HAL
open science

When Mating Improves On-line Collective Robotics

Amine Boumaza

To cite this version:

Amine Boumaza. When Mating Improves On-line Collective Robotics. GECCO'19 Proceedings of the 2019 Genetic and Evolutionary Computation Conference, Jul 2019, Prague, Czech Republic. hal-02185645

HAL Id: hal-02185645

https://inria.hal.science/hal-02185645v1

Submitted on 16 Jul 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

When Mating Improves On-line Collective Robotics

Amine Boumaza
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
amine.boumaza@loria.fr

Initial draft: December 17th, 2018
Updated: July 16, 2019

1 Introduction

Since the seminal work of Ficici *et al.* [10], Embodied Evolutionary Robotics (EER) as a field has greatly matured. Its main objective is two-fold: on the one hand it provides an on-line distributed learning method for designing swarm behaviors and, on the other hand, it allows making predictions and verifying hypothesis stemming from natural or artificial life models [4]. There has been a lot of work on both ends by the research community, furthermore there has also been some work to improve and deepens our understanding of the algorithm themselves.

In a nutshell, EER are algorithms in which evolution is carried out in a decentralized manner. Each agent, typically a mobile robot, runs an EA onboard and exchange genetic material with other agents when they meet. Selection and variation are performed locally by the agent. As such, there is no central process that governs evolution, in contrast with traditional evolutionary robotics (ER).

Analyzing and understanding these algorithms present many challenges some of which exist also in traditional ER and some of which are inherent to their distributed nature and the fact that they operate online and on which many questions remain open. In this paper, we attempt to contribute to this latter issue by studying the effect of recombination or what is commonly known as crossover in these algorithms.

From the theoretical work on evolution strategies (ES) [3], it has been shown that recombination is a central component of these algorithms. It allows the strategy to speed up its progress toward good solutions and to improve robustness against selection errors. These benefits were discussed in [2, 3], where the author argues that recombination is beneficial due to the *genetic repair principal* (GR) which reduces the effect of the harmful components of mutation as a result of the averaging process. This can be seen as a compensation of the statistical error rather than the combination of good traits, as advocated by the *building bloc hypothesis*. As a consequence of GR, recombinant evolution strategies are able to cope and operate under larger mutation steps than their non recombinant counterparts. Furthermore, the benefit of GR has also been emphasized in the presence of noise in fitness evaluations where [1] argues that, combined with higher mutation steps, recombination can reduce the signal-to-noise ratio in the evaluation. This is of particular interest in EER, where evaluations are subject to noise and evaluation conditions are dynamic.

Recombination is not new in on-line EER, different authors have proposed implementations with different forms of crossover. To our knowledge however, none attempted the weighted intermediate recombination from ES. We briefly review some of that work in the following. For a more comprehensive review, we recommend [4].

In the Probabilistic Gene Transfer Algorithm (PGTA) [18], as the name suggests, recombination is implemented at a gene level. Agents broadcast randomly sampled genes from their genomes and when they receive genes from other agents, they replace the corresponding gene with a probability. The rate at which the agents broadcast their genes is proportional to their energy level (fitness) and conversely, the rate at which they accept a received gene is inversely proportional to their energy level. This way, selection

pressure is introduced in the fact that fit agents transmit their genes to unfit ones.

In the Embodied Evolutionary Algorithm (EEA) [13], agents select a gnome from their local population (received from potential mates) using a binary tournament. This selected genome is then recombined with the current active genome with a probability proportional to the ratio of its fitness and that of the currently active genome. The newly created genome is the average of both parents. In this case, the fitter the selected genome is, the higher is the recombination probability. More recently [12] used a uniform crossover between a genome selected from the local population and the active genome. In [14], when agents meet, they exchange genetic material and accept the partner's genome if it is fitter. On replacement, the newly generated genome is obtained using a binary crossover.

The above mentioned articles used fixed topology controllers where the controller elements need not be matched prior recombination due to the competing conventions problem [15]. Few implementations adapted the innovation marking introduced in the Neuro-Evolution of Augmenting Topologies algorithm [17], to the distributed case. In this case, innovations appear locally in the population and are not known to all agents who must order them correctly before performing a crossover [16, 9].

1.1 Objectives

EER algorithms are different from their traditional ER counterparts in many aspects. We will emphasize our discussion on two aspects. On the one hand, in addition to learning the task, solutions must also spread their genetic material in order to survive. On the other hand solutions get evaluated in different conditions. The consequence of the first aspect is obvious, solutions can only overcome environmental selection pressure, if they manage to spread copies of themselves to other vehicles, having a good fitness is not sufficient. The consequence of the second aspect is less obvious. When solutions manage to survive and are reevaluated on other agents, their behaviors will be different due to the interactions with the environment. Therefore, having been evaluated favorably on one agent, does not guarantee survival, the solution must be robust to noisy evaluations. In a sense EER can be viewed as constantly searching for an equilibrium between finding solutions that are fit and robust and that can spread to ensure survival.

We propose the $(\mu/\mu_{\mathrm{W}}, 1)$-On-line EEA which adapts the well-known weighted intermediate recombination [11] scheme to on-line EER settings. Since this operator was designed for real vectors, we consider controllers with a fixed topology and evolve only their weights. We study the impact of different variants of this recombination scheme on different collective robotics scenarios. More precisely, we would like to show if the observed effects of recombination in ES can also benefit EER.

We state the following hypotheses claiming that recombination:

H.1 allows the algorithm to reach better solutions than without, and,

H.2 reduces the effect of an inappropriately chosen mutation step.

We will start by describing the $(\mu, 1)$ and introduce its recombinant counterpart on two variants $(\mu/\mu_{\mathrm{W}}, 1)$ and $(\mu/\mu_{\mathrm{D}}, 1)$. We then describe our experimental procedure along with the tested scenarios, and conclude with the discussion of the results.

