Binary Particle Swarm Optimisers: toolbox, derivations, and mathematical insights

Maurice Clerc

Binary Particle Swarm Optimisers: toolbox, derivations, and mathematical insights

Maurice.Clerc@WriteMe.com

January 5, 2007

2005-02-02

Abstract

A canonical Particle Swarm Optimisation model requires only three algebraic operators, namely "modifying a velocity", "combining three velocities", and "applying a velocity to a position", which can have a lot of explicit transcriptions. In particular, for binary optimisation, it is possible to define a toolbox of specific ones, and to derive then some à la carte optimisers that can be, for example, extremely efficient only on some kind of problems, or on the contrary just reasonably efficient but very robust. For "amatheurs" who would like to better understand the behaviour of binary PSO algorithms an Appendix gives some theoretical results.

1 Canonical PSO and Binary model

1.1 Canonical PSO

A detailed description can be found in [CLE 04]. Let's give here just a quick one. As usually, in a search space of dimension D , we have the following D-vectors for a given particle:

- the position x
- the velocity v
- the best previous position p
- the best previous position g found in its informant group.

We assume here *the information links are initialised at random, and re-initialised the same way after each non efficient iteration* (i.e. the best fitness value is still the same), except below for Derivation 100 that uses the classical circular neighbourhood. More precisely, each particle informs itself and choose also at random $K-1$ particles to informs. So, conversely, it is important to note that each particle is informed by a number of particles that is not necessary equal to K .

Positions and velocities are initialised at random as usually. After that, at each time step, each move of a particle is computed by combining three tendencies:

1. keeping some diversity, i.e. modifying the velocity. Most of the time, in classical PSO, the direction is not modified
2. going back more or less towards the best previous position p, i.e. choosing a point "around" p, usually by modifying the vector $p - x$
3. going more or less towards the best previous position g of the informants, i.e. choosing a point "around" g, usually by modifying the vector $g - x$

This combination gives a new velocity that is "added" to the current position to obtain the new one. This new position itself may be updated to take some constraints into account. The most common ones are "interval constraints": for each dimension, the position component has to be in a given interval (continuous or discrete). More generally, let's define $\lfloor a \rfloor_k$ as a post-constraint k applied to the vector a.

The process can be summarised by

$$\left\{ \begin{array}{lcl} v & \leftarrow & \bigoplus \widetilde{v^{\alpha}} \widetilde{p^{\beta}} \widetilde{g^{\gamma}} \\ x & \leftarrow & x \oplus v \\ x & \leftarrow & \lfloor x \rfloor_{\eta} \\ v & \leftarrow & \lfloor v \rfloor_{\delta} \end{array} \right. \quad \textbf{or} \quad \left\{ \begin{array}{lcl} v & \leftarrow & \bigoplus \widetilde{v^{\alpha}} \widetilde{p^{\beta}} \widetilde{g^{\gamma}} \\ v & \leftarrow & \lfloor v \rfloor_{\delta} \\ x & \leftarrow & x \oplus v \\ x & \leftarrow & \lfloor x \rfloor_{\eta} \end{array} \right.$$

where

$\widetilde{a^k}$	means "modify the vector a by using a given method $\widetilde{k}$"
$\bigoplus abc$	means "combine the three vectors a , b, and c"
$a \oplus b$	means ""add" the two vectors a and b".

In the first form, the velocity is "added" to the position and constrained only *after that* for the next iteration, as in the second form, it is constrained *before* to be used to modify the position.

For example, the classical PSO equations are

$$\left\{ \begin{array}{lcl} v_d & \leftarrow & c_1 v_d + rand\,(0,c)\,(p_d - x_d) + rand\,(0,c)\,(g_d - x_d) \\ x_d & = & x_d + v_d \end{array} \right.$$

For this algorithm, we have the following operator definitions:

$\widetilde{\alpha}$	is "multiply each component by the same coefficient c_1" (the direction is then not modified)
$\widetilde{\beta}$	"applied to a , multiply each component of $a-x$ by a random number (uniform distribution) between 0 and c"
$\widetilde{\gamma}$is	identical to $\widetilde{\beta}$
$\bigoplus$	is "add the three vectors"
$\oplus$	is "add the two vectors"

1.2 Toolbox

Now we have to give some precise definitions of the different operators for binary optimisation. Note that there is always at least the interval constraint $x_d \in \{0,1\}$. In the table 1 you can see a set of possible operators. The mod^+ function is a modulo function giving only non negative values. By choosing three operators in column A, one in column B, one in column C, and two in column D, you can define a new kind of binary optimiser. For example, an algorithm A1-A1-A3-B1-C1-D1 (which is the pivot method defined below as Derivation 11) can be described as follows:

don't	use v
don't	use p
modify	at random a constant bit number of g
use	the modified g as the new position

	A:$\widetilde{a}$	B:$\oplus abc$	C:$a \oplus b$	D:$\lfloor a \rfloor$
1	nothing	c	b	a
2	a	$a+b+c$	$a+b$	$INT(a) mod^{+} 2$
3	modify at random n bits, with n randomly chosen between 1 and a constant value	probabilistic choice using $a_d b_d c_d$		$-1 + INT(a) mod 3$
4	modify at random n bits with n randomly chosen between 1 and a decreasing value	majority choice using $a_d b_d c_d$		probabilistic mapping $\begin{cases} u = \frac{1}{1+e^{-a}} \\ r = rand(0,1) \\ u > r \Rightarrow 1 \\ u \leq r \Rightarrow 0 \end{cases}$
5	multiply each bit by a different random number	probabilistic choice using $b_d c_d$		
6	multiply each bit by the same random number	majority choice using $b_d c_d$		
7	multiply each bit by the same constant number			

Table 1: Toolbox example for Binary PSO. You can define an algorithm by picking up operator definitions in each column. However, you still have to precise some details, for example the decreasing rule for the random number of bits to switch. Also some choices, like using only line 1, are completely uninteresting

Note that some choices are meaningless or not consistent. For example, with A2-A2-A2-B1-C1-D1-D1 the particles do not move at all! Also, clearly, D3 does not make sense if applied to a position, as it does if applied to a velocity, keeping it in $\{-1, 0, 1\}$. Some choices still have to be more detailed, by giving the precise rules, for example for the probabilistic choice, or the random number of bits to modify. We will see that in the Derivation section. However, we have first to decide on what kind of problems the resulting optimisers will be tested.

2 Benchmarks

2.1 Some deceptive deceptive problems

In binary optimisation, it is very easy to design some algorithms that are extremely good on some benchmarks (and extremely bad on some others). It means we have to be very careful when we choose a test function set. It is not rare that some authors use a too small benchmark set and conclude a bit too rapidly that their algorithm is an improvement. For example, the benchmark used in [ALK 02] in order to present the M-DiPSO algorithm contains only three problems (described below) that the authors call "deceptive". By comparing M-DiPSO to a genetic algorithm, the authors conclude that their method is better.

What is wrong in this approach? The point is the three problems they have chosen are in fact what we could call "deceptive deceptive problems". I do not say M-DiPSO is not good, I just say this paper does not really prove it (particularly for there are not enough details to recode the algorithm, so there is no way to run it on other problems). As we will see, these precise problems can be solved extremely quickly by some simple PSO derivations. In the following, $|y|$ denotes the sum of the bits of the string y.

2.1.1 Goldberg's order-3

The fitness f of a bit-string is the sum of the result of separately applying the following function to consecutive groups of three components each:

$$f_1(x) = \begin{cases} 0.9 \text{ if } |y| = 0 \\ 0.6 \text{ if } |y| = 1 \\ 0.3 \text{ if } |y| = 2 \\ 1.0 \text{ if } |y| = 3 \end{cases}$$

For example, if the string is $x = 010110101$, the total value is $f_1(010) + f_1(110) + f_1(101) = 0.9 + 0.3 + 0.3 = 1.5$

If the string size is D, the maximum value is obviously $D/3$, for the string $1111...111$. In practice, we will then use as fitness the value $D/3 - f$ so that the problem is now to find the minimum 0.

2.1.2 Bipolar order-6

The fitness is the sum of the result of applying the following function to consecutive groups of six components each:

$$f_2(y) = \begin{cases} 1.0 \text{ if } |y| = 0 \text{ or } 6 \\ 0.0 \text{ if } |y| = 1 \text{ or } 5 \\ 0.4 \text{ if } |y| = 2 \text{ or } 4 \\ 0.8 \text{ if } |y| = 3 \end{cases}$$

So the solutions are all combinations of sequences 6x1 **and** 6x0**. In particular,** 1111...111 **and** 0000...000 **are solutions. The maximum value is** $D/6$**.**

2.1.3 Mühlenbein's order-5

The fitness is the sum of the results of applying the following function to consecutive groups of five components each:

$$f(y) = \begin{cases} 4.0 \text{ if } y = 00000 \\ 3.0 \text{ if } y = 00001 \\ 2.0 \text{ if } y = 00011 \\ 1.0 \text{ if } y = 00111 \\ 3.5 \text{ if } y = 11111 \\ 0.0 \text{ otherwise} \end{cases}$$

So the solution is 0000...000 **and the maximum value is** $3.5D/5$ **.**

2.2 Some other problems

2.2.1 Clerc's Zebra3

As already said, some PSO binary derivations are *too* good on the three above problems. So I designed another problem, slightly modifying the Goldberg's ones, but that really deceives these derivations.

The fitness f **of a bit-string is the sum of the result of separately applying the following function to consecutive groups of three components each. If the rank of the group is even (first rank=0):**

$$f_{z3}(y) = \begin{cases} 0.9 \text{ if } |y| = 0 \\ 0.6 \text{ if } |y| = 1 \\ 0.3 \text{ if } |y| = 2 \\ 1.0 \text{ if } |y| = 2 \end{cases}$$

If the rank of the group is odd:

$f_{z3}(y) = \begin{cases} 1.0 \textbf{ if } |y| = 0 \\ 0.3 \textbf{ if } |y| = 1 \\ 0.6 \textbf{ if } |y| = 2 \\ 0.9 \textbf{ if } |y| = 2 \end{cases}$ So, the solution point is 111000111000... and the maximum value is $D/3$.Here again, we will use in practice as fitness the value $D/3 - f$ so that the problem is now to find the minimum 0.

2.2.2 Quadratic problem

It is defined by

$f(x) = |D + xQ\widetilde{x}|$

where Q is a DxD matrix.

For convenience, we can generate Q at random, with a given density of 1 values. Note that we use here the classical binary (logical) algebra. In particular we have $a^2 = a$, and $a + a = 0$. So the problem is in fact not that difficult, even for large instances, also for there usually are several solutions. Some authors do solve some instances with 9000 variables [GLO 02], on a ten processors Cray though. I don't have one at hand, just a small laptop, so the example below will be far more modest, with $D = 100$.

