8 Discrete Particle Swarm Optimization, illustrated by the Traveling Salesman Problem

Maurice Clerc

8.1 Introduction

The classical Particle Swarm Optimization is a powerful method to find the minimum of a numerical function, on a continuous definition domain. As some binary versions have already successfully been used, it seems quite natural to try to define a framework for a discrete PSO. In order to better understand both the power and the limits of this approach, we examine in detail how it can be used to solve the well known Traveling Salesman Problem, which is in principle very "bad" for this kind of optimization heuristic. Results show Discrete PSO is certainly not as powerful as some specific algorithms, but, on the other hand, it can easily be modified for any discrete/combinatorial problem for which we have no good specialized algorithm.

8.2 A few words about "classical" PSO

The basic principles in "classical" PSO are very simple. A set of moving particles (the swarm) is initially "thrown" inside the search space. Each particle has the following features.

1. It has a position and a velocity.
2. It knows its position, and the objective function value for this position.
3. It remembers its best previous position found so far.
4. It knows its neighbours' best previous position and objective function value (variant: current position and objective function value). See below for neighbour definition.

From now on, we will consider a particle to be one of its own neighbours, and so 3 and 4 can be combined.

G. C. Onwubolu et al., *New Optimization Techniques in Engineering*

There are many ways to define a "neighbourhood " [1], which is set of particles related to a given one, but we can distinguish two classes.

- "Physical" neighbourhood, which takes distances into account. In practice, distances are recomputed at each time step, which is quite costly, but some clustering techniques need this information.
- "Social" neighbourhood, which just takes "relationships" into account. In practice, for each particle, its neighbourhood is defined as a list of particles at the very beginning, and does not change. Note that, when the process converges, a social neighbourhood becomes a physical one.

At each time step, the behaviour of a given particle is a compromise between three possible choices:

- to follow its own way,
- to go towards its best previous position,
- to go towards the best neighbour's best previous position, or towards the best neighbour (variant).

This compromise is formalized by the following equations:

$$\begin{cases} v_{t+1} = c_1 v_t + c_2 \left(p_{i,t} - x_t \right) + c_3 \left(p_{g,t} - x_t \right) \\ x_{t+1} = x_t + v_{t+1} \end{cases} \tag{8.1}$$

where

v_t := velocity at time step t
x_t := position at time step t
$p_{i,t}$:= best previous position, found so far, at time step t
$p_{g,t}$:= best neighbour's previous best position, at time step t
c_1, c_2, c_3 := social/cognitive confidence coefficients

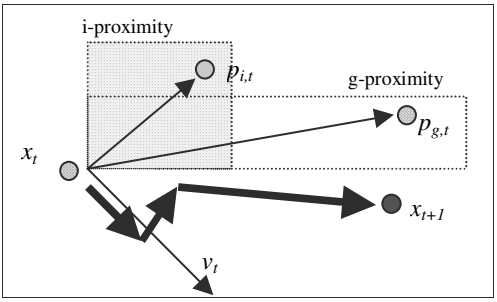


Fig. 1. Weighted combination of three possibles moves

The three social/cognitive coefficients respectively quantify:

- how much the particle trusts *itself now*,
- how much it trusts its *experience*,

- how much it trusts its *neighbours*.

It is important to note that the social/cognitive coefficients are usually randomly chosen within some given intervals, at each time step, and *for each velocity component*. It means a rule like "to go towards its best previous position" should be understood as "to go towards an area (proximity area) around of its best previous position". In classical PSO, these proximity areas are hyperparallelepids.

Of course, much works have been done to study and generalize this method [2, 3, 4, 5, 6, 7, 8]. In particular, in the following discussion, I use a "nohope/rehope" technique as defined in [9], and the convergence criterion proved in [10].

8.3 Discrete PSO

So, what do we really need for using PSO ?

- A search space of positions/states $S = \{s_i\}$
- A cost/objective function *f* on *S*, mapping *S* on a set of values $S \xrightarrow{f} C = \{c_i\}$, and the minimums of *f* are the solution states.
- An order on *C*, or, more generally, a semi-order, so that for every pair of elements of C (c_i, c_j), we can say we have either $c_i < c_j$ or $c_i \geq c_j$.
- If we want to use a physical neighbourhood, we also need a distance *d* in the search space.

Usually, *S* is a real space R^D , and *f* a numerical continuous function. However, Eq. 8.1 does not imply that this must be the case. In particular, *S* may be a finite set of states and *f* a discrete function, and, as soon as you are able to define the following basic mathematical objects and operations, you can use PSO:

- position of a particle
- velocity of a particle
- subtraction $(position, position) \xrightarrow{\text{minus}} velocity$
- external multiplication $(real\_number, velocity) \xrightarrow{\text{times}} velocity$
- addition $(velocity, velocity) \xrightarrow{\text{plus}} velocity$
- move $(position, velocity) \xrightarrow{\text{move (plus)}} position$

To illustrate this assertion, I have written yet-another-traveling-salesman-algorithm. This choice is intentionally very far from the "usual" use of PSO, so that we can have an idea of the power but also of the limits of this approach. The aim is certainly not to compete with powerful dedicated algorithms like LKH [11], but mainly to say something like "If you do not have a specific algorithm for your discrete optimization problem, use PSO: it works". We are in a finite case, so, to anticipate any remark concerning the NFL (No Free Lunch) theorem [12], let us say immediately it does not hold here for at least two reasons: the number of possible objective functions is infinite and the algorithm does use some specific in-

formation about the chosen objective function by modifying some parameters (and even, optionally, uses several different objective functions).

