_opt_next, two_opt_incr_eval,
two_opt_aspir_crit, two_opt_tabu_list, cont, full_route_eval);

/* It applies the TS to the solution */
ts (route);
```