2.2.3 Multimodal problems

We use here the random problem generator defined in [KEN 98]. The parameters are the dimension D and the number of peaks. The minimum value is 0.

First, the peaks are randomly put on the search space. The landscape is defined by the following C code:

```
for(i = 0; i < peaks; i++)
for(j = 0; j < D; j++) landscape[i][j] = rand()&01;
```

After that, for each position x, the fitness is computed as follows:

```
f = 0.0;
for (j = 0; j < peaks; j++)
{
f1 = 0.0;
for (k = 0; k < D; k++) if (x[k] = = landscape[j][k]) f1++;
if (f1 > f) f = f1;
}
f=1-f/(double)D;
```

3 Binary PSO derivations

Actually, I have written about twenty PSO derivations for binary optimisation, for it is quite easy by using the table 1. I present here just the most interesting ones (their code number is purely arbitrary, and referring to the C source code you can download from my site Math stuff about PSO `http://clerc.maurice.free.fr/pso/index.htm`**).**

3.1 Derivation 0

According to the table 1 its codification is A1-A5-A5-B2-C2-D2-D3. For this first one, let's give a more complete explanation. As a x component is either 0 or 1, to find another value we can add 0, 1 (modulo 2), or -1 (modulo 2). So the idea is to use velocities whose components have just these three possible values$\{-1,0,1\}$, and to combine the three tendencies so that the result is always 0 or 1 just by using the modulo function. The pseudo-code of the algorithm is given below.

For each particle
Initialise each x component at random in $\{0,1\}$. Set $g =$
x and $p = x$.
Initialise each v component at random in $\{-1,0,1\}$.
Say the particle informs itself.
Initialise information links at random towards $K -$
1 particles (not necessarily different).
Loop as long as the stop criterion is not satisfied:
For each particle
  Choose at random a coefficient c_2 in $\{-1,1\}$
  Choose at random a coefficient c_3 in $\{-1,1\}$
  Find the best informant (the one that has the best previous position g)
  Apply the following transformations:

$$\begin{cases} v & \leftarrow & v + c_2 (p - x) + c_3 (g - x) \\ x & \leftarrow & x + v \\ x & \leftarrow & (4 + x) \, mod \, 2 \\ v & \leftarrow & (3 + v) \, mod \, 3 - 1 \end{cases}$$

  Compute the fitness$f(x)$
  If $f(x) < f(p)$then $p \leftarrow x$
If there has been no improvement of the best position in the whole swarm, re-initialise

As usually, the stop criterion is either "the solution has been found" or "a given maximum number of evaluations has been done."

This algorithm looks quite similar to the classical PSO. However, there is a trick, it is almost a hoax for it works extremely well on the

p_d $g_d \Rightarrow$	0	1
0	0	$rand\{0,1\}$
1	$rand\{0,1\}$	1

Table 2: Choice of the new $x_d$value "at the majority" for all possible values of the two bits $p_d g_d$

three above "deceptive deceptive" problems, and not at all on the others. Try to guess why ...

3.2 Derivation 7

This one is more serious. Its codification is A1-A4-A4-B6-C1-D1. The structure of the algorithm is the same as in Derivation 0 but here the velocity is not used, and the two other tendencies are computed by modifying at random some bits.

Then, for each dimension d, we have to choose between three possible values. The final one is chosen "at the majority", according to the table 2

To improve the convergence, the numbers of modified bits, i.e. respectively n_p , and n_g, are regularly decreased, according to the number of evaluations T, if the previous iteration gave an improvement (the best value found by the swarm is better than the previous one). The following formulas are not arbitrary, but not really proved either, so, for the moment, just say they are rules of thumb:

After initialisation

$$\begin{cases} c_0 & = & 0.5 \ln(D) \frac{ln(K)}{ln(S)} \\ z & = & \frac{K}{S} \end{cases}$$

At each time step, if no improvement, $z = z + z_0$

$$\begin{cases} n_p & = & \frac{c_0}{ln(z)} D \\ n_g & = & \begin{cases} \frac{n_p}{ln(K-1)} \text{ if } K > 2 \\ n_p \text{ else} \end{cases} \end{cases}$$

3.3 Derivation 11

This is a simplification and a binary adaptation of the pivot method [SER 97]. As we have seen, its codification is A1-A1-A3-B1-C1-D1. The idea is to just look "around" the best previous position g of the best informant, that is to say to modify at random a few number of g components. The basic algorithm is the following

$$
\begin{array}{rcl}
x & \leftarrow & g \\
\Delta & \leftarrow & INT\left(\ln\left(D\right)\right) \\
\rho & \leftarrow & 1 + INT\left(rand\left(1, \Delta\right)\right)
\end{array}
$$

`choose at random` ρ `components of` x, `and switch them.`

Difficult to define something more simple! In practice, for small D values (≤ 20), it is sometimes a good idea to keep Δequal to 3. It is important to note that due to the integer part function INT, the ρ value is never equal to $\Delta + 1$. So, for $\Delta = 3$, we choose 2 or three components to switch. There is here a subtility: it does *not* mean we never switch just one bit, for there is a small probability that the ρ components chosen at random are in fact the same. Actually, the more intuitive variant with $\rho \leftarrow rand\left(1, 2, \ldots, \Delta\right)$works sometimes far better (for example on Multimodal problem), but also sometimes far worse: it seems it is less robust.

3.4 Derivation 100

This derivation is an improvement of the original algorithm defined by Jim Kennedy and Russ Eberhart [KEN 97]. Its codification is A7-A5-A5-B2-C2-D4-D1. The only difference compared to the classical (historical) continuous PSO is that each new component is set to 0 or 1 by applying a sigmoid transformation and a probabilistic rule.

Also, note that here the informant group is usually the classical circular one, and it is computed just once, at the beginning. For example, for $K = 3$ and if the particles are numbered from 0 to $S - 1$, the informants of the particle i are particles $(i-1) \, mod^{+} S$, i , $(i+1) \, mod^{+} S$. Of course, you can also use it with the random information links we have defined above. Actually, it is then often better.

The movement equations are

$$
\left\{
\begin{array}{rcl}
c_1 & = & 1 \\
c_2 & = & 2 \\
c_3 & = & 2 \\
v_d & \leftarrow & c_1 v_d + rand\left(0, c_2\right)\left(p_d - x_d\right) + rand\left(0, c_3\right)\left(g_d - x_d\right) \\
u & \leftarrow & x_d + v_d \\
u & \leftarrow & \frac{1}{1+e^{-u}} \\
r & = & rand\left(0, 1\right) \\
r < u & \Rightarrow & x_d \leftarrow 1 \\
r \geq u & \Rightarrow & x_d \leftarrow 0
\end{array}
\right.
$$

Note that for this algorithm it is sometimes better to use the "global best" method to define the informants, i.e. $K = S$.

Algorithm	30D	60D	90D	150D
GA	0.58 (100000)	4.05 (100000)	5.55 (100000)	13.26 (100000)
M-DiPSO	0 (5417)	0 (26368)	0 (39885)	0 (75150)
Derivation 0 S2 K2	0 (30)	0 (31)	0 (32)	0 (32)
Derivation 0 S35 K3	0 (372)	0 (400)	0 (411)	0 (426)

Table 3: Goldberg's order-3 problem for different dimensions. Fitness and number of evaluation, averaged over 10 trials for GA and M-DiPSO, over 1000 for Derivation 0

4 Results

4.1 Pure strategies

We now apply each derivation on each problem, just "as it is". We will see later how to add more adaptation or to combine some of them. For easier comparison, we use here most of the time the same swarm size (35), although for some problems best results can be obtained with a different value. As usually, a small K parameter is often better for difficult problems, so we use either 2 or 3. For Derivation 11 and Derivation 100, we also try $K = S$. As pointed out in [CLE 05], using just such "extreme" values is enough to give us a pretty good idea of the algorithm efficiency. In the following results, each strategy is coded by something like "11 S35 K3", meaning:

use	of Derivation 11
swarm	size 35
each	particle informs itself, and 2 particles chosen at random after each time step if there has been no improvement

4.2 Derivation 0 on "deceptive deceptive problems"

First, let us try our "hoax" on the three deceptive deceptive problems we have seen in section 2. Results with GA and M-DiPSO are coming from [ALK 02]. As you can see on the tables 3 to 5, results with Derivation 0 are indeed too good to be honest. Note that for GA and M-DiPSO, only 10 runs had been launched by the authors, as Derivation 0 has been ran 1000 times in order to reduce the standard deviation (not noted here). In the tables the first number is the mean best fitness value found and the second one (in parenthesis) the mean number of evaluations. The stop criterion is either a solution is found or the number of evaluations is 100000.

Algorithm	30D	60D	90D	150D
GA	0.32 (100000)	1.52 (100000)	3.08 (100000)	none
M-DiPSO	0 (15690)	0 (45902)	0 (83348)	none
Derivation 0 S2 K2	0 (21)	0 (22)	0 (22)	0 (21)
Derivation 0 S35 K2	0 (254)	0 (265)	0 (266)	0 (266)

Table 4: Bipolar order-6 problem. Fitness and number of evaluation, averaged over 10 trials for GA and M-DiPSO, over 1000 for Derivation 0

Algorithm	30D	60D	90D	150D
GA	0.95 (100000)	11.7 (100000)	24.1 (100000)	55.75 (100000)
M-DiPSO	0 (5354)	0 (15344)	0 (42358)	0 (88488)
Derivation 0 S2 K2	0 (24)	0 (25)	0 (25)	0 (25)
Derivation 0 S35 K2	0 (268)	0 (285)	0 (294)	0 (299)

Table 5: Mühlenbein's order-5 problem. Fitness and number of evaluation, averaged over 10 trials for GA and M-DiPSO, over 1000 for Derivation 0

Of course, there is a trick. The point is Derivation 0 works well only on this kind of problem. By "this kind of problem", I mean a problem for which either 111...111 **or** 000...000 **is a solution. And the trick is it is quite difficult to guess it just by reading the algorithm. For other problems, as we will see now, it doesn't work at all.**

4.3 All derivations on all problems

4.3.1 Performance curves

For each problem and for each derivation, we plot the success rate over 100 **runs, for a range of maximum numbers of evaluations** T**. The swarm size is always equal to** 35**, and several** K **values have been tested. Only the best one is retained, except for Derivation 100 for which there are two plots for the two best ones, just to see the difference. The purpose of these figures is not to decide whether an algorithm is better than another one or not, for it is here impossible (the benchmark problems have precisely been chosen for that!), but to point out some rules and, if I dare say, some "non rules":**

- on a given problem, if algorithm A is better than algorithm B for a given T, it is better for any greater T value. False: see for example 5 and derivations 7 and 11