Note that some discrete PSO versions have already been defined and used successfully[13, 14], in particular an improved version of this "PSO for TSP" [15].

8.4 PSO elements for TSP

8.4.1 Positions and state space

Let $G = \{E_G, V_G\}$ be the weighted graph in which we are looking for Hamiltonian cycles. E_G is the set of weighted edges (arcs) and V_G the set of vertices (nodes). Graph nodes are numbered from 1 to *N*, so each element of V_G can be seen as just a label $i, i \in \{1,...,N\}$. Each element of E_G is then a triplet $(i, j, w_{i,j}), i \in V_G, j \in V_G, w_{i,j} \in R^+$. As we are looking for cycles, we can consider just sequences of *N*+1 nodes, all different, except the last one equal to the first one. Such a sequence is here called a *N*-cycle and seen as a "position". So the search space is defined as follows: the finite set of all *N*-cycles.

8.4.2 Objective function

Let us consider a position like $x = (n_1, n_2, \ldots, n_N, n_{N+1}), n_i \in V_G, n_1 = n_{N+1}$. It is "acceptable" only if all arcs (n_i, n_{i+1}) do exist. In the graph, each existing arc has a value. In order to define the "cost" function, a classical way it to just complete the graph, and to create all non existent arcs with an arbitraty value l_{sup} large enough to ensure no solution could contain such a "virtual" arc, for example

$$\begin{cases} l_{\text{sup}} > l_{\max} + (N-1)(l_{\max} - l_{\min}) \\ l_{\max} = MAX(w_{i,j}) \\ l_{\min} = MIN(w_{i,j}) \end{cases} \tag{8.2}$$

So, each arc (n_i, n_{i+1}) has a value, a real one or a "virtual" one. Now, at each position, a possible objective function can be simply defined by

$$f(x) = \sum_{i=1}^{N-1} w_{n_i, n_{i+1}} \tag{8.3}$$

This objective function has a finite number of values and its global minimum is indeed reached at a position (*N*-cycle) that is the best solution.

8.4.3 Velocity

We want to define an operator *v* which, when applied to a position during one time step, gives another position. So, here, it is a permutation of *N* elements, that is to say a list of transpositions. Let $\|v\|$ be the length of this list. A *velocity* is then defined by

$$v = ((i_k, j_k)), i_k \in V_G, j_k \in V_G, k = 1,...,\|v\| \tag{8.4}$$

or, in short $v = ((i_k, j_k))$, which means "exchange nodes (i_1, j_1), then nodes (i_2, j_2), etc. and at last nodes $(i_{\|v\|}, j_{\|v\|})$". Note that we need two parentheses, for it is a list of pairs.

Two such different lists can generate the same result when applied independently to any position. We call such velocities *equivalent* and use the notation $\cong$ to express it. Thus, if v_1 and v_2 are equivalent, we say $v_1 \cong v_2$. For example, we have $((1,3),(2,5)) \cong ((2,5),(1,3))$. In fact, in this example, they are not only equivalent, they are opposite (see below): when using velocities to move on the search space, this one is like a « sphere »: you can reach the same point following two opposite paths. A null velocity is a velocity equivalent to $\varnothing$, the empty list.

8.4.4 Opposite of a velocity

It is defined by

$$\neg v = ((i_{\|v\|-k+1}, j_{\|v\|-k+1})) \tag{8.5}$$

This formula means "to do the same transpositions as in *v*, but in reverse order". It is easy to verify that we have $\neg\neg v = v$ (and $v \oplus \neg v \cong \varnothing$, see below Addition "velocity plus velocity").

8.4.5 Move (addition) "position *plus* velocity"

Let *x* be a position and *v* a velocity. The position $x' = x + v$ is found by applying the first transposition of *v* to *x*, then the second one to the result etc.

8.4.5.1 Example

$$\begin{cases} x = (1,2,3,4,5,1) \\ v = ((1,2),(2,3)) \end{cases} \tag{8.6}$$

Applying *v* to *x*, we obtain successively

$$\begin{matrix}(2,1,3,4,5,2)\\(3,1,2,4,5,3)\end{matrix} \tag{8.7}$$

8.4.6 Subtraction "position *minus* position"

Let x_1 and x_2 be two positions. The difference $x_2 - x_1$ is defined as the velocity v, found by a given algorithm, so that applying v to x_1 gives x_2. The condition "found by a given algorithm" is necessary, for, as we have seen, two velocities can be equivalent, even when they have the same size. In particular, the algorithm is chosen so that we have $x_2 - x_1 = \neg(x_1 - x_2)$, and $x_1 = x_2 \Rightarrow v = x_2 - x_1 = \varnothing$.