2 Methods

2.1 The $(\mu, 1)$-On-line EEA

The main inspiration of the $(\mu, 1)$-On-line EEA (Algorithm 1) is the original version of mEDEA [5]. The algorithm considers a swarm of λ mobile agents a^j with $j = 1, \ldots, \lambda$ each executing a neuro-controller whose parameters are x^j (the active genome). Each agent maintains a list L^j, initially empty, in which it stores other genomes that it receives from other agents.

At each time step $t < t_{\max}$, an agent executes its active controller and broadcasts its genome within a limited range. In parallel, it listens for genomes originating from other agents, and when a genome

is received (a mating event), it is stored in the agent's list L^j (its local population). This procedure is executed in parallel on all agents during $t_{\max}$ steps, the evaluation period or one generation.

At the end of a generation, the agent selects a genome from its list L^j (the selection step), and replaces its active genome with a mutated copy of the selected one. The list is then emptied and a new generation begins.

In the event where an agent had no mating opportunities and finishes its evaluation period with an empty list $L^j = \emptyset$, it becomes inactive; a state during which the agent is motionless. During this period, the inactive agent continues to listen for incoming genomes from other agents, and once $L^j \neq \emptyset$ the agent becomes active again at the beginning of the next generation.

The number of genomes the agents collects $\mu^j = |L^j|$ $(0 \leq \mu^j \leq \lambda)$ is conditioned by its mating encounters. Since the communication range is limited, agents that travel long distances will increase their probability of mating. We should note that mating encounters allow agents to spread their active genome and to collect new genetic material. The algorithm stores only one copy of the same genome[1] if it is received more than once.

Since the algorithm runs independently on each agent, fitness values are assigned to the individual agents based on their performance with regard to the given task. These values are continuously updated during the agent's lifetime (line 11). Each agent stores this value internally and broadcasts it along with its genome during mating events. Agents on the receiving end store the genome and its fitness in their lists. If an agent receives an already seen genome, it updates the genome's fitness as it is the most up to date value. Furthermore, to ensure that genomes are transmitted with accurate fitness values, a *maturation age* is required of the agent before broadcasting [19, 13]. This ensures that the fitness is sufficiently updated before transmission. Finally, we should emphasize that fitness evaluations are intrinsically noisy since measurements are performed in varying circumstances.

The $(\mu, 1)$-On-line EEA can be viewed as λ instances of $(|L^j|, 1)$-EA running independently in parallel, where for each instance, $L^j \subseteq \{x^1, \ldots, x^\lambda\}$, *i.e.* each local population is a subset of the set of all active genomes of the generation. Furthermore, the sizes of the individual local populations μ^j are not constant although bounded by λ and depend on the number of mating events (possibly none) the agent had in the previous generation. The selection procedure (line 16) is based on the fitness of the genomes in

Algorithm 1: $(\mu, 1)$-On-line EEA.

```
1  for 1 ≤ j ≤ λ in parallel do
2      x^j ← random()
3      a^j is active
4  repeat
5      for 1 ≤ j ≤ λ in parallel do
6          t ← 0, f^j ← L^j ← ∅
7          while t < t_max do
8              t ← t + 1
9              if a^j is active then
10                 execute(x^j)
11                 update(f^j)
12                 if t > τ t_max then
13                     broadcast(x^j, f^j)
14             L^j ← L^j ∪ listen()
15         if L^j ≠ ∅ then
16             x^j ← mutate(select(L^j))
17         else a^j is not active
18 until termination condition met
```

[1] The term same is here used in the sense "originating from the same agent".

the individual lists. In this work, the genome is a vector $x \in \mathbb{R}^N$, which represents the weights of the neuro-controller and $f\colon \mathbb{R}^N \to \mathbb{R}$. Only the weights undergo evolution (fixed-topology). We will note $L = \{x_1 \dots x_\mu\}$ the local population on some agent, and we consider a maximization scenario where x_i is fitter than x_k imply $f(x_i) \geq f(x_k)$.

2.1.1 Selection in $(\mu, 1)$-On-line EEA

The next active genome is the winner of an adapted k-tournament selection scheme which randomly draws $k > 0$ solutions from the local population L of μ parent solutions and returns the best solution among the parents. The tournament size k, sets the selection pressure of the procedure, and is usually a parameter of the EA. However, since the size of the parent population varies from agent to agent and from one generation to another, the size of the tournament cannot be chosen beforehand. In practice, we fix k as a function of μ and in its simpler form this function can be a linear projection such as $k = \lfloor \alpha\mu \rfloor + 1$ with $\alpha \in (0, 1)$, but more sophisticated functions could also be considered. In this case, α is the parameter that tunes the selection pressure. We present results for the following variants $\alpha \in \{0, 0.25, 0.5, 1.0\}$, where the case $\alpha = 0$ (no selection pressure) is similar to the original mEDA algorithm.

The selected genome, which we note $\bar{x}$, is then mutated to generate the next genome. Mutation is Gaussian with a fixed step size $\sigma \in \mathbb{R}$:

$$x := \bar{x} + \sigma^2 \times \mathcal{N}(1, 0) \tag{1}$$

2.2 Adding recombination

The only difference between algorithm 1 and $(\mu/\mu, 1)$-On-line EEA, its variant with recombination, lies in the generation of the next active genome. When recombination is considered, all the genomes in the local list undergo mutation before recombination. Line 16 is replaced by the lines in Algorithm 2 in which mutation is identical to eq. 1. We consider two types of recombination schemes that are detailed below. All individuals in L^j participate in the recombination, one can also implement truncation by selecting a

Algorithm 2: $(\mu/\mu, 1)$-On-line EEA.

```
15 ...
16 foreach x_i ∈ L^j do
17     x_i ← mutate(x_i)
18 x^j ← recombine (L^j)
19 ...
```

portion of the list containing the most fit elements [2] this allows to further increase selection pressure.