- if an algorithm is excellent on some problems, it is very bad on some others. It seems to be true, according to the figures, but it is not, as we will see later
- a problem with a lot of local optima is more difficult. Meaningless: it depends on the algorithm. See for example the figure 6, where Derivation 100 is excellent on the Multimodal problem

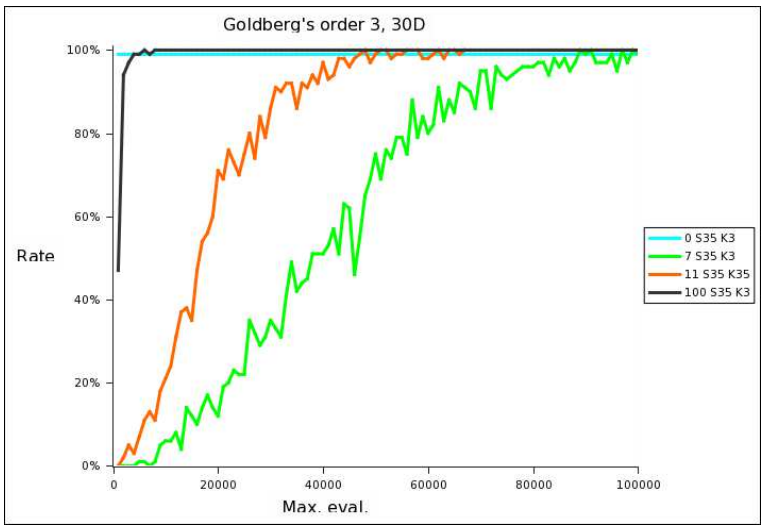

Figure 1: Goldberg's order 3. Success rate vs maximum number of evaluations

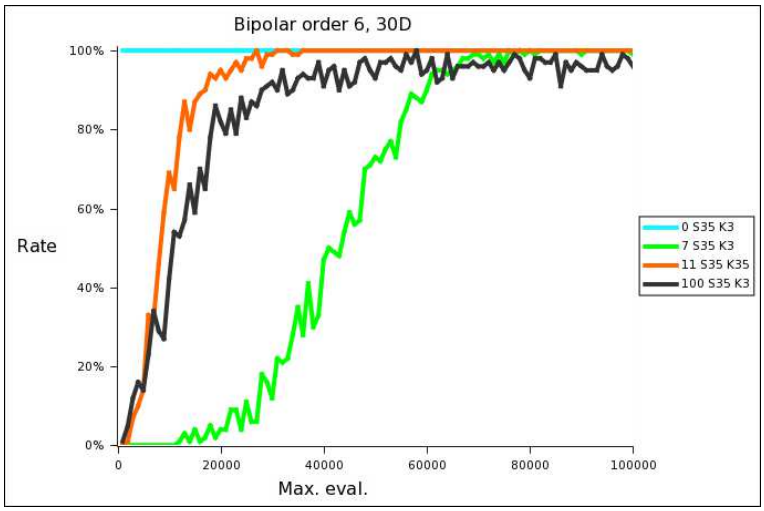

Figure 2: Bipolar order 6. Success rate vs maximum number of evaluations

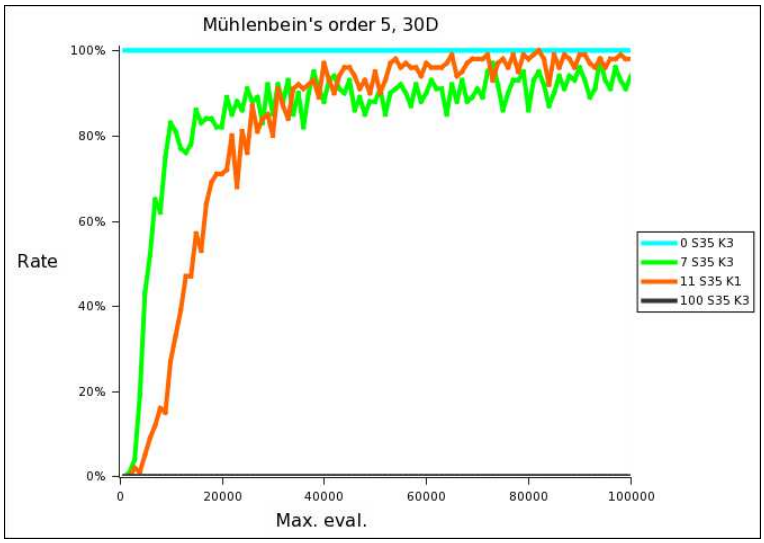

Figure 3: Mühlenbein's order 5. Success rate vs maximum number of evaluations

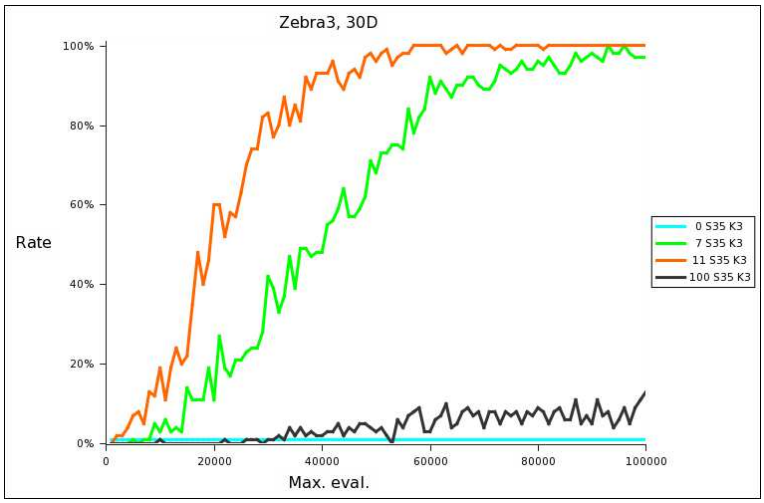

Figure 4: Zebra3. Success rate vs maximum number of evaluations

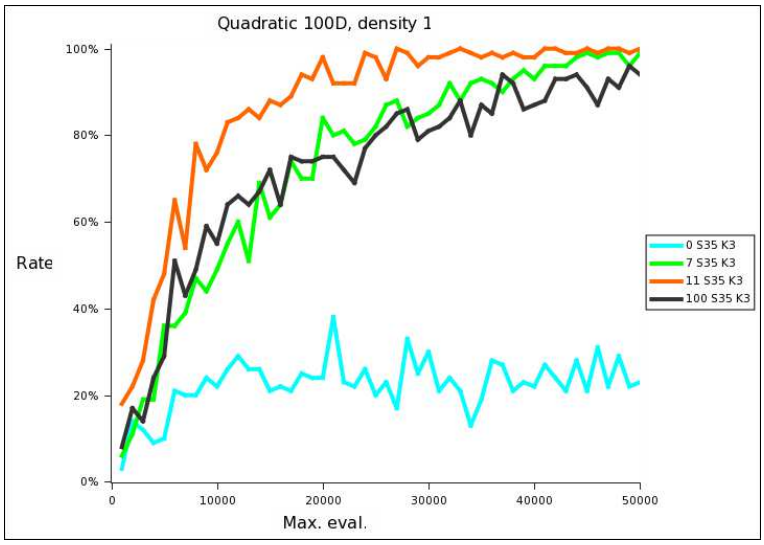

Figure 5: Quadratic. Success rate vs maximum number of evaluations

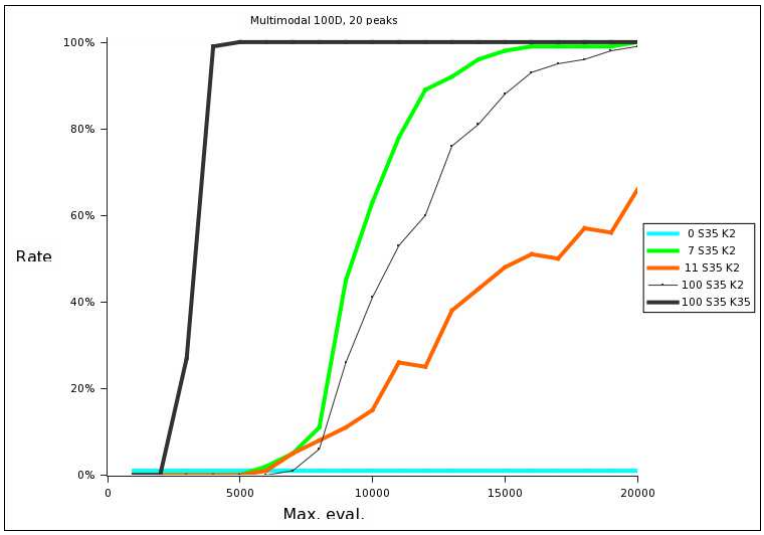

Figure 6: Multimodal. Success rate vs maximum number of evaluations

For the Multimodal problem (100 **dimensions**, 20 **peaks**), **the choice** $K = 3$ **gives very bad results. For example, for Derivation 7, the success rate for at most** 20000 **evaluations is completely null. With** $K = 2$**, we obtain more reasonable success rates. However, for Derivation**

	0 S35	7 S35	11 S35	100 S35
Goldberg	100%	60%	78%	99%
Bipolar	100%	58%	89%	84%
Mühlenbein	100%	85%	18%	0%
Zebra3	0%	60%	78%	4%
Quadratic	22%	74%	87%	72%
Multimodal	0%	54%	25%	86%
Mean	**54%**	**65%**	**63%**	**58%**
Standard dev.	**47%**	**11%**	**29%**	**40%**

Table 6: Mean success rates and standard deviations

100, the best information links topology is what is usually called the "global best" one, i.e. each particle is informed by the whole swarm ($K = S = 35$).

4.3.2 Robustness versus efficiency

What do you prefer: never bad results *or* sometimes excellent ones? Unfortunately, it appears that the *and* is not really possible, at least with just the simple mono-strategies we have defined. Table 6 indicates the averaged success rates and the standard deviations over the six problems. We clearly see that Derivation 0 and Derivation 100 can be used in the second case, as Derivation 7 is the only one that is never really bad, and quite robust (small standard deviation). However, it is also never excellent.

Now, it wouldn't be very interesting to just notice that whitout trying to explain what happens. The two keywords are *adaptation* and *DPNP* (Distribution of Possibilities for the Next Positions), as explained in [CLE 05]. For all derivations but Derivation 100, there is a rudimentary adaptation of the information links, for they are modified at random if there has been no improvement. However, Derivation 0 has the poorest DPNP, for the confidence coefficients have just two possible values, 1 and -1. It is then not surprising that this algorithm is very good on some "suitable" problems, but most of the time very bad. The DPNP for Derivation 100 is more complete, so it is a bit better, but still not very robust, for the topology of the information links is constant: on some problems, this "circular" topology is not a good one.