8.4.7 Addition "velocity *plus* velocity"

Let v_1 and v_2 be two velocities. In order to compute $v_1 \oplus v_2$ we define the list of transpositions which contains first the elements (pairs, or transpositions) of v_1, followed by the elements of v_2. Optionally, we "contract" it to obtain a smaller equivalent velocity. In particular, this operation is defined so that $v \oplus \neg v = \varnothing$. Then, we have $\|v_1 \oplus v_2\| \le \|v_1\| + \|v_2\|$, but we usually do *not* have $v_1 \oplus v_2 = v_2 \oplus v_1$.

8.4.8 Multiplication "coefficient *times* velocity"

Let c be a real coefficient and v be a velocity. There are several cases, depending on the value of c.

8.4.8.1 Case c = 0

We have $cv = \varnothing$.

8.4.8.2 Case c ∈]0,1]

We just "truncate" v. Let $\|cv\|$ be the greatest integer smaller than or equal to $c\|v\|$. So we define $cv = ((i_k, j_k)), k = 1,...,\|cv\|$.

8.4.8.3 *Case* c > 1

It means we have $c = k + c', k \in \mathbf{N}^*, c' \in [0,1[$. So we can define $cv = \underbrace{v \oplus v \oplus \ldots \oplus v}_{k \text{ times}} \oplus c'v$.

8.4.8.4 *Case c < 0*

By writing $cv = (-c)\dashv v$, we just have to consider one of the previous cases. Note that we have $v_1 \cong v_2 \Rightarrow cv_1 \cong cv_2$ if c is an integer, but it is usually not true in the general case.

8.4.9 Distance between two positions

Let x_1 and x_2 be two positions. The distance between these positions is defined by $d(x_1, x_2) = \|x_2 - x_1\|$. Note: it is a metric, for we do have (x_3 is any third position):

$$\|x_2 - x_1\| = \|x_1 - x_2\|$$
$$\|x_2 - x_1\| = 0 \Leftrightarrow x_1 = x_2$$
$$\|x_2 - x_1\| \leq \|x_2 - x_3\| + \|x_3 - x_1\|$$

8.5 The algorithm "PSO for TSP". Core and variations

8.5.1 Equations

We can now rewrite Eq. 8.1 as follows:

$$\begin{cases} v_{t+1} = c_1 v_t \oplus c_2 (p_{i,t} - x_t) \oplus c_3 (p_{g,t} - x_t) \\ x_{t+1} = x_t + v_{t+1} \end{cases} \tag{8.8}$$

In practice, we assume $c_3 = c_2$. No experiment has found that a different choice gives significantly better result on some class of problems. More precisely, for a given problem, you may find a better choice after a lot of trials, but there is no known rule to find it "in advance". Therefore, choosing $c_3 = c_2$ doesn't change the algorithm in any significant way. By defining an intermediate position $p_{ig,t} = p_{i,t} + (p_{g,t} - p_{i,t})/2$, we finally use the following system:

$$\begin{cases} v_{t+1} = c_1 v_t \oplus c_2' \left(p_{ig,t} - x_t \right) \\ x_{t+1} = x_t + v_{t+1} \end{cases} \quad (8.9)$$

The advantage of this equation is that it can be used as a guideline for choosing "good" coefficients [10], even if the proofs given in this article is applied to differentiable systems. It is indeed possible to use it just like that, but with such a straightforward and simple approach, the swarm size may need be quite big to avoid getting stuck at local minimums. Thus, some modifications of the core algorithm are quite helpful to improve performances. In particular, the NoHope/ReHope process can be very powerful, which we describe in the next section.

8.5.2 NoHope tests

8.5.2.1 Criterion 0

If a particle has to move "towards" another one which is at distance 1, either it does not move at all, or it goes to exactly at the same position as the second one, depending on the social/confidence coefficients. When the swarm size gets smaller than $N(N\text{-}1)$, it may happen that all moves computed according to Eq. 8.9 are null. In this case there is absolutely no hope to improve the current best solution.

8.5.2.2 Criterion 1

The NoHope test defined in [9] is "Is the swarm to small?". This demand a computation of the swarm diameter at each time step, which is expensive. However, in a discrete case like the one being presented here, as soon as the distance between two particles tend to become "too small", the particles become identical (usually first by positions and then by velocities). So, at each time step, a "reduced" swarm is computed, in which all particles are different, which is not very expensive, and the NoHope test becomes "Is the swarm too reduced?", say by 50%.

8.5.2.3 Criterion 2

Another criterion has been added: "Is the swarm too slow?". This is done by comparing the velocities of all particles to a threshold, either individually or globally. In one version of the algorithm, this threshold is in fact modified at each time step, according to the best result obtained so far and to the statistical distribution of arc values.

8.5.2.4 Criterion 3

Another very simple criterion is defined: "No improvement for too many time steps". However, in practice, it appears that criteria 1 and 2 are sufficient.

8.5.3 ReHope process

As soon as there is "no hope", the swarm is re-expanded. The first two methods used here are inspired by the ones described in [9] and [16].

8.5.3.1 Lazy Descent Method (LDM)

Each particle goes back to its previous best position and, from there, moves randomly and slowly ($\|v\| = 1$) and stops *as soon as* it finds a better position or when a maximal number of moves $M_{\max}$ is reached (in the examples below $M_{\max} = N$). If the current swarm is smaller than the initial one, it is completed by a new set of randomly chosen particles.