2.2.1 Weighted intermediate recombination

This recombination scheme is similar to weighted intermediate recombination from CMA-ES [11]. The newly generated genome is defined as:

$$\bar{x} = \sum_{i=1}^{\mu} w_i \, x_{i:\mu}, \text{ with } \sum_{i=1}^{\mu} w_i = 1, \text{ and } w_1 \geq w_2 \geq \ldots \geq w_\mu > 0,$$

where $w_{i=1\ldots\mu} \in \mathbb{R}^+$ are the recombination weights and the index $i{:}\mu$ denotes the i-th ranked individual and $f(x_{1:\mu}) \geq f(x_{2:\mu}) \geq \ldots \geq f(x_{\mu:\mu})$. By assigning different values to w_i it can also be considered as a selection scheme since fitter solutions have greater weights in the sum. We call this version of the algorithm $(\mu/\mu_{\mathrm{W}}, 1)$-On-line EEA.

[2] Truncation should take into account variable population sizes similarly as in tournament size.

In [11], the weight generation function was defined as $w_i = \frac{w'_i}{\sum_{k=0}^{\mu} w'_k}$ where $w'_i = \log(\mu + 0.5) - \log i$. In our implementation, we chose to define a more flexible weight generation function which can be tuned to specify a desired selection pressure. Let $g(x) = (1-x)^\beta$ with $g\colon \mathbb{R} \to \mathbb{R}$ and $\beta \in \mathbb{R}^+$, and $h(i) = (i-1)/\mu$ with $i = 1, \ldots, \mu$, we define the weight generation function as:

$$w'_i = (g \circ h)(i) \qquad i = 1, \ldots, \mu$$

and normalize to make sure the sum is 1. We emphasize that g is monotonically decreasing and positive on $(0, 1)$, to ensure that $w'_i \geq w'_{i+1}$ and that $w'_i \geq 0$. Furthermore, β allows to define the *profile* of g which in turn define a selection pressure: the amount of weight we give to the fittest individuals (Figure 1 left). Note that when $\beta = 0$, all weights are equal and there is no advantage given highly ranked individuals, we can consider this as a case where there is no selection pressure. We present results for the following variants $\beta \in \{0, 1, 4\}$ and the original log rule from [11].

2.2.2 Dominant recombination

This recombination scheme [3] is closely related to recombination in PGTA [18]. In this version, offspring are composed of genes originating each from different agents. The newly generated genome is constructed gene by gene, where each gene at a given locus is selected among the parents in the local population at the same locus. We call this version of the algorithm $(\mu/\mu_{\mathrm{D}}, 1)$-On-line EEA.

We test different variants, each parent in $L = \{x_1, \ldots, x_\mu\}$ is assigned a selection probability: 1) proportionate to its fitness, 2) proportionate to its rank and 3) uniform. If we note the locus k of a genome x_i as $x_i^{[k]}$ then we write these probabilities as:

$$Pr\left(\bar{x}^{[k]} = x_i^{[k]}\right) = \begin{cases} \dfrac{f(x_i)}{\sum_{j=0}^{\mu} f(x_j)} & \text{fitness prop.} \\ \dfrac{1 - \exp^{(-(i:\mu))}}{c} & \text{rank prop.} \\ \dfrac{1}{\mu} & \text{uniform.} \end{cases}$$

where $(i{:}\mu)$ is the rank of x_i and c is a probability normalizing constant. The last instance corresponds to the case where there is no selection pressure towards fit individuals.

2.3 The effect of variable population sizes

The fact that the size of the local lists L varies from agent to agent depending on their respective mating encounters, affect the sampling process that generate the new active genome, which in turn affects the selection pressure applied on the agent level. It is higher on agents with small population sizes.

For example, in $(\mu, 1)$, the probability of selecting the best solution from the list by tournament is $\alpha < \frac{\lfloor \alpha\mu \rfloor + 1}{\mu} \leq \alpha + \frac{1}{\mu}$. This probability is higher on agents with small populations than it is on agent that have larger ones. This is also the case of $(\mu/\mu_{\mathrm{D}}, 1)$: the probability of selection the fittest individuals is higher on smaller population.

Variable population sizes also affects $(\mu/\mu_{\mathrm{W}}, 1)$, even thought selection is a byproduct of the weight values and not a sampling process. The weights w_i may also be different from agent to agent due to their normalization (Figure 1 right). And here again the EA gives a selection advantage to agents that had less mating encounters.

The importance of this fact remains an open question. We conjuncture that this is a minor issue since over time its effects may be smoothed out. Nevertheless, this side effect suggests that the algorithm naturally gives an advantage (an incentive or a chance to survive) to genomes that performed poorly (whose agents had the less mating encounters) which could be a desirable feature.

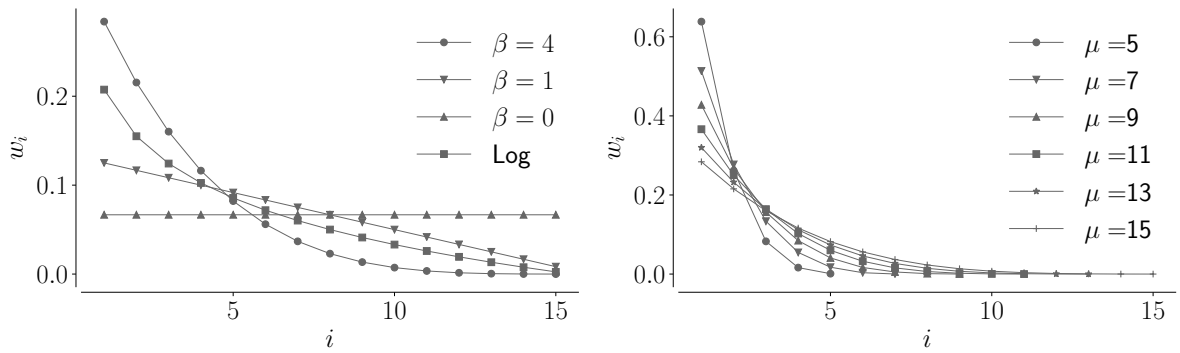

Figure 1: Recombination weights with different generating functions (left), same generating function for different population sizes (right).