Derivation 7 and 11 both have a quite good DPNP, but 7 is more adaptive, for the number of modified bits is decreased if there has been some improvement. That is why it is the most robust (the

smallest standard deviation). However, it is possible to design something better, by adding more adaptation or by combining two "pure" strategies

5 Adaptive strategies

As some performances are sometimes very different for different K values, a more robust algorithm may be obtained just by modifying K during the process. Similarly, it is perfectly possible to use several derivations chosen more or less cleverly after a given number of non efficient time steps (no improvement). Also, some more clever adaptive rules can be used, like a kind of "taboo"to limit the DPNP. As it is not the purpose of this paper to describe in detail an adaptive binary PSO, I just give below a few examples of what can be done even with very simple rules.

5.1 Derivation 16 = Derivation 11 with taboo

One of the three tendancies in the classical PSO equations can be seen as "keeping more or less the same velocity". However, just applied "as it is" for a binary problem, it is meaningless. If the position is 0 and the velocity is 1, applying two times the same velocity to the position gives again 0, as the underlying idea of this tendency is on the contrary "keep moving in *this* direction".

A possible way to convey this idea for a binary problem is to say something like "if a component has just been modified, don't modify it again for the next move". So the particle needs at least three moves to possibly go back to the same position.

Let's apply this "taboo" rule to the simplest derivation we have seen, i.e. Derivation 11. Note that this pivot method is already quite good for almost all our test functions, except for the Multimodal problem. As we can see on figure 7, the resulting algorithm is then never bad ... assuming that we choose the right K value for each problem. This illustrates that adding just one adaptation rule is sometimes not enough. Here, we would need another rule saying how to modify K during the process.

5.2 Combining strategies

As it clearly appears that some derivations are pretty good on some problems, and pretty bad on some others, another idea is to automatically decide during the search process which one has to be used. So, from time to time, the algorithm has to check if it would be a good

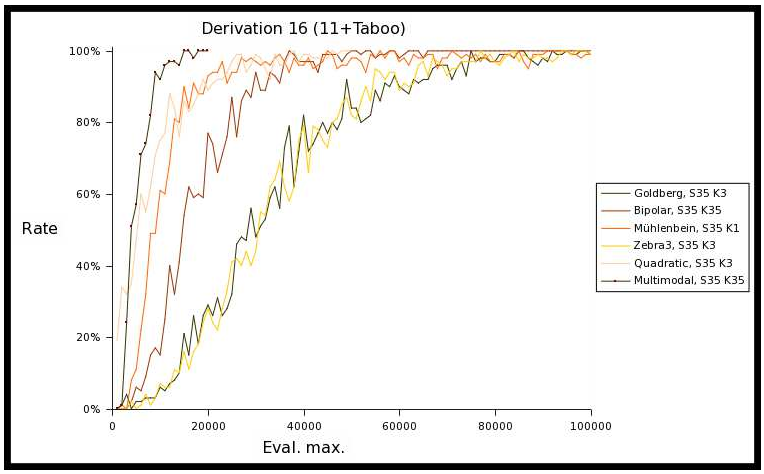

Figure 7: Adding a "taboo" rule to Derivation 11. Only results with the best K values are shown

	16 S35 (11+Taboo)	16/0_parall S35 K3	16/0_seq S35 K3
Goldberg	67% K3	99%	99%
Bipolar	83% K35	98%	99%
Mühlenbein	89% K1	99%	99%
Zebra3	67% K3	50%	52%
Quadratic	85% K3	77%	81%
Multimodal	77% K35	68%	79%
Mean	**78%**	**82%**	**85%**
Standard dev.	**8%**	**18%**	**17%**

Table 7: Mean success rates and standard deviations when using some adaptations. On the whole, the algorithme is better. With the taboo option, it is also more robust (small standard deviation), but the best K value is not always the same. So the sequential approach, in which another strategy (derivation) is chosen if there has been no improvement for a while, is the most interesting

idea to switch to another strategy. A theoretical reasoning detailed in [CLE 05] shows that a good estimation of the number of time steps between two such adaptations is $A = \frac{S}{K-1}$ (the idea is to give enough time so that information can spread all over the swarm). Note that in complete adaptive systems, neither S nor K are constant, but we don't consider here this case. To give an idea of the process, let's suppose we have to choose just between two strategies. There are mainly two ways to do that: by doing parallel trials or sequential trials.

5.2.1 Parallel trials

This method is quite expensive in terms of number of evaluations but nevertheless better than all the ones we have seen. At a given time t all values defining the current status of the swarm are saved and, from this starting point, the algorithm tries the strategy already in use and after that another one for A iterations. If the final result is better, this second strategy is kept until the next check time.

As Derivation 16 was already the best, we can now try to combine it with say Derivation 0. Of course, it is just to illustrate the process for, in practice, Derivation 0 is far too specific to really be useful. Times t at which the trials are done are A, $2A$, $3A$, etc. As we can see on table 7, the results are on the whole better, but the standard deviation is not as small as with the previous method (with taboo): 18% versus 8%. However, the good point is that there is no need to look for the best K value: the "standard" one 3 can be kept for all problems of our benchmark function set.

5.2.2 Sequential trials

Here the iterations are not repeated with another method. Simply, if there has been no improvement during A iterations, another strategy is chosen. Let's try again with Derivation 16 and 0. On table 7we see that it is slightly better than the previous one, and have the same robustness property for the K value. It is also easier to code. So we may say it is our best choice.

5.3 Saving computation time

As you may have noted, for some of the objective functions the fitness is computed by "accumulating" some positive elements (Goldberg, Bipolar, Mühlenbein, Zebra3). In such a case it is not always necessary to completely compute the fitness: we can stop as soon as there is no hope to find a better value than the one that is memorised. Note that this trick is not specific to binary optimisation. For

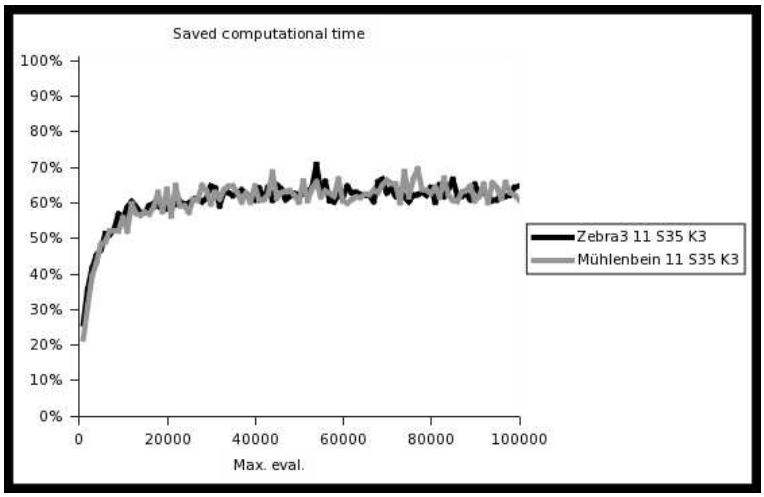

Figure 8: Saved computational time (mean over 10 runs for each maximum number of evaluations). By not completely computing some fitness values, as soon as it is absolutely sure the position has not been improved, it is possible to significantly reduce the computational time if each fitness evaluation is quite time comsuming

example it can be used for all optimisation problems where the fitness is a distance, usually an Euclidean one.

This can be seen as a kind of small adaptation. Let us see in detail the process for the Zebra3 function. Let f_{best} be the memorised fitness value. It means the maximum value ever found by the current particle is $f_{max} = D/3 - f_{best}$. Now the fitness is computed by progressively adding $D/3$ values, each one being at most equal to 1. Let $f(d)$ the partial sum of the d first 3-bit sub-strings. The maximum value that we can hopefully reach is then $f(d) + D/3 - d$. If this value is equal to or smaller than f_{max} there is no need to continue: in any case we won't use the real fitness value, for we need it only if the position is improved. Of course for initialisation the computation *must* be complete, but after that it can save time.

How much? For small K values, a simplified model can estimate it as a function of the number of "best neighbours" and of the swarm size (see 7.6.4). As we can see on figure 8 the saving may be quite significant. Note that in this figure the saving is not given in terms of time (which is, anway, depending on the computer in use), but in terms of elementary operations. For Zebra3, if a given fitness evaluation is stopped after having added the partial values of d 3 bits

substrings, the saving is estimated by $(D - 3d)/D$.

6 Some possible future general improvements

The improvement techniques presented below are inspired by some less specific (continuous) PSO versions, but may also be as source of inspiration for some other versions. That is why I call them "general".

6.1 Initialisation

When you have decided to use S particles, you also have to define the starting configuration. Usually the initial positions are completely at random. However, it has been experimentally shown that starting from a more evenly distribution (by using centroidal Voronoi tesselations) significantly improves the performance for some continuous test functions as soon as the dimension is high [RIC 03]. So it would be interesting to do the same for binary problems. The first question is then "How to do that?" in this specific case. More precisely, is it possible to define a specific algorithm better than the one used in [RIC 03], that needs a lot of computation. And the second one is of course "Is it useful"? Preliminary results seem to show it is not so obvious.