8.5.3.2 Energetic Descent Method (EDM)

Each particle goes back to its previous best position and, from there, moves slowly ($\|v\| = 1$) *as long as* it finds a better position in at most $M_{\max}$ moves. If the current swarm is smaller than the initial one, it is completed by a new set of particles randomly chosen. You may find, by chance, a solution, but it is more expensive than LDM. Ideally, it should be used only if the combination Eq. 8.9 + LDM seems to fail.

8.5.3.3 Local Iterative Levelling (LIL)

This method is more powerful … and more expensive. In practice, it should be used only when we *know* there is a better solution, but the combination Eq. 8.9 + EDM fails to find it.

The idea is as follows. There are infinitely possible objective functions with the same global minimum, so we can locally and temporarily use any of them to guide a particle. For each immediate physical neighbour y (at distance 1) of the particle x, a temporary objective function value $f_t(y)$ is computed by using the following algorithm.

LIL algorithm

- Find all neighbours at distance 1.
- Find the best one, $y_{\min}$.

- Assign to y the temporary objective function value $f_t(y) = (f(y_{\min}) + f(x))/2$ and, according to these temporary evaluations, x moves to its best immediate neighbour.

In TSP, the cost of such an algorithm is $O(N^2)$. If repeated too often, it considerably increases the total cost of the process. That is why it should be used only if there is really "no hope", and if you do want the best solution. In practice, the algorithm will usually find a good one, even without using LIL.

8.5.4 Adaptive ReHope Method (ARM)

The four methods described above (NoHope ReHope/LDM/EDM/LIL) can be used automatically in an adaptive way, depending on how many time steps has passed from the last improvement of the solution. A possible strategy is given in the following table.

Table 8.1. A possible ReHope strategy

Number of time steps without improvement	ReHope type
0-1	No ReHope
2-3	Lazy Descent Method
4	Energetic Descent Method
>4	Local Iterative Levelling

8.5.5 Queens

Instead of using the best neighour/neighbour's previous best for each particle, we can use an extra particle, which "summarizes" the neighbourhood. This method is a combination of the (unique) queen method defined in [9] and the multi-clustering method described in [17]. For each neighbourhood, we iteratively build a centroid and take it as the best neighbour ($p_{g,t}$ in Eq. 8.8).

8.5.6 Extra-best particle

In order to speed up the process, the algorithm can also use a special extra particle that stores the best position found so far. It is not absolutely necessary, for this position is also memorized as "previous best" in at least one particle, but it may avoid a whole sequence of iterations between two ReHope processes.

8.5.7 Parallel and sequential versions

The algorithm can run either in (simulated) parallel mode or in sequential mode. In the parallel mode, at each time step, new positions are computed for all particles and then the swarm is globally moved. In sequential mode, each particle is moved at a time: particle 1, then particle 2, … then particle *N*, then again particle 1, etc. Note that Eq. 8.9 implicitly supposes a parallel mode, but in practice there is no clear difference in performances on a given sequential machine, and the second method is a bit less expensive.

8.6 Examples and results

8.6.1 Parameters choice

The purpose of this paper is mainly to show how PSO can be used, in principle, on a discrete problem. For that reason, we use some default values for the parameters. Of course, a better set of parameters can certainly be found by "optimizing the optimization" (in fact, PSO itself may be used to find an optimum in a parameter space).

8.6.1.1 Social/cognitive coefficients

If nothing else is explicitly mentioned, in all the examples we use $c_1 \in]0,1[$ (typically 0.5 if NoHope/ReHope is used or 0.999 if not). c_2 is randomly chosen at each time step in $[0,2]$. The convergence criterion defined in [10] is satisfied. This criterion is proved only for differentiable objective functions, but it is not unreasonable to think it should work here too. At least, in practice, it does.

8.6.1.2 Swarm size and neighbourhood size

Ideally, the best swarm and neighbourhood sizes depend on the number of local minimums of the objective function, say $n_{\text{local\_min}}$, so that the swarm can have sub-swarms around each of these minimums. Usually, we simply do not have this information, except the fact that there are at most *N*-1 such minimums. A possible way is to use a very small swarm at the beginning, just to have an idea of the landscape defined by the objective function (see the example below). Also, a statistical estimation of $n_{\text{local\_min}}$ can be made, using the arc values distribution. In practice, we usually use a swarm size S equal to *N*-1 (see below Structured Search Map).

Some considerations, not completely formalized, about the fact that each particle uses three velocities to compute a new one, indicate that a good neighbourhood size should be simply 4 (including the particle itself).

But all this is still "rules of thumb", and this is the main limitation of the approach: on a given example, we do not know in advance what are the best parameters and usually have to try different values. That is why, in particular, some adaptive PSO versions are now under development.

8.6.1.3 Performance criteria

Normally, the performance measures are reported in literature with too specific data. For example, "The program took 3 seconds on XYZ machine to solve instance xxxx from TSPLIB". Unfortunately, such data is not easy to compare with another performance measure which uses machine ZYX. Thus, we think it is better to have a measure like the following: "it needed 7432 tour evaluations to reach a solution".