2.4 Learning tasks

We considered three different learning tasks: locomotion, item collection and item foraging (Figure 2). All tasks share the same circular environment with the same obstacles (rectangles in the figure). In the collection and foraging scenarios, items (red dots in the figure) are randomly placed in the environment.

Agent perceive obstacles within their sensing range using 12 range sensors and an additional 12 sensors that are high when the obstacle is a wall. All sensors are evenly placed around the agent's body and measure distances on a line of sight. Agents move using 2 motor effectors (differential drive motion). The neuro-controller we consider in this work is a simple feed-forward multi-layered perceptron with one hidden layer that maps sensory inputs (values in $[0, 1]$) to motors outputs (values in $[-1, 1]$). This is the base architecture shared by all learning tasks. Additional sensors and effectors depending on the task are detailed in the following.

Finally, we define *learning assessment* measures, m_1 and m_2, that give insights on how well the task is performed. They take values in $[0, 1]$, where large values reflect good behaviors. These measures are not used in the evolutionary process, although they are recorded during evolution.

2.4.1 Locomotion

In this scenario, there are no additional sensor or effectors and the goal of the swarm is to learn to travel long distances. The fitness function in this case is the distance traveled since the beginning of the generation. To measure how well the agent perform we compute:

$$m_1 = \frac{1}{t_{\max}} \sum_{t=1}^{t_{\max}} |v^t| \qquad \text{and} \qquad m_2 = \frac{1}{t_{\max}} \sum_{t=1}^{t_{\max}} 1 - |\omega^t|$$

where v^t and ω^t are the normalized instantaneous translation and rotation velocities. When the agent travels at maximum speed in a straight line $m_1 = m_2 = 1$.

2.4.2 Collection

In this scenario, the goal is to maximize the number of collected items. The fitness function is simply the number of items collected since the beginning of the generation. Agent must bump on items to collect them, and when collected, the item reappear at some random location. There is always the same number of items in the environment. For this task, 12 additional sensors are added to the agents that are active if the sensed obstacle is an item. For this task:

$$m_1 = \frac{1}{t_{\max}} \sum_{t=1}^{t_{\max}} c^t \qquad \text{and} \qquad m_2 = \frac{1}{t_{\max}} \sum_{t=1}^{t_{\max}} \cos(s^t)$$

where $c^t \in \{0, 1\}$ depending on whether the agent collected an item or not[3], s^t is the angle between the most activated item sensor and the heading of the agent. m_2 measures how well the agent aims at the sensed items.

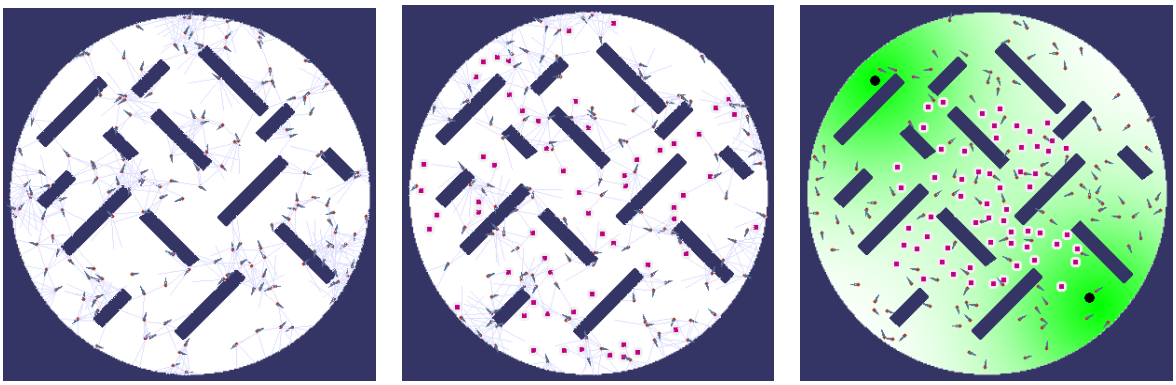

Figure 2: An overview of the Roborobo simulator, locomotion (left) collection (center) and foraging (right).

2.4.3 Foraging

This last task is the most difficult among the three. The goal here is to collect items and deposit them the closest to one of two specified locations (black circles in Figure 2 right). Using the ant foraging metaphor, a constant pheromone gradient (green color level in the figure) is present in the environment whose value is the highest (1) at the goal locations and lowest (0) at the lighter areas.

Agent can carry more than one item at a time up to a given limit, their maximum basket capacity. In addition to the item sensors described for the previous task, agents have 12 additional pheromone sensors. At each time step, only the one on the highest pheromone value is active all the rest remain inactive, giving a sense of direction of the pheromone concentration. Agents have one extra sensor that gives the number of items in the basket (0 meaning empty and 1 meaning full).

Finally, to deposit the items, agents must make a decision to drop one or more items at their current position. For that they have as many effectors as the maximum capacity of their basket, each taking values in $[-1, 1]$. The number of effectors that output a positive value is the number of items dropped. The items reappear at some random location as in the previous task.

The fitness function is updated at each deposit decision, and is computed as the number of dropped items multiplied by the pheromone value at the drop location. For this task:

$$m_1 = \frac{1}{t_{\max}} \sum_{t=1}^{t_{\max}} \begin{cases} cos(s^t) & \text{if basket empty} \\ cos(p^t) & \text{otherwise} \end{cases}$$

where s^t it the same as in the previous task and p^t is the angle between the most activated pheromone sensor and the heading of the agent. And,

$$m_2 = \frac{1}{t_{\max}} \sum_{t=1}^{t_{\max}} 2(\phi^t)^2 d^t - ((\phi^t)^2 + d^t) - 1$$

where ϕ^t is the pheromone value at the drop position, and d^t is the number of dropped items normalized by the maximum capacity. m_1 measures how well the agent aims at the sensed items or the nest depending on if it is carrying items or not, and m_2 measure the suitability of the dropping decision.

[3]The value of m_1 is equal to the average fitness in one time-step.