6.1.1 Definition of a regular distribution

First, let us give a precise definition of what means here "a more evenly distribution", or *regular distribution*. We have S binary positions $\{s_1, ..., s_i, ...s_S\}$ to define in the binary search space B . For a given s_i let $I_i$its "influence domain", defined by

$$I_i = \{x, x \in B, \forall j, j \# i, d(x, s_i) \leq d(x, s_j)\}$$

where d is the Hamming distance. Actually, we define here a binary Voronoi tessellation. Let V_i be the "volume" of I_i, i.e. its number of elements. The distribution is perfectly regular if all V_i have the same value. However, it is not always possible, so a regular distribution will simply be one that minimise the variance of the V_i, i.e. the quantity v defined by

$$\begin{cases} v & = & \frac{1}{S}\sum_{i=1}^{S}(V_i - m)^2 \\ \text{with} & & \\ m & = & \frac{1}{S}\sum_{i=1}^{S}V_i \end{cases}$$

6.1.2 Building a regular distribution

It is quite easy to write a deterministic algorithm that progressively builds a regular distribution. The idea is to add the particles one at a time, and for each bit to try both values 0 and 1, trying to keep the distances between particles as big as possible. Pseudo-code

```
s_1 = 000...000
For each particle i, from 2 to S
{
  For each dimension d, from 1 to D
  {
  s_i(d) = 0 // Try 0
  Compute the distances from s_i to the  s_j with j < i
  Compute the mean m_0, and the variance v_0 of these distances
  Do the same with s_i(d) =
1, and compute the mean m_1, and the variance v_1
  If m_1 > m_0 then s_i(d) = 1; continue with next d
  If m_1 < m_0 then s_i(d) = 0; continue with next d
  // Case m_1 = m_0
  If v_1 < v_0 then s_i(d) = 1 else s_i(d) = 0;
  }
}
```

Examples

For $D = 4$, $S = 3$ the algorithm gives

$$\begin{cases} s_1 &= 0000 \\ s_2 &= 1111 \\ s_3 &= 0101 \end{cases}$$

The "volumes" of the influence domains are respectively 9, 10, and 9. There is no way to do better, simply for 2^4 is not divisible by 3. For any position x, there exists at least one particle s_i so that $d(x, s_i) \leq 2$. For $D = 3$, and $S = 4$, we have $2^D/S = 2$, and the algorithms indeed gives a perfectly regular distribution $\{000, 111, 010, 101\}$, with all "volumes" equal to 3:

$$\begin{cases} I_1 &= \{000, 001, 100\} \\ I_2 &= \{111, 011, 110\} \\ I_3 &= \{010, 011, 110\} \\ I_4 &= \{101, 001, 100\} \end{cases}$$

	Random inititialisation	Regular initialisation + random "rotation"
Goldberg	80,36 %	80,48 %
Bipolar	90,37 %	90,56 %
Mühlenbein	11,72 %	12,52 %
Zebra3	80,67 %	80,56 %
Quadratic	85,34 %	85,44 %
Multimodal	53,55 %	54,75 %
Mean	67,00 %	67,39 %

Table 8: Influence of a regular initialisation on the success rate. Here the swarm size is "optimum", and the improvement is very small

In practice, once you have such a regular distribution, it is a good idea to build some others, obtained by randomly "rotating" it. You just have to add the same random binary vector to all the s_1. Note that for adding you have to use the binary algebra, for which $1 + 1 = 0$.

6.1.3 Result comparison

We now try to compare the result with the classical pure random initialisation on the one hand, and with randomly rotated regular distributions on the other hand. It is easier to see the differences when using small swarms. We use here the "optimal swarm size", as computed below in 7.1.2, i.e. $S = 12$ for $D = 30$, and $S = 20$ for $D = 100$. Table 8 is build as the one in 4.3.2. However here only Derivation 11 with $K = 3$ has been used. As we can see, with these swarm sizes there is almost no difference with regular initialisation, just most of the time a very slight improvement. Also (it is not reported in the table), the standard deviation of the mean best value is slightly smaller. So, at least for small dimensions (≤ 100) it is not really worthwhile to use a regular distribution. However it would be interesting to check it in high dimension, for the rate swarm_size/search_space_volume becomes quite small.

6.2 When the unit is the tribe

We have always used expressions like "update the memory of the particle". However it may be interesting to work at a higher level, by considering a whole set of particles as an "unit" and by modifying the memory of this "unit".

Let's call here *tribe* at time step t the set of the informants of a particle x_t, including itself, i.e. what we have above called its informant group. Note that it is not exactly the same concept that has been used for a complete adaptive PSO version (precisely called TRIBES), in which the swarm size and the information links are judiciously modified during the process [ONW 04, CLE 05] . Here, the meaning is weaker, and the swarm size is still constant. The very basic principle of a simple PSO can now be symbolicaly written

$$x_{t+1} \leftarrow tribal\_memory \oplus creativity$$

where "tribal_memory" means "the set of the best known positions by the tribe/informant group", and "creativity" means "some randomness". Note that this equation is quite important for it summarises all PSO versions. For example in Derivation 11, a new position of a given particle is computed by

1. looking for g, the best of the best positions known by the informants of the particle (make use of the tribal memory)
2. choosing the new position x' at random "around" g (creativity)
3. if x' is better than the best position p known by x, then replace p it by x'

However, if we work at the tribe level, there are some other ways for the step 3. In particular, this two ones are quite intuitive

3a. for all best positions p known by the tribe, if x' is better than p, then replace p by x'. It means several memories may be modified.

3b. if p is the worst of the best positions known by the tribe, and if x' is better than p, then replace p by x'. It means the modified memory is not necessarily the one of the current particle.

References

[ALK 02] Al-Kazemi, B., Mohan, C. K., "Multi-phase discrete particle swarm optimization. Proceeding of Fourth International Workshop on Frontiers in Evolutionary Algorithms (FEA)," 2002

[BAR 03] Baritompa, W. P., Hendrix, E. M. T., "On the investigation of Stochastic Global Optimization algorithms," Journal of Global Optimization, 2003.

[CLE 04] Clerc, M., "Discrete Particle Swarm Optimization, illustrated by the Traveling Salesman Problem," in New Optimization Techniques in Engineering. Heidelberg, Germany: Springer, 2004, p. 219-239.

[CLE 05] Clerc, M., L'optimisation par essaims particulaire paramétrique et adaptative, Hermès Science, 2005.

[GLO 02] Glover, F., Alidaee, B., Rego, C., Kochenberger, G. A., "One-Pass Heuristics for Large-Scale Unconstrained Binary Quadratic Problems," European Journal of Operational Research, 2002.

[KEN 97] Kennedy, J., Eberhart, R. C., "A discrete binary version of the particle swarm algorithm," presented at Conference on Systems, Man, and Cybernetics, 1997, p. 4104-4109.

[KEN 98] Kennedy, J., Spears, W. M., "Matching Algorithms to Problems: An Experimental Test of the Particle Swarm and Some Genetic Algorithms on the Multimodal Problem Generator," presented at International Conference on Evolutionary Computation, 1998, p. 78-83.

[ONW 04] G. C. Onwubolu, "TRIBES application to the flow shop scheduling problem," in New Optimization Techniques in Engineering. Heidelberg, Germany: Springer, 2004, p. 517-536.

[RIC 03] Richards, M., "Improving Particle Swarm Optimization", Master of Science thesis, Brigham Young University, 2003

[SER 97] Serra, P., Stanton, A. F., Kais, S., "Pivot method for global optimization," in Physical Review, vol. 55, 1997, p. 1162-1165.

7 Appendix for "amatheurs"

A remarkable property of binary PSO is that a lot of behaviours can be modelled just by carefully counting some well chosen configurations, in order to compute some probabilities. It is then possible to give some practical formulas for the main parameters, namely the swarm size, the number of informants of a particle and the number of bits to switch at each time step.

7.1 Swarm size

7.1.1 A spherical search space

In a classical (discrete) Euclidean space, the number of points at distance δ from g is increasing like δ^D. In a binary space with the Hamming distance (number of bits to switch) it is equal to C_D^δ(the number of possibilities to choose δ elements amongst D, which gives a well known bell shape curve, centered on $D/2$. In particular there is just one point at distance D. Like on a sphere of radius $D/2$, the number of points at distance δ begins to increase and then decreases.

7.1.2 Swarm size estimation

Let's suppose the initialisation is "perfect", that is to say each particle has about the same number of positions in the search space that are closer to it than to the other particles. The search space can then be seen as a union of "cells", and each cell contains about $2/S$ positions. Let g be a particle, and we are looking around g by switching at most Δ bits, as in Derivation 11. The number of positions we can reach this way is $\sum_{\delta=0}^{\Delta} C_D^{\delta}$. So, in order to be able to reach any position in the cell, we have to find the smallest "radius" Δ so that

$$\sum_{\delta=0}^{\Delta} C_D^{\delta} \geq \frac{2^D}{S} \tag{1}$$

Although there is no simple formula to derive Δ from 1, it is always possible to numerically compute it when D and S are given, and to tabulate some curves Δ vs S. The good news is that as soon as S is greater than a given value (which is a priori depending on D), this radius decreases very slowly when S increases.

So, as we can see on figure 9, the radius is almost the same (about 10 for $D = 30$, and about 40 for $D = 100$) for any swarm size between 20 and 50. It means that for for a given "search effort" of T evaluations, using $S = 20$ or$S = 50$ will give more or less the same success rate, and more or less the same mean number of evaluations for the successful runs. Actually, because of the way PSO works, it is better to have as many iterations as possible, for the information has then more time to be spread between particles [CLE 05] . As this number of iterations is equal to T/S, it means the smallest "acceptable" S value is the best one.

It is then possible to build a simplified formula giving a swarm size that is good for a large range of dimensions. In order to do that, for each D the curve Δ vs S is interpolated by a three times differentiable one, and we are looking for the S_0 value that minimise the curvature radius. After that, a good estimation of the wanted S_{min} value can be performed in three steps:

1. find the point $O = (S_0, \Delta(S_0))$ where the osculator circle has the smallest radius R
2. find the center C of this circle. Let S' be its first coordinate
3. define S_{min} by $S_{min} = INT(S' + 0.5)$, i.e. the nearest integer value.

According to the shape of the curve, we could say it is indeed a point where the "flat" part begins.

First, let us find a differentiable function that approximates $\sum_{\delta=0}^{\Delta} C_D^{\delta}$. It is quite easy by using a logistic curve, for example

$$H(\Delta) = \frac{2^D}{1 + e^{-\gamma(\Delta - \beta D)}}$$

By solving the equation $H(\Delta) = 2^D/S$ we find a formula for Δ

$$\Delta(S) = \beta D - \frac{\ln(S-1)}{\gamma} \tag{2}$$

The γ coefficient can be tabulated, but it is easier to use a good approximation given by

$$\gamma = \frac{\gamma_1}{1 + D^{\gamma_2}}$$

with $\gamma_1 = 4.5$ and $\gamma_2 = 0.54$.

We have then $\Delta'(S) = -\frac{1}{\gamma(S-1)}$, $\Delta''(S) = \frac{1}{\gamma(S-1)^2}$, and $\Delta'''(S) = -\frac{2}{\gamma(S-1)^3}$.