However, completely compute the objective function value on a *N*-cycle or to update it after having swapped two nodes is not all the same: the first case is about *N*/4 times more expensive. So we use two criteria: the number of position evaluations *and* the number of arithmetic/logical operations. It is still not perfect, for the same algorithm can be written in different ways even in the same language, one more efficient, and the other less. However, if you know your machine is a xxx Mips computer, you can estimate how long it would take to run the example.

8.6.2 A toy example as illustration

In this example, we use here a very simple graph (17 nodes) from TSPLIB, called br17.atsp (cf. Appendix). The minimal objective function value is 39, and there are several solutions. Although it has only a few nodes, it is designed so that finding one of the best tours is not so easy, so it is nevertheless an interesting example.

8.6.2.1 What the landscape looks like

We define randomly a small swarm of, say, five particles, where p_{best} is the best one. For each other particle p_i, we now consider the linear sequence $p_k = p_i + k\,(p_{best} - p_1)/(N-1), k = 0,...,N-1$, and plot the graph $(k/(N-1), f(p_{best}) - f(p_k))$. It give us an idea of the landscape (see Fig. 2 and Fig. 3). The landscape is clearly quite chaotic, so we will surely need to use the NoHope/ReHope process quite often. Also, the Queen option is probably not efficient (a centroid does not give a better position).

Fig. 2. One section of the search space landscape

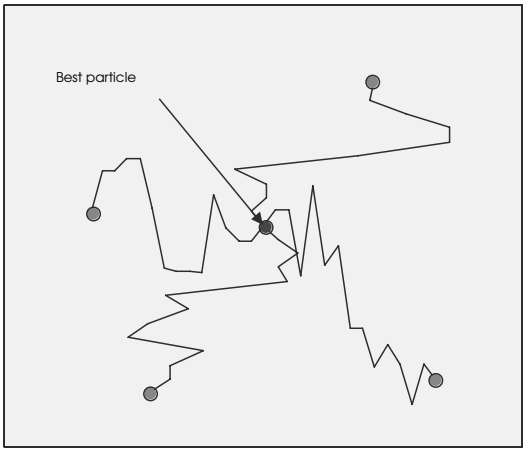


Fig. 3. Main sections for a 5-swarm

8.6.2.2 Structured Search Map. How the swarm moves

Let us suppose we know a solution x_{sol}. We consider all possible positions in the search space, and plot the map $(d(x_{sol},x), f(x))$. On this map, at each time step, we highlight the positions of the particles, so that we can see how it moves. We define some equivalence classes according to the equivalence relation $x \cong x' \Leftrightarrow d(x_{sol},x) = d(x_{sol},x')$. As we can see, not only the swarm globally moves towards the solution, but some particles also tend to move towards the minimum (in terms of objective function value) of the equivalence classes. It means that even if it does not find the best solution, it finds at least (and usually quite rapidly) interesting quasi-solutions. It also means that if the swarm is big

enough, we may simultaneously find several solutions (this property is used in multiple objective optimization [18, 19]).

In our example, as the number of positions is quite big ($16! \approx 21^{12}$), we plot only a few of them (4700), as the "background" of the diagram sequence of Fig. 4 and 5. We clearly see that the example has been designed to be a bit difficult, for there is a "gap" between distances 4 and 8. Here, we use a swarm of 16 particles, and it finds three solutions in one "standard" run, and some others if we modify the parameters (see Appendix).