3 Results

All the results, data analysis scripts and the source code of the implementations of the algorithms are publicly available.[4]

3.1 Experiments

The experiments were performed on the Roborobo simulator [6] an environment that allows us to run experiments on large swarms of agents[5]. In this simulator, agents are e-puck like mobile robots with limited range sensors and two differential drive wheels. We emphasize, that our main goal is to compare algorithm instances, and not solve the tasks optimally. All algorithms are run on the exact same environment, with the same conditions. All the parameters of the experiments are summarized in Table 1.

3.2 Measures

To assess of the performance of an algorithm, we use the fitness, it variance, and both learning assessment measure m_1 and m_2. At each generation, these values are summed over the entire population (except for the variance) to summarize the performance of the swarm in one run of a given algorithm on a given scenario. For example, the fitness and variance are measured as:

$$\hat{f}(g) = \sum_{j=1}^{\lambda} f^j(g), \quad \mathrm{Var}[f](g) = \frac{1}{\lambda}\sum_{j=1}^{\lambda}\left(f^j(g) - \frac{\hat{f}(g)}{\lambda}\right)^2, \quad \text{for } g = 1, \ldots, g_{\max} \tag{2}$$

where $f^j(g)$ is the fitness of agent j at generation g. $\hat{f}(g)$ indicates the total fitness of all the agents and $Var[f](g)$ indicates how the fitness values of the individual agents are spread at generation g.

To compare all three algorithms and their variants, each algorithm is executed 30 times, and we compare the above measures using median and the percentiles. Furthermore, we compare the algorithms using a Mann-Whitney U test with the null hypothesis being "*the distributions of results of both algorithms are equal*". Here again we employ a summation to compress the results of one run of one algorithm instance in one value. For instance in the case of fitness:

$$\overline{f} = \sum_{g=1}^{g_{\max}} \hat{f}(g) \tag{3}$$

This is done for all measures, and their distributions (30 values) are compared in the statistical test. We establish that instance "A" outperforms instance "B" on a given measure, if the median value for instance "A" is better than for "B" and there is significant statistical difference between the distributions. The level of statistical significance is indicated on the figures by the number of asterisks where for example 4 asterisks indicate a p-value $< 10^{-4}$.

3.3 Discussion

Before discussing the result, we give short recap of the algorithms and their variants we tested, this is shown on Table 2. Figures 3, 4 and 5 present the results. The curves illustrate fitness values $\hat{f}(g)$ during evolution (left column) and box-plots of $\overline{f}$ of 30 independent runs. On the rightmost column, we show the scatter plot of m_1 and m_2 (the closest to the top right corner, the better is the behavior). The size of the colored circles is scaled according to $\overline{f}$, and the gray ellipses indicate where 50% (inner ellipse) and 90% (outer ellipse) of the points lie. The black dot being the median.

[4] `https://gitlab.inria.fr/boumaza/public-code/`

[5] Roborobo3 at commit `95702593ae37d2a2b313fd0f084f98a6dab081f6`

Table 1: Simulation parameters.

Arena diam. (pix)	400	λ	120
$t_{\max}$	600	$g_{\max}$	300
Sens./com. range	32	α	$\{0, 0.25, 0.5, 1\}$
Agent/item diam.	6	β	$\{0, 1, 4\}$
Max trs. vel. (pix.)	2 / tic	σ, σ^0	0.25
Max rot. vel. (deg)	30 / tic	(σ^-, σ^+)	(0.01, 1.1)
Init. weights	$[-1, 1]$	γ	0.35
Nb. runs	30	Mat. Time	$0.8 \times t_{\max}$
	Loc.	Col.	For.
Nb. items	-	$\lambda/2$	$\lambda/2$
Basket Size	-	-	5
Sensors/in	24	36	49
Effector/out	2	2	7
Hid. Neuron	5	5	10
N (genome size)	135	195	570

Table 2: Algorithm variants.

Algorithm	**Variants**
$(\mu, 1)$	$\alpha \in \{0, 0.25, 0.5, 1\}$
$(\mu/\mu_{\mathrm{D}}, 1)$	fitness proportional, rank proportional, uniform
$(\mu/\mu_{\mathrm{W}}, 1)$	$\beta \in \{0, 1, 4\}$, log

3.3.1 Comparing Variants

Before comparing algorithms, we first look at their variants (Figure 3), which will allow us to choose the variant that performs the best for the rest of the comparison.

In the case of $(\mu, 1)$, the selection pressure (the tournament size) is varied from less selective to most selective. Overall, in all three scenarios, the more selective the strategy is, the higher its fitness is. This result is not new, at least for locomotion and collection, it has been reported by several authors, see for example[6] [8]. Interestingly, if we look at m_1 and m_2 there is no statistical significance ($p - \text{value} > 0.05$) between $\alpha = 0.5$ and $\alpha = 1$.

In the case of $(\mu/\mu_{\mathrm{D}}, 1)$, the instance that selects genes proportionally to the fitness of their host genome, outperforms all other instances in all scenarios. The difference is significant on all measures. On the other hand, rank-based selection performs as badly as uniform selection.

Finally, in the case of $(\mu/\mu_{\mathrm{W}}, 1)$, except for case where the recombination weights are equal, all other instance performed almost equally (although statistically different). Fitness-wise, the overall winner is the variant ($\beta = 4$). However, on m_1 and m_2, it does not win by much.

We should note that there is a high fitness variance in locomotion than in the other tasks. This is probably due to the fact that fitness is updated at a higher frequency (every time step) in locomotion whereas only when collecting or dropping items in the other tasks, which smooths the values. The second observation we can make, is that in all task, the algorithm variant that is the most selective performs better. This is due to the fact that selection pressure does not affect ERR algorithms similarly as in traditional ER algorithms[7]. Agents act as quasi separate islands on which selection is performed after which the population are emptied. Although there may be a high selection pressure locally within agents, globally on the swarm this is not the case. Finally, foraging is the task for which the results are the less discriminant. It is the hardest task (sequential decision making) and our choice of fitness function or controller architecture[8] might not be the best. But for the purpose of this work it is sufficient.