Now, for $\Delta''$is always positive, the radius curvature is given by the classical formula

$$R = \frac{\left(1 + \Delta'^2\right)^{3/2}}{\Delta''}$$

We are looking for the S value for which it is minimum. The equation $R' = 0$ gives us

$$3\Delta' - \left(1 + \Delta'^2\right)\Delta''\Delta''' = 0$$

and we finally have to solve

$$-3\gamma(S-1)^6 + 2(S-1)^2 + 2 = 0$$

Let us define $X = (S-1)^2$ and $p = -2/3\gamma$. Then the equation becomes $X^3 + pX + p = 0$. By applying the Cardan formula, we find that the real solution is

$$S_0 = 1 + \sqrt{\sqrt[3]{-\frac{p}{2} + \sqrt{\frac{p^2}{4} + \frac{p^3}{27}}} + \sqrt[3]{-\frac{p}{2} - \sqrt{\frac{p^2}{4} + \frac{p^3}{27}}}}$$

and the value we are looking for is then

$$S_{min} = INT\left(S_0 - \Delta' \frac{1 + \Delta'^2}{\Delta''} + 0.5\right)$$

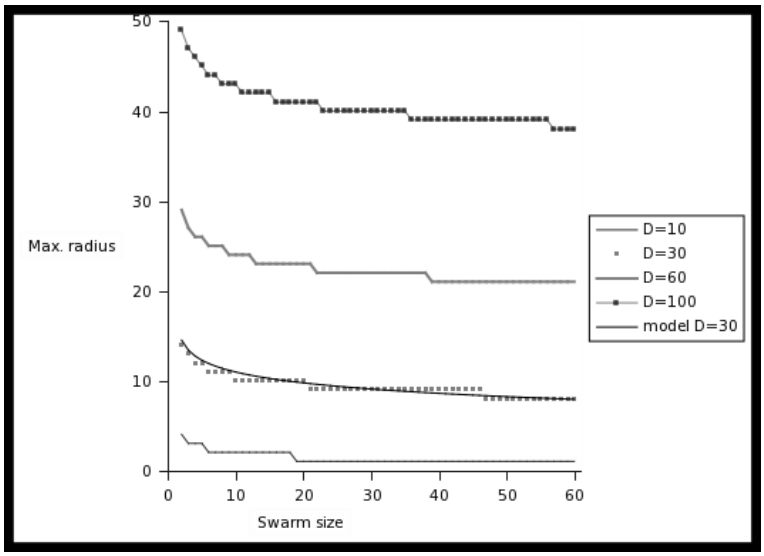

Figure 9: Maximum radius. If the distribution of the particle is enough evenly, any point of the search space is at a distance of the nearest particle at most equal to this radius. As expected, this radius decreases when the swarm size increases, but very slowly as soon as the swarm is big enough. These curves can be approximated by a differentiable model

As Δ' is negative and Δ'' positive, it is as expected greater than S_0. For $D = 30$, we find $S_0 = 2.41$, and $S_{min} = 12$. For $D = 100$, $S_0 = 2.56$, and $S_{min} \simeq 21$. The whole process is quite complicated but now we have a theoretical formula, we can find some far more simple empirical ones that give similar values, at least for not too high dimension (typically smaller than 500), for example

$$S_{min} = INT\left(9.5 + 0.124\left(D - 9\right)\right) \tag{3}$$

Using such "optimal" swarm sizes usually slightly increases the effectiveness, particularly when T is too small to obtain a high success rate. For example, for the 30D Zebra3 function and $T = 15000$ the success rate is 29% with $S = 12$, as it is 22% with $S = 35$ (with Derivation 11).

7.2 Performance curve model

Let's try to model the performance curves "Success rate vs Maximum number of evaluations". Let $s\left(T\right)$ be such a curve. It may be easier to understand what happens if we suppose each success rate is estimated by running N times the algorithm (on a given problem). So $Ns\left(T\right)$ is

the number of successful runs, that is to say the number of runs that have found a solution after at most T evaluations.

Now let's consider N runs with a greater maximum evaluation number $T + \Delta T$. It is quite obvious there are at least $Ns(T)$ successful ones (we neglect here the probabilistic fluctuations), found during the first T evaluations. What about the $N - Ns(T)$ other ones? They still have ΔT evaluations to possibly find a solution. It seems reasonable to apply to them a success rate proportional to the previous one, i.e. $\mu s(T) \Delta T$. So, finally, the total number of successful runs is $Ns(T) + \mu s(T)(N - Ns(T))\Delta T$. By dividing by N, we obtain the new success rate:

$$s(T + \Delta T) = s(T) + \mu s(T)(1 - s(T))\Delta T \tag{4}$$

This is a classical iterative equation for a logistic curve. The s function is given by:

$$\left\{ \begin{array}{rcl} \alpha & = & \left(\frac{1}{s(T_0)} - 1\right) e^{\mu T_0} \\ s(T) & = & \frac{1}{1+\alpha e^{-\mu T}} \end{array} \right. \tag{5}$$

Theoretically, if the model was perfect, we could take $T_0 = 1$ and then, if there is just one solution, $s(T_0) = 2^{-D}$ (i.e. the probability to find the solution just by chance at the very beginning). However, of course, the model is *not* perfect, and it doesn't match then very well (far too pessimistic), so it is better to "forget" the first equation, and to consider there are two independent parameters, α and μ(another way, which is equivalent and might be more intuitive, is to start from a "reasonable" T_0 value, say 1000, to consider that $s(T_0)$ is a parameter, and to compute α). Then, by running the algorithm N times with two different maximum number of evaluations T_1 and T_2 we obtain two success rate estimations $s(T_1)$and $s(T_2)$. From then, it is easy to compute the two parameters, by solving the system

$$\left\{ \begin{array}{rcl} s(T_1) & = & \frac{1}{1+\alpha e^{-\mu T_1}} \\ s(T_2) & = & \frac{1}{1+\alpha e^{-\mu T_2}} \end{array} \right.$$

We find

$$\left\{ \begin{array}{rcl} \mu & = & \frac{1}{T_2 - T_1} \ln\left(\frac{1/s(T_1)-1}{1/s(T_2)-1}\right) \\ \alpha & = & \left(\frac{1}{s(T_1)} - 1\right) e^{\mu T_1} \end{array} \right.$$

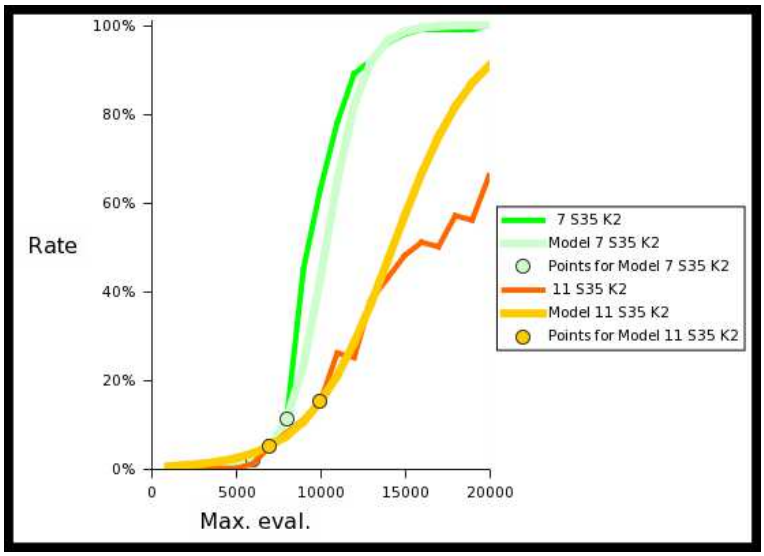

Figure 10: Model for performance curves for the Multimodal problem. Thanks to such a model, it is something possible to estimate how many evaluations will be needed to reach a given success rate, just after having ran the optimiser with two different small maximum numbers of evaluation (see Derivation 7). However it does not work for all strategies (see Derivation 11)

For example, we can try this model for some strategies on the Multimodal 100D problem. As we can see on figure 10, with just two points rapidly computed with quite small numbers of evaluations, we sometimes can have a pretty good idea of the whole performance curve (see Model 7 S35 K2). However with some less efficient strategies the adequacy is not that good (see Model 11 S35 K2), for the model tends here to be too optimistic for high number of evaluations. So it still has to be refined.

However, it is always possible to adjust quite well the sigmoid curve to the real one by directly carefully choosing the two parameters α and μ. After all, it is just a 2D optimisation problem! As we can see from the equations 5, the μ parameter is the most important: it characterizes how efficient is the algorithm on the given problem. For example, for the basic strategies, these characteristic values are respectively 0 for Derivation 0, which is completely unefficient, 0.0009 for Derivation 7, 0.0002 for Derivation 11, and 0.004 for Derivation 100, which is largely the best one.

7.3 Probability distributions

7.3.1 Switching bits

Let g be a position (a D-vector). Let's suppose we are looking at random around g, more or less like in Derivation 11. It means:

1. choose a maximum number of bits to switch, Δ in $[1, D]$
2. choose at random (uniform distribution) a given number k of bits to switch, k in $[1, \Delta]$ .Note that for Derivation 11, k is in $[2, \Delta]$.
3. choose k times at random (uniform distribution) a bit in g, and switch it

The question is: what is the probability $\rho(D, \delta, \Delta)$ to find a point that is at the Hamming distance δ of g? Or, in other words, what is the probability distribution around g? Let's see what happens for each possible k value. Obviously, if $k < \delta$ there is no way to find such a point. For greater k values, we have to compute the number of surjections $s(k, \delta)$ from a set with k elements to a set with δ elements. For $\delta = 1$, this number is of course just 1, and for other values, we have the classical formula

$$s(k, \delta) = \delta \left(s(k-1, \delta) + s(k, \delta - 1) \right)$$

and we can then progressively compute all these numbers of surjections. Now, for a given D and a given δ, there are C_D^δpossible combinations. For a given k we consider all possible δ values, and, at last, each k value has the probability $1/\Delta$ to be chosen. So, finally, we obtain our probability by the formula

$$\rho(D, \delta, \Delta) = \frac{1}{\Delta} \sum_{k=1}^{\Delta} \frac{s(k, \delta) C_D^\delta}{\sum_{i=1}^{\Delta} s(k, i) C_D^i}$$

This quantity tends towards $1/\Delta$ when Δ/D tends towards zero. So the resulting distribution tends to be uniform, although it is not for small D values, as shown on figure 11.