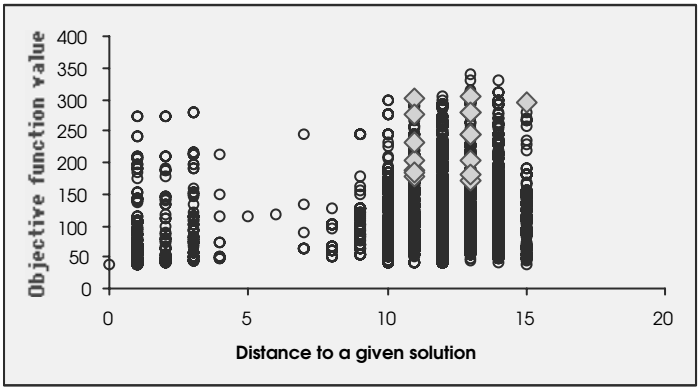
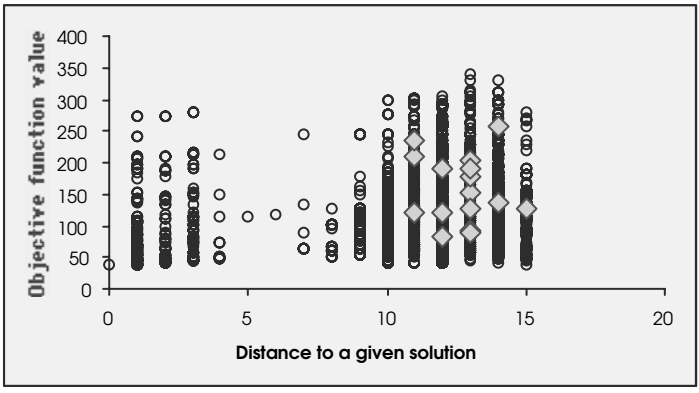
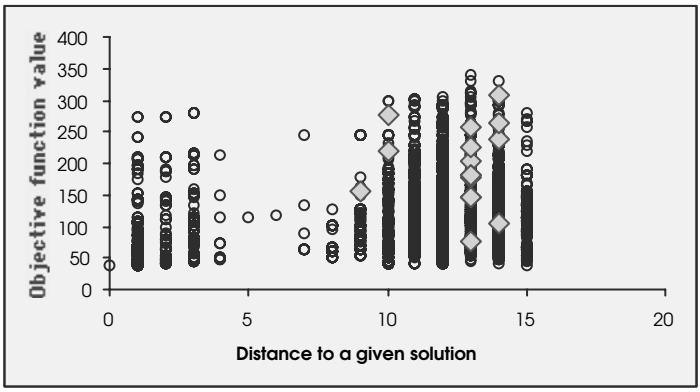
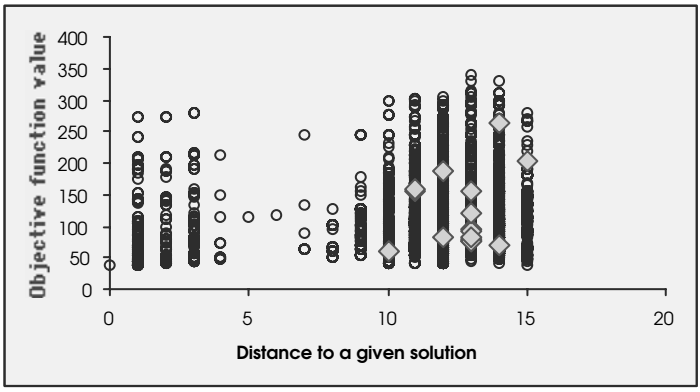
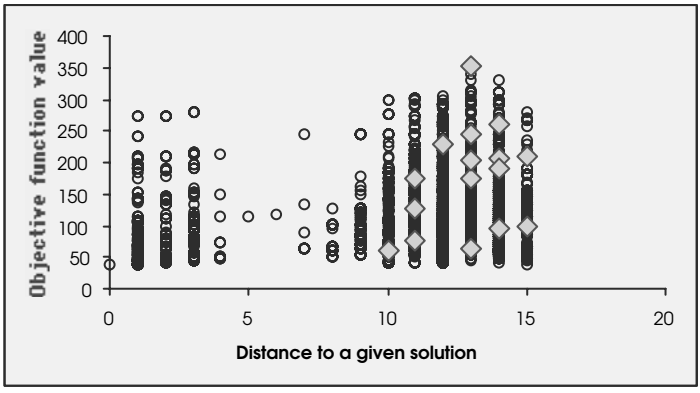


Fig. 4. Initial position on the Structured Search Map (swarm size = 16)




Objective function value
400
350
300
250
200
150
100
50
0
0
5
10
15
20
Distance to a given solution


Objective function value
400
350
300
250
200
150
100
50
0
0
5
10
15
20
Distance to a given solution


Objective function value
400
350
300
250
200
150
100
50
0
0
5
10
15
20
Distance to a given solution

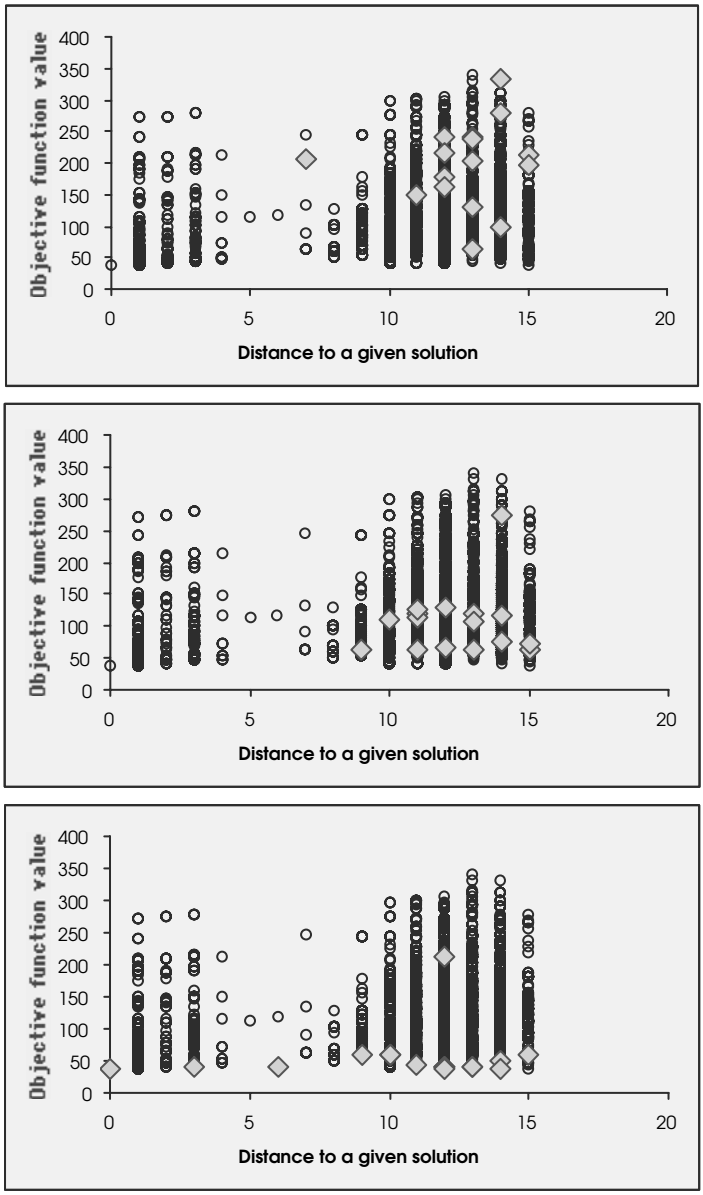


Fig. 5. How the swarm moves on the Structured Search Map (some times step between 1 and 480). It takes some times to jump the gap betwee distance 9 and 4