[6] Even though we use a different fitness function for locomotion, the result holds.
[7] Where selection is performed from one population.
[8] Feed-forward no recurrent connections.

3.3.2 Verifying claim H.1

In order to establish if recombination improves the quality of the solution, we compare the best variants of each algorithm *i.e.* $(\mu, 1)$ with $\alpha = 1$ (the highest selection pressure), $(\mu/\mu_{\mathrm{D}}, 1)$ with fitness proportional selection and $(\mu/\mu_{\mathrm{W}}, 1)$ with $\beta = 4$. Figure 4 shows the results. When comparing fitness values, we observe clearly that $(\mu/\mu_{\mathrm{W}}, 1)$ outperforms and by far both $(\mu, 1)$ and $(\mu/\mu_{\mathrm{D}}, 1)$ on all three scenarios this behavior is also observed on the scatter plots. This establishes the fact that recombination can improve the results, although not any recombination, only weighted intermediate.

3.3.3 Verifying claim H.2

Now that we established that recombination can improve solution quality in the tested scenarios, let consider our second claim. *i.e.* recombination reduces the effect of badly chosen mutations steps. In other to test this hypothesis, we compare recombination with a fixed step-size to a self-adaptive version to $(\mu, 1)$-On-line EEA with a σ-update rule [5]. Each genome has its own step-size σ whose initial values is σ^0. When broadcasting, in addition to the genome and its fitness, agents broadcast their step-sizes. On the receiving end, when a genome is selected from the local list it is mutated using its σ. Updates take place on the sending end, where at each broadcast, the agent transmits with equal probability (0.5) either an increased or a decreased value of its genome step-size. The motivation behind this rule is that the most adapted value between the two, will survive and spread in the swarm. The update rule is defined as:

$$\sigma = \begin{cases} \min\left(\sigma(1+\gamma),\ \sigma^{+}\right) & \text{if increase} \\ \max\left(\sigma(1-\gamma),\ \sigma^{-}\right) & \text{if decrease} \end{cases}$$

where σ^+ and σ^- are the allowable upper and lower bounds (Table 1). This version of the algorithm is dubbed $(\overline{\mu}, 1)$.

The results (Figure 5) show that, not only $(\overline{\mu}, 1)$ outperforms it non self-adaptive counterpart, it performed better than $(\mu/\mu_{\mathrm{W}}, 1)$ (on locomotion and collection). The difference is not that clear when we look at the curves, it is however statistically significant. Both these tasks are considered easy and do not require learning complex behaviors. On the foraging task the results are reversed, $(\overline{\mu}, 1)$ does not win, but it increased its performance by a large factor in comparison to $(\mu, 1)$. This suggests that in higher dimensional search spaces, recombination with a fixed step size can perform a slightly better than a strategy with an adaptive step size. From the values of $\overline{f}$, $m1$ and $m2$, we see that both $(\mu/\mu_{\mathrm{W}}, 1)$ and $(\overline{\mu}, 1)$ produce behavior that are roughly similar. Furthermore, if we look at the values of the step-size, we observe that they converge to a low value which suggests a convergent behavior of $(\overline{\mu}, 1)$.

The poor performance of $(\mu, 1)$ could thus be explained by the value of the fixed step-size that was wrongly fixed not allowing the algorithm to converge. The same fixed step-size with which $(\mu/\mu_{\mathrm{W}}, 1)$ reached much better results. [7] emphasized that mutation step size adaptation is important in EER settings. Although they used a different adaptation rule, our result is coherent with theirs.

The results on the foraging scenario could also be explained by the fact that the combination of noise and the size of the search space, might require a longer time for the adaptation rule to find optimal values for σ. A less conservative rule might perform better. Further, [1] states that in the presence of noise, higher mutation steps and recombination reduce the signal-to-noise ratio which improves the robustness against selection errors. This could explain the performance of $(\mu/\mu_{\mathrm{W}}, 1)$, the averaging effect of recombination reduces the effect of the inadequate mutation step. Perhaps a related argument could be made regarding the adaptation of σ. Upon convergence, lower step size values create a noise smoothing effect, however, they also reduce the chance to reach better solutions in high dimensional spaces, something that $(\mu/\mu_{\mathrm{W}}, 1)$ could do with its higher step size. A better σ adaptation rule, one that integrates the progress history, could avoid this premature convergence. This can however be a very challenging task in EER settings.

4 Conclusions

It has been argued that recombination is beneficial in evolution strategies because of the genetic repair principle. The main idea in this work was to test if these could also be observed in EER. We proposed $(\mu/\mu, 1)$-On-line EEA that implements two types of recombination and studied its behavior on three different tasks. The results suggest that it is the case at least for intermediate weighted recombination: solutions are improved significantly and the algorithm can in fact cope with inadequately set mutation steps. At this stage, we can identify different research directions to pursue. Since recombination performs an averaging, does it have an effect on the genetic diversity of the population? Could we design a step-size adaptation mechanism that could be applied in the case of recombination and that would help further convergence? Finally, to further strengthen these results, other scenarios should be considered.

References

[1] Dirk V. Arnold. *Noisy Optimization With Evolution Strategies.* Springer, 2002.

[2] Hans-Georg Beyer. Toward a theory of evolution strategies: On the benefits of sex - the $(\mu/\mu, \lambda)$ theory. *Evolutionary Computation*, 3(1):81–111, 1995.

[3] Hans-Georg Beyer. *The Theory of Evolution Strategies.* Springer-Verlag Berlin Heidelberg, 2001.

[4] Nicolas Bredeche, Evert Haasdijk, and Abraham Prieto. Embodied evolution in collective robotics: A review. *Front, in Robo. and AI*, 5:12, 2018.

[5] Nicolas Bredeche and Jean-Marc Montanier. Environment-driven Embodied Evolution in a Population of Autonomous Agents. In *Proc. PPSN 2010*, pages 290–299, Krakow, 2010.