Intuitively, it could seem better to use for k a bell-shape distribution in order to privilegiate small δ values, for example something like

$$proba(k = \delta) = \frac{1/\delta}{\sum_{i=1}^{\Delta} 1/i} \tag{6}$$

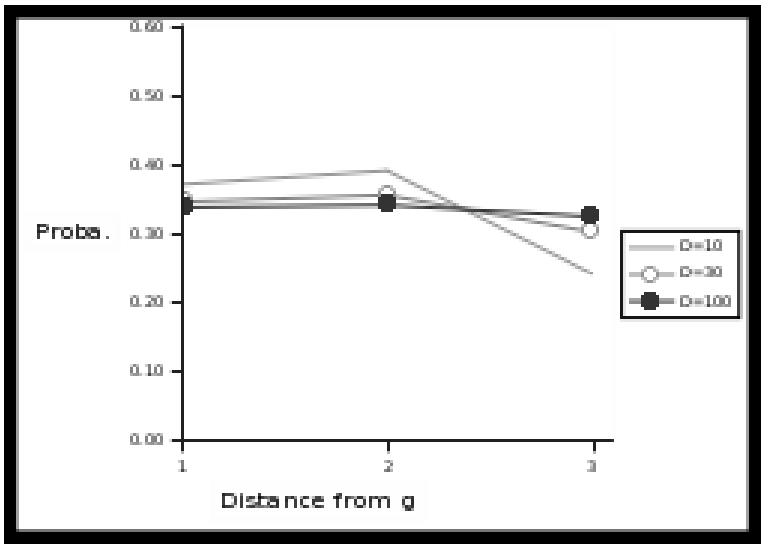

Figure 11: Probability to find a position at distance $\delta$when the "radius" Δ is equal to3, and whith $k \in \{1, \Delta\}$. It tends to be uniform when the dimension increases

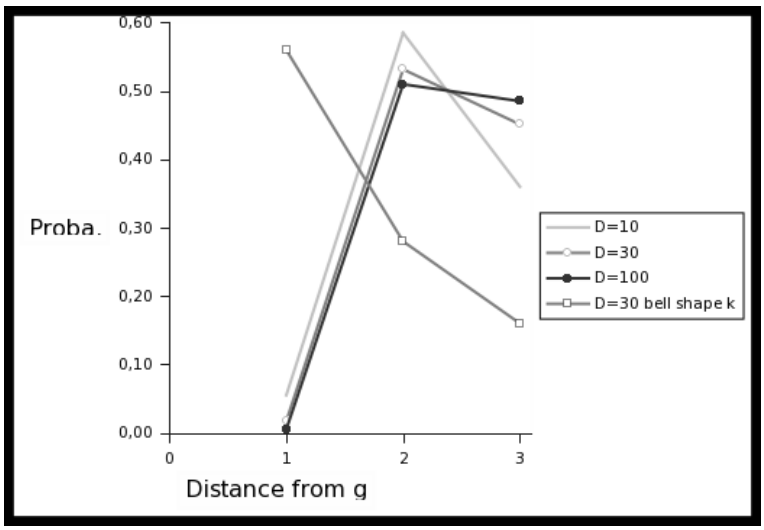

Figure 12: The first three curves show the probability to find a position at distance $\delta$when the "radius" Δ is equal to3, and with$k \in \{2, \Delta\}$ as in Derivation 11. The probability to switch just one bit is not null, but very low. The last curve is obtained when the k distribution is a bell-shape one given by equation ??

Some preliminary tests are not really convincing. Also, Derivation 11 works pretty well with a distribution that gives on the contrary a very low weight to positions at distance 1 (see figure 12). However, the "search power", as defined below, is a bit higher, so it may be worthy to more carefully study this approach.

7.4 Informant group size

7.4.1 Exact formula

We have seen that each particle generates at random $K-1$ information links. It is perfectly possible that two or more of these links point towards the same particle. So, an interesting question is to estimate how many information links a given particle *receives*, or, in other words, what is the mean size of its informant group.

Let's call A this particle. When another particle B generates its links, the probability that none of them reachs A is simply $((S-1)/S)$. And of course on the contrary the probability that B does inform A is the complement to 1 of this value. Now, if the informant group size is exactly L (not taking A into account), it means that L other particles do inform A and that the $S-1-L$ other don't. There are C_{S-1}^{L} such possibilities. So, finally, the probability that exactly L other particles inform A is given by

$$\eta(S,K,L) = C_{S-1}^{L}\left(1-\left(\frac{S-1}{S}\right)^{K-1}\right)^{L}\left(\frac{S-1}{S}\right)^{(K-1)(S-1-L)} \tag{7}$$

The mean value we are looking for is then

$$\widehat{L} = \sum_{L=0}^{S-1}\eta(S,K,L) \tag{8}$$

We just have to add 1 for the complete informant group that include the particle itself.

7.4.2 Example

We can apply these formulas to an example with say $S = 35$. For the three curves on 13 (on the left) the mean values are respectively 1.9 for $K = 3$, 7.8 for $K = 10$, and 21.3 for $K = S = 35$. So, although it could intuitively seem that these numbers should be more or less equal to S/K, there are always smaller, even for $K = S$.

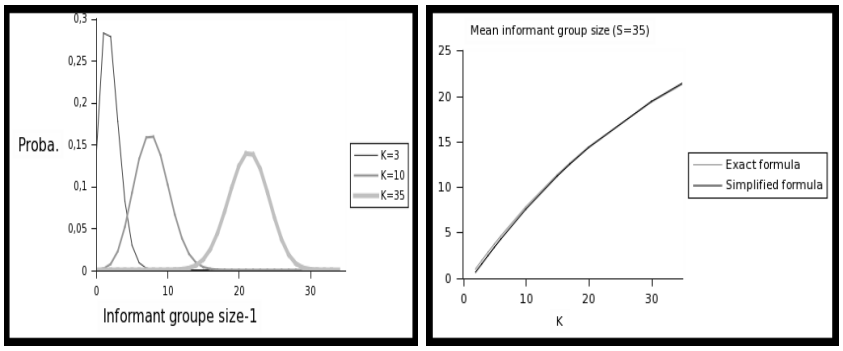

Figure 13: Informant groupe size for different K values (on the left), and mean group size (on the right). The swarm size S is equal to 35. As the links are randomly chosen, even when $K = S$ the mean number of informants of a given particle is significantly smaller than S

7.4.3 Simplified formula

Formula 8 to find the mean informant group size is a bit complicated. We can make an approximation by noting that the curves on figure 13 are almost symmetrical, and the maximum value for $\eta(S, K, L)$ is then reached on a $\widetilde{\widehat{L}}$ value just a bit smaller than $\widehat{L}$. We have to solve

$$\widetilde{\widehat{L}} = MAX_L \left(C_{S-1}^{L} (1-u)^L u^{S-1-L} \right)$$

As the curve is (almost) symmetrical we can write for any p value smaller than $\widetilde{\widehat{L}}$

$$C_{S-1}^{p} (1-u)^p u^{S-1-p} = C_{S-1}^{2\widetilde{\widehat{L}}-p} (1-u)^{2\widetilde{\widehat{L}}-p} u^{S-1-2\widetilde{\widehat{L}}+p}$$

In particular this must be true for $p = \widetilde{\widehat{L}} - 1$. The idea is of course here that if $\widetilde{\widehat{L}}$ is the point where the maximum is reached, the function value must be the same for $\widetilde{\widehat{L}} - 1$ and for $\widetilde{\widehat{L}} + 1$. It gives us immediately

$$\frac{\widetilde{\widehat{L}} \left(\widetilde{\widehat{L}} + 1 \right)}{\left(S - 1 - \widetilde{\widehat{L}} \right) \left(S - 1 - \widetilde{\widehat{L}} + 1 \right)} = \left(\frac{1-u}{u} \right)^2$$

D δ	1	2	3	Total
10	0.037	0.0427	0.002000	0.048
30	0.011	0.0086	0.000074	**0.012**
100	0.003	0.0008	0.000002	0.003
30, bell-shape	0.018	0.0006	0.000039	**0.019**
30, uniform	0.011	0.0007	0.000082	**0.012**

Table 9: Some search powers. For a given dimension (here 30) it seems a bell shape distribution should be better. However in practice it is not always the case, probably for with such a distribution the swarm tends to be trapped in a local optimum

We now just have to solve a second degree equation, and we find

$$\left\{\begin{array}{rcll} \alpha & = & \left(\frac{1-u}{u}\right)^2 & \\ a & = & 1-\alpha & \\ b & = & 1+2\alpha\left(S-1\right) & +\alpha \\ c & = & -\alpha\left(S-1\right)S & \\ \widetilde{\widehat{L}} & = & \frac{1}{2a}\left(-b+\sqrt{b^2-4ac}\right) & \end{array}\right. \tag{9}$$

Now the formula7 is itself quite complicated, so finding its maximum is not so easy. Let's define $u=\left(\frac{S-1}{S}\right)^{K-1}$.

As we can see on figure 13, this simplified formula gives a result that is almost perfect.

7.5 Search power

We can define a partial power search (for one particle) as the product $\rho\left(D,\delta,\Delta\right)\left(1/C_D^\delta\right)$ **i.e. "probability to reach a position at distance** δ**"x"probability for a position to be a that distance". The total power search is then the sum over all** δ **values:**

$$w_1\left(D,\Delta\right)=\sum_{\delta=1}^{\Delta}\rho\left(D,\delta,\Delta\right)/C_D^\delta$$

For S **particles, we find**

$$w_S\left(D,\Delta\right)=1-\left(1-w_1\left(D,\Delta\right)\right)^S$$

i.e. simply $w_1\left(D,\Delta\right)S$ **if** $w_1\left(D,\Delta\right)$ **is small enough.**

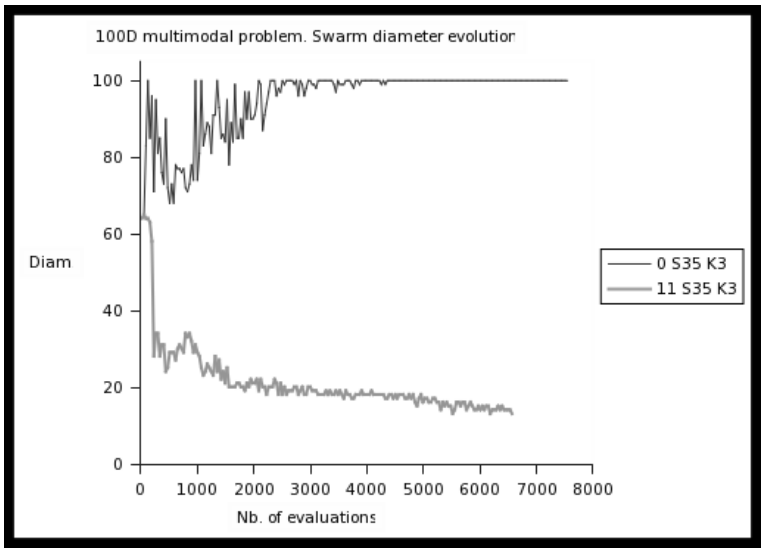

Figure 14: Swarm diameter evolution. When the process converges the swarm tends to "collapse", as intuitively expected. If there is no convergence it tends to cover all the search space, that is to say the diameter tends towards D

7.6 Why $ln(D)$?

7.6.1 Swarm diameter

The precise definition is "the maximum distance between any pair of particles". Actually we can consider two diameters: the one of the memory-swarm, that is to say the swarm of best previous positions [CLE 05], and the one of the current swarm. We are mainly interested in this last one, for we will use it to define an adaptive Δ radius. Let Σ be the swarm. So the formula for the diameter Θ at time t is

$$\Theta(t) = MAX_{\substack{x \in \Sigma \\ x' \in \Sigma}} (h(x, x'))$$

where $h(x, x')$ is the Hamming distance between the positions x and x'.