8.6.3 Some results, and discussion

Table 8.2. Some result using Discrete PSO on graph br17.atsp

Swarm size	Hood size	c_1	Random c_2	ReHope type	Best solution
16	4	0.500	]0,2]	ARM	39 (3 solutions)
16	4	0.500	]0,2]	ARM	39
16	4 with queens	0.500	]0,2]	ARM	39 (3 solutions)
8	4	0.500	]0,2]	ARM	39
1	1	0.500	]0,2]	ARM	44
128	4	0.999	]0,2]	no	47
32	4	0.999	]0,2]	no	66
16	4	0.999	]0,2]	no	86

Table 8.2 (cont.)

Hood type	Tour evaluations	Arithmetical/logical operations
social	7990	4.8 M
physical	7742	6.3 M
social or physical	9051	5.8 M
social	4701	2.8 M
social	926 to ∞	0.6 M
social	41837 to ∞	33.2 M to ∞
social	14351 to ∞	10.8 M to ∞
social	2880 to ∞	1.9 M to ∞

As we can see from Table 8.2:

- Using "physical" neighbourhood may need less tour evaluations, but is much more expensive than "social" option, if we consider the total number of arithmetical/logical operations (distances have to be recaculated at each time step). The fact that with social neighbourhood you find more solutions is not a general rule.
- Using only ReHope methods (in this case, using s particles for T time steps is equivalent to using just one particle for sT time steps) is not enough to reach the best solution.
- Using only core algorithm, with no ReHope at all, is not enough to reach the best solution, unless, probably, you use a huge swarm.
- Using queens is not interesting.

Results are not particularly good nor bad, mainly because we use a generic ReHope method which is not specifically designed for TSP. However, this is done on

purpose. This way, by just changing the objective function, you could use Discrete PSO for different problems.

Appendix

Graph br17.atsp (from TSPLIB)

In the original data, diagonal values are equal to 9999. Here, they are equal to 0. For PSO algorithm, it changes nothing, and it simplifies the visualization program.

```
NAME: br17
TYPE: ATSP
COMMENT: 17_city_problem_(Repetto)_Opt.:_39
DIMENSION: 17
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: FULL_MATRIX
EDGE_WEIGHT_SECTION
        0  3  5 48 48  8  8  5  5  3  3  0  3  5  8  8  5
        3  0  3 48 48  8  8  5  5  0  0  3  0  3  8  8  5
        5  3  0 72 72 48 48 24 24  3  3  5  3  0 48 48 24
       48 48 74  0  0  6  6 12 12 48 48 48 48 74  6  6 12
       48 48 74  0  0  6  6 12 12 48 48 48 48 74  6  6 12
        8  8 50  6  6  0  0  8  8  8  8  8  8 50  0  0  8
        8  8 50  6  6  0  0  8  8  8  8  8  8 50  0  0  8
        5  5 26 12 12  8  8  0  0  5  5  5  5 26  8  8  0
        5  5 26 12 12  8  8  0  0  5  5  5  5 26  8  8  0
        3  0  3 48 48  8  8  5  5  0  0  3  0  3  8  8  5
        3  0  3 48 48  8  8  5  5  0  0  3  0  3  8  8  5
        0  3  5 48 48  8  8  5  5  3  3  0  3  5  8  8  5
        3  0  3 48 48  8  8  5  5  0  0  3  0  3  8  8  5
        5  3  0 72 72 48 48 24 24  3  3  5  3  0 48 48 24
        8  8 50  6  6  0  0  8  8  8  8  8  8 50  0  0  8
        8  8 50  6  6  0  0  8  8  8  8  8  8 50  0  0  8
        5  5 26 12 12  8  8  0  0  5  5  5  5 26  8  8  0
```

Some solutions found by Discrete PSO

Each line of Table 8.3 gives the sequence of nodes of a minimum tour (common total weight=39). For example, according to the above matrix, the first tour gives the following weights: 5+0+3+0+0+0+8+0+0+0+6+0+12+0+0+5+0=39.