[6] Nicolas Bredeche, Jean-Marc Montanier, Berend Weel, and Evert Haasdijk. Roborobo! a fast robot simulator for swarm and collective robotics. *CoRR*, abs/1304.2888, 2013.

[7] Agoston E. Eiben, Giorgos Karafotias, and Evert Haasdijk. Self-adaptive mutation in on-line, on-board evolutionary robotics. In *Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshop*, pages 147–152, 2010.

[8] Iñaki Fernández Pèrez, Amine Boumaza, and François Charpillet. Comparison of selection methods in on-line distributed evolutionary robotics. In *Proc. of Alife'14*, pages 282–289, New York, 2014. MIT Press.

[9] Iñaki Fernández Pèrez, Amine Boumaza, and François Charpillet. Decentralized innovation marking for neural controllers in embodied evolution. In *Proc. of GECCO '15*, pages 161–168, Madrid, 2015. ACM.

[10] Sevan Ficici, Richard Watson, and Jordan Pollack. Embodied evolution: A response to challenges in evolutionary robotics. In *Proc of the Eighth Europ. Wrks. on Learning Robots*, 1999.

[11] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evol. Comput.*, 9(2):159–195, 2001.

[12] Jacqueline Heinerman, Dexter Drupsteen, and Agoston E. Eiben. Three-fold adaptivity in groups of robots: The effect of social learning. In *Proc. of GECCO'15*, pages 177–183. ACM, 2015.

[13] Giorgos Karafotias, Evert Haasdijk, and Agoston E. Eiben. An algorithm for distributed on-line, on-board evolutionary robotics. In *Proc. of GECCO '11*, pages 171–178. ACM, 2011.

[14] Abraham Prieto, Josè Antonio Becerra, Francisco Bellas, and Richard J. Duro. Open-ended evolution as a means to self-organize heterogeneous multi-robot systems in real time. *Rob. and Auto. Sys.*, 58(12):1282 – 1291, 2010.

[15] David J. Schaffer, , Darrell Whitley, and Larry J. Eshelman. Combinations of genetic algorithms and neural networks: a survey of the state of the art. In *Proc. of COGANN '92*, pages 1–37, 1992.

[16] Fernando Silva, Paulo Urbano, Sancho Oliveira, and Anders Lyhne Christensen. odneat: an algorithm for distributed online, onboard evolution of robot behaviours. In *Artificial Life*, volume 13, pages 251–258. MIT Press, 2012.

[17] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evol. Comput.*, 10(2):99–127, June 2002.

[18] Richard Watson, Sevan Ficici, and Jordan Pollack. Embodied evolution: Distributing an evolutionary algorithm in a population of robots. *Rob. and Auto. Sys.*, 39:1–18, 2002.

[19] Steffen Wischmann, Kristin Stamm, and Florentin Wörgötter. Embodied evolution and learning: The neglected timing of maturation. In *Proc of Alife'07*, pages 284–293, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