As the swarm tends to converge, its *diameter* decreases, as we can see on figure 14when Derivation 11 is used. On the same figure there is also the evolution given by Derivation 0. Here the process does not converge at all: the swarm is more and more "spread" all over the search space, and the diameter tends towards the dimension of the binary problem, i.e. the length of the bit string.

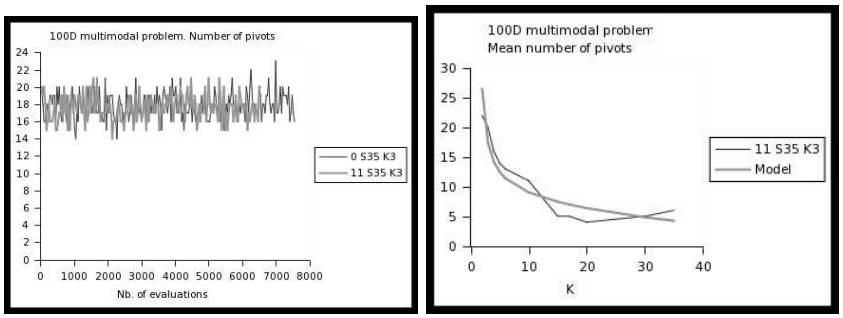

Figure 15: Number of pivots evolution. No matter the process converges or not, it has the same random evolution (left figure). However, as intuitively expected, it decreases when the mean informant group size increases (right figure). The probabilistic model gives a quite good approximation, except that it should be refined for high K values

7.6.2 Number of pivots

Let's call here *pivot* a particle which is the best informant for another one (at a given time step). During the process the number of pivots $\omega(t)$ can easily be computed at each iteration. Intuitively it should decrease whent the mean informant group size increases, and this is indeed true. What is not so intuitive is that it does not depend on the fact that the process convergences or not, as we can see on figure 15 (left part).

Now, during a process we can compute the mean number of pivots, and see how it is modified for different K values on a given problem. Let us call $\widehat{\omega}(S,K)$ this number. A very simplified theoretical approach tell us that is does not depend on the problem, and should be given by something like

$$\widehat{\omega}(S,K) = \frac{1}{2}\left(\frac{S}{\widehat{L}} + \frac{S-\widehat{L}}{2}\right) \simeq \frac{1}{2}\left(\frac{S}{\widetilde{\widehat{L}}} + \frac{S-\widetilde{\widehat{L}}}{2}\right) \tag{10}$$

This formula is obtained just by evaluating a minimum number, a maximum number, and by computing the mean. As we can see on figure 15 (right part) it gives a curve that fits quite well with the real one.

7.6.3 Choosing the Δ radius

For Derivation 11 we used a constant "radius" that is to say a constant maximum number of bits to switch. The formula given is $\Delta = \ln(D)$.

However this is just an empirical oversimplification of a more exact formula, and valid only if the swarm size is near of the optimum. A better formula needs to continuously (i.e. at least at each iteration t) compute the swarm diameter $\Theta(t)$ and the number of pivots $\omega(t)$. It is not very difficult but it spends some computing time.

If $\Theta(t) = D$ (the maximum possible value) the search space is completely "covered" in probability if the DNPP (Distribution on Next Possible Positions) around each pivot has a radius equal to $INT(1 + D/\omega(t))$ (the "1" is here because $\omega(t)$ is usually not an exact divisor of D). More generally, the hypothesis is that at each time step the solution point is still inside (or very near of) the convex envelope defined by the swarm. And then the radius has just to be equal to $INT(1 + \Theta(t)/\omega(t))$. It appears that setting Δ(which we will now write $\Delta(t)$) to this value indeed gives a better algorithm, and it is really worthwhile to try exactly this formula. However, we may have sometimes $\Delta(t) = 1$ or $\Delta(t) = 2$, which are quite bad for some problems, even near of the end of the process. So, a more robust compromise is to use the following formula:

$$\Delta(t) = MAX(3, 1 + \Theta(t)/\omega(t)) \tag{11}$$

Note that the algorithm can now clearly be seen as an adaptive one. As we can see on figure 16 it is significantly better for the Multimodal problem than the one with the constant Δ=$\ln(D)$. Also, as for this last one the best K value was 2, it is now 3, as for most of problems. For the other problems we have studied here there is no real difference.

7.6.4 Saved computational time estimate

Let us suppose the run converges. The idea is that after a while all best known positions are very near of the solution, and all current positions are very near of these ones. For example, for Zebra3 (or Goldberg, Bipolar, Mühlenbein) let us call a *sequence* a k-bits substring that is used to progressively compute the fitness. The hypothesis is then that the best common known positions have just one "bad" sequence, and the current positions have just two. Let n be the total number of sequences. When examinating each sequence of the current position, from 1 to n, the evaluation can be stopped as soon as a bad one is found, for it is then already sure that the current position is not better than the previous best one . We have to compute the elementary probability of the following event:

- the sequence s is bad
- all sequences 1 to $s-1$ are good (for we haven't stopped before)
- we know there are two bad sequences

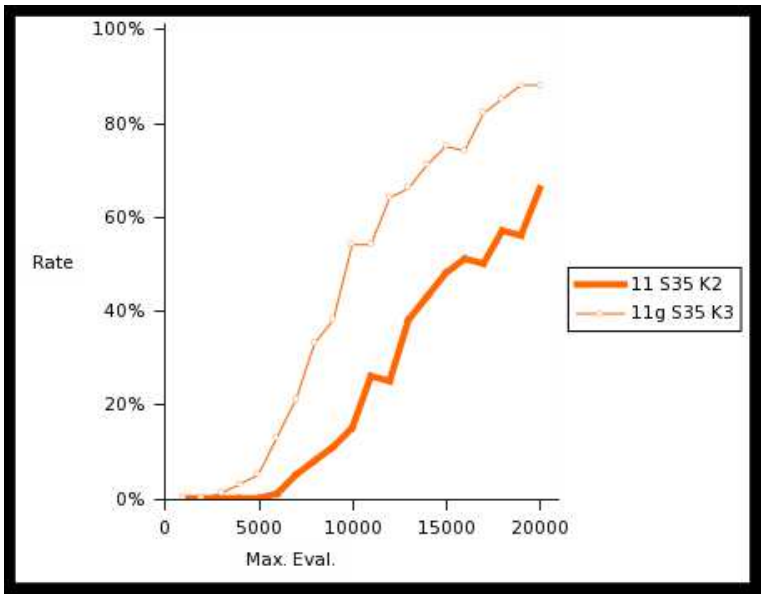

Figure 16: Improvement with an adaptive Δ radius. For the multimodal problem results are far better (Derivation 11g) than with the constant one (Derivation 11). For the five other problems (not shown here) they are similar. Note that the best K value for Derivation 11g is now the "standard" one 3

This probability is given by the formula $\frac{2}{n}\frac{n-s}{n-1}$. In such a case, the rate of saved time is $\frac{n-s}{n}$. Of course, if $s = n$, no time can be saved. Finally the total rate of saved time is given by summing over all possible s values

$$\sigma(n) = \frac{2}{n^2(n-1)} \sum_{s=1}^{n-1} (n-s)^2 = \frac{2n-1}{3n} \tag{12}$$

For Zebra3, whe have $n = D/3 = 10$, which gives $\sigma(10) \simeq 0.63$. For Mühlenbein, we find $\sigma(6) \simeq 0.61$. Both values are quite near of the real ones, as we can see on figure 8 (the averages over the last 20 runs are 0.626 and 0.634).

7.7 The fundamental hypothesis

The underlying hypothesis of PSO is that "nearer is better". If x and y are two positions in the search space, and if $x^*$is a solution point (i. e. the function f to minimise reachs its minimum in x^*), then the hypothesis is the following:

$$distance(x, x^*) < distance(y, x^*) \Rightarrow f(x) < f(y) \tag{13}$$

Note that this hypothesis is also the fundamental one for almost any iterative optimisation algorithm. The more it is true, the more the algorithm can be efficient. For example, if it is ***always*** true, for any x and y, then even a simple pseudo-gradient algorithm would easily find the solution, for it simply means there is no local optima and not "flat" areas. For binary optimisation, as the search space is finite (2^D elements) it is theoretically easy to compute the "truth value" of this hypothesis, by checking all pairs (x, y). If H is the number of pairs that respect the hypothesis 13, then the truth value is simply the rate $H/\left(2^{D-1}\left(2^D - 1\right)\right)$.

In practice, of course, as soon as the dimension is high, one can only have an estimation by sampling at random a given rate of such pairs, say 10%.

Even like that, dimension 30 is far too much for my small computer, so I did it for dimension 12 (or 10 for Muhlenbein, for it has to be a multiple of 5). Results are given in table 10. It appears that this truth value is a better estimation of the practical difficulty for iterative algorithms like PSO than the theoretical one (see below the definition), at least for the simplest form, with just one particle. Therefore it was tempting to define a Trap function that does *not* respect the hypothesis, in order to deceive the algorithm.

7.7.1 Theoretical difficulty

It is just the probability to find a solution by chance ([BAR 03]). As this value is usually very small (for example 2^{-12} for Goldberg 12D), comparisons are easier to do by using the opposite of the logarithm ([CLE 05]) . So, for example, for Goldberg 12D the difficulty is $-ln\left(2^{-12}\right) = 12ln\left(2\right) = 3.6$. For Bipolar 12D, which has 4 solutions, the difficulty is $12ln\left(2\right) - ln\left(4\right) = 3.0$.

7.7.2 Trap function

The idea is to design a test function that does not respect at all the fundamental hypothesis, or as least, as few as possible. Let x be a position, i.e. a bit string. As previously, $|x|$ denotes the sum of the bits. Then, the fitness f of our Trap function is defined by

$$f_{Trap}\left(x\right) = \left\{ \begin{array}{cc} 0 & if \quad |x| = 0 \\ 1/\left|x\right| & else \end{array} \right.$$

For almost all (x, y) pairs "farer is better". The hypothesis is valid only for pairs with x or y equal to the solution $000...000$. It is easy to compute that the truth value of the hypothesis is then $1/\left(2^D - 1\right)$. As expected, all the methods we have seen fail to find the solution (except, of course, the "hoax" Derivation 0, for it is precisely designed to easily find the solution $000...000$).

	Truth value	Success rate Derivation 11, S1 K1 800 evaluations	Theoretical difficulty
Goldberg 12D	0.15	69%	3.6
Bipolar 12D	0.54	90%	3.0
Zebra3 12D	0.06	46%	3.0
Muhlenbein 10D	0.08	63%	3.6
Trap 12D	0.00024	0%	3.6

Table 10: Truth value of the hypothesis "Nearer is better". For each function, the value has been estimated by sampling 10% of all possible pairs of positions. The truth value estimate the practical difficulty for an iterative algorithm like PSO, which has nothing to do with the theoretical one