Table 8.3. Some solutions for br17.atsp

3	14	11	13	2	10	15	7	6	16	4	5	9	17	8	12	1
6	7	15	16	5	4	17	9	8	10	2	13	11	14	3	12	1
7	6	15	16	5	4	9	8	17	10	11	13	2	14	3	12	1
9	8	17	5	4	15	6	7	16	13	10	11	2	3	14	12	1
12	3	14	10	2	13	11	17	8	9	5	4	7	6	16	15	1
12	3	14	10	11	2	13	17	8	9	4	5	16	6	15	7	1
12	6	7	15	16	5	4	9	17	8	2	11	13	10	14	3	1
12	8	9	17	5	4	16	15	7	6	10	2	11	13	14	3	1
12	14	3	2	10	11	13	17	9	8	5	4	15	6	7	16	1
12	14	3	11	2	10	13	15	6	16	7	4	5	9	8	17	1
12	15	16	7	6	5	4	8	17	9	11	10	2	13	14	3	1
12	16	15	7	6	4	5	9	8	17	10	2	11	13	14	3	1
12	16	15	7	6	5	4	8	9	17	10	11	2	13	14	3	1
12	16	7	6	15	5	4	9	17	8	11	10	13	2	14	3	1
12	17	8	9	5	4	15	16	7	6	2	10	11	13	3	14	1
12	17	8	9	5	4	15	16	7	6	2	10	11	13	3	14	1
14	3	10	13	11	2	17	8	9	4	5	15	6	16	7	12	1
14	3	10	13	11	2	8	17	9	4	5	15	6	7	16	12	1
14	3	10	11	2	13	8	17	9	4	5	15	7	6	16	12	1
16	6	7	15	4	5	9	8	17	13	11	10	2	3	14	12	1
16	6	7	15	4	5	9	8	17	13	10	2	11	3	14	12	1
17	9	8	4	5	7	6	16	15	10	11	2	13	14	3	12	1
12	14	3	2	10	13	11	16	6	7	15	4	5	17	9	8	1

References

[1] Kennedy J., "Small Worlds and Mega-Minds: Effects of Neighborhood Topology on Particle Swarm Performance", *Congress on Evolutionary Computation*, Washington D.C., 1999, p. 1931-1938.

[2] Angeline P. J., "Using Selection to Improve Particle Swarm Optimization", *IEEE International Conference on Evolutionary Computation*, Anchorage, Alaska, May 4-9, 1998, p. 84-89.

[3] Eberhart R. C., Kennedy J., "A New Optimizer Using Particle Swarm Theory", *Sixth International Symposium on Micro Machine and Human Science*, Nagoya, Japan, 1995, p. 39-43.

[4] Kennedy J., Eberhart R. C., "Particle Swarm Optimization", *IEEE International Conference on Neural Networks*, Perth, Australia, 1995, p. 1942-1948.

[5] Kennedy J., "The Particle Swarm: Social Adaptation of Knowledge", *International Conference on Evolutionary Computation*, Indianapolis, Indiana, 1997, p. 303-308.

[6] Kennedy J., "The behavior of particles", *Evolutionary VII*, San Diego, CA, 1998, p. 581-589.

[7] Shi Y., Eberhart R. C., "Parameter Selection in Particle Swarm Optimization", *Evolutionary Programming VII*, 1998,

[8] Shi Y. H., Eberhart R. C., "A Modified Particle Swarm Optimizer", *International Conference on Evolutionary Computation*, Anchorage, Alaska, May 4-9, 1998, p. 69-73.

[9] Clerc M., "The Swarm and the Queen: Towards a Deterministic and Adaptive Particle Swarm Optimization", *Congress on Evolutionary Computation*, Washington DC, 1999, p. 1951-1955.

[10] Clerc M., Kennedy J., "The Particle Swarm-Explosion, Stability, and Convergence in a Multidimensional Complex Space", *IEEE Transactions on Evolutionary Computation*, vol. 6, 1, 2002, p. 58-73.

[11] Helsgaun K., An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic, Department of Computer Science,Roskilde University, Denmark, 1997.

[12] Wolpert D. H., Macready W. G., No Free Lunch for Search, The Santa Fe Institute, 1995.

[13] Kennedy J., Eberhart R. C., "A discrete binary version of the particle swarm algorithm", *Conference on Systems, Man, and Cybernetics*, 1997, p. 4104-4109.

[14] Yoshida H., Kawata K., Fukuyama Y., "A Particle Swarm Optimization for Reactive Power and Voltage Control considering Voltage Security Assessment", *IEEE Trans. on Power Systems*, vol. 15, 4, 2001, p. 1232-1239.

[15] Secrest B. R., Lamont G. B., "Communication in Particle Swarm Optimization Illustrated by the Travelling Salesman Problem", *Workshop on Particle Swarm Optimization*, Indianapolis, IN: Purdue School of Engineering and Technology, 2001,

[16] He Z., Wei C., Jin B., Pei W., Yang L., "A New Population-based Incremental Learning Method for the Traveling Salesman Problem", *Congress on Evolutionary Computation*, Washington D.C., 1999, p. 1152-1156.

[17] Kennedy J., "Stereotyping: Improving Particle Swarm Performance With Cluster Analysis", *Congress on Evolutionary Computation,*, 2000, p. 1507-1512.

[18] Coello Coello C. A., Toscano Pulido G., Lechuga M. S., Handling Multiple Objectives with Particle Swarm Optimization, EVOCINV-02-2002, CINVESTAV, Evolutionary Computation Group, 2002.

[19] Hu X., Eberhart R. C., "Multiobjective Optimization Using Dynamic Neighborhood Particle Swarm Optimization", *Congress on Evolutionary Computation (CEC'2002)*, Piscataway, New Jersey, 2002, p. 1677-1681.