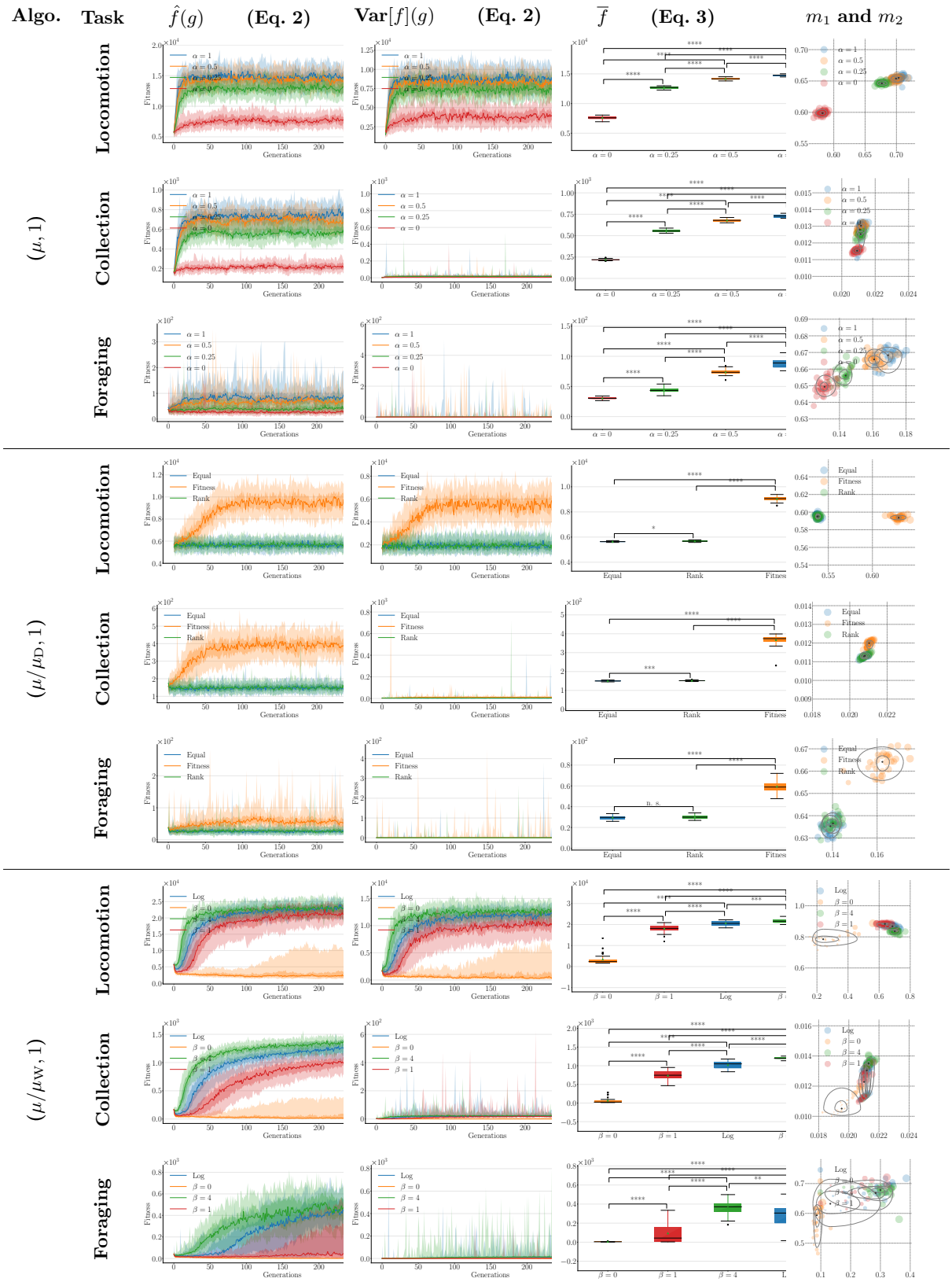

Figure 3: Results of all algorithms on all tasks. Curves represent median (solid line), the range between the 25th and the 75th percentile (darker area) and between the 5th and the 95th percentile (lighter area). On the scatter plots, m_1 lies on the abscissa and m_2 on the ordinate.

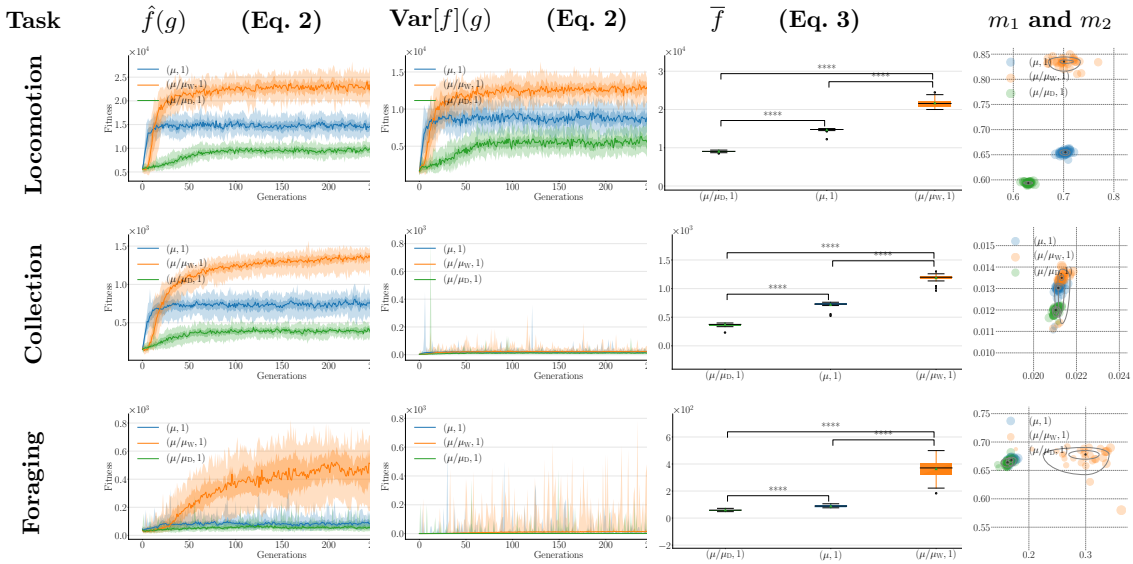

Figure 4: Comparison of $(\mu, 1)$ with $\alpha = 1$, $(\mu/\mu_{\mathrm{D}}, 1)$ with fit. prop. selection and $(\mu/\mu_{\mathrm{W}}, 1)$ with $\beta = 4$.

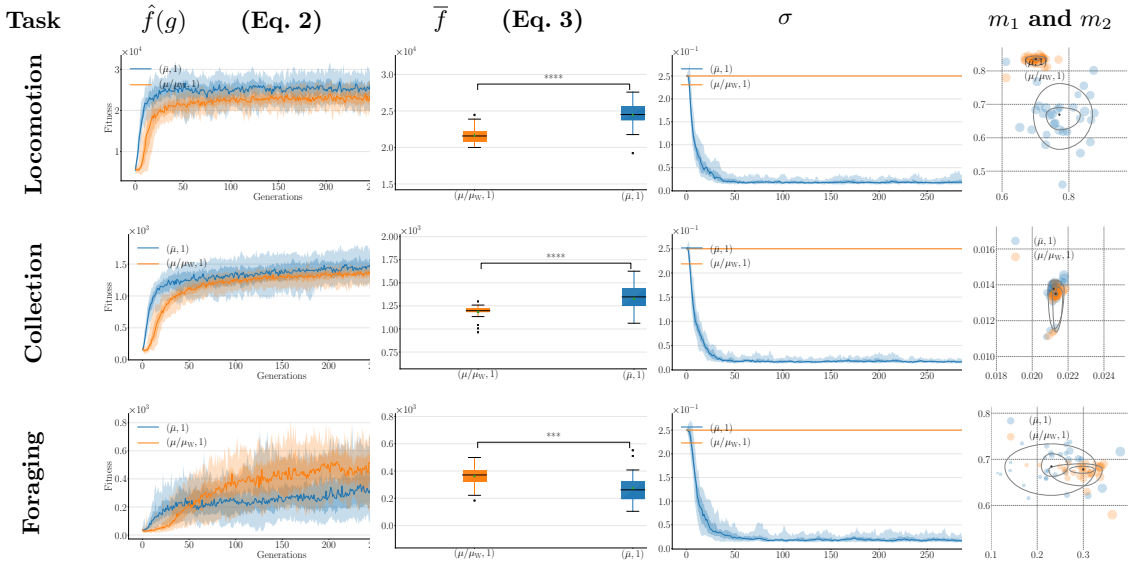

Figure 5: Comparison of $(\overline{\mu}, 1)$ with $\alpha = 1$, and $(\mu/\mu_{\mathrm{W}}, 1)$ with $\beta = 4$.