Algorithmes Évolutionnistes dans les Jeux Vidéo

Yu Guan Hsieh

June 6, 2016

Partie Algorithmique

1 Présentation des Algorithmes

1.1 Problème d'optimisation

Les algorithmes évolutionnistes font partie de la famille des algorithmes métaheuristiques, qui ont but pour résoudre des problèmes d'optimisation difficile. On se donne une fonction f, et on vise à approcher la solution qui maximalise/minimalise la valeur de f, que l'on appelle désormais *fitness*.

Trois différents algorithmes évolutionnistes sont alors considérés, l'algorithme génétique (AG), l'algorithme à évolution différentielle (ED), et l'algorithme des chauves-souris (AC), parmi eux, le travail est principalement axé sur le troisième, qui est premièrement introduit par X.-S. Yang dans [1] en 2010.

1.2 Algorithme des Chauves-souris

Mirco chauves-souris utilisent l'écholocation pour chasser les proies et se localiser, inspirant la création de cet algorithme. Une population de chauves-souris est maintenue au cours du temps, chaque individu possède une position x_i et une vitesse v_i qui se modifient, et émet des ultrasons avec une puissance $A_i \in [A_{min}, A_{max}]$ à une fréquence $f_i \in [f_{min}, f_{max}]$. Les émissions d'ultrasons s'effectuent en rafale selon le taux d'impulsions $\tau_i \in [0, 1]$.

1.2.1 Initialisation

À l'initialisation, les chauves-souris sont par défaut réparties selon une loi uniforme dans l'espace de recherche, et les vitesses sont des valeurs aléatoires respectant les bornes données.

1.2.2 Générer des nouvelles solutions

À chaque instant, des nouvelles solutions sont générées selon les éqautions (1) $\sim$ (3), où $rand(0, 1)$ est un facteur tiré uniformément de $[0, 1[$ et x_* la meilleure solution globale. Ce processus resemble à celui de l'optimisation par essaim particulaire (OEP).

$$f_i = f_{min} + rand(0, 1)(f_{max} - f_{min}) \tag{1}$$

$$v_i = v_i + f_i\,(x_i - x_*) \tag{2}$$

$$\tilde{x}_i = x_i + v_i \tag{3}$$

NB. On remarque que dans l'équation (2), le choix de $(x_* - x_i)$ au lieu de $(x_i - x_*)$ semble plus raisonnable, ce point est mieux discuté dans l'annexe D.3.

1.2.3 Recherche Locale

Une structure de recherche locale est introduite, on cherche une nouvelle solution x_{new} autour d'une solution x_{old} de fitness élevé dans la poplulation. Dans l'équation (4), $\bar{A}$ désigne la puissance moyenne des ultrasons émis par toutes les chauves-souris.

$$x_{new} = x_{old} + rand(-1, 1)\bar{A} \tag{4}$$

1.2.4 Puissance d'émission et taux d'impulsions

Quand la chauve-souris se rapproche de sa proie, elle émet plus fréquemment des ultrasons avec une puissance plus faible. Les A_i et τ_i sont donc mises à jour selon les équations (5) et (6). Les paramètres α et γ sont en général choisis proches de 0.9. Comme A_i et τ_i contrôlent respectivement la probablité d'accepter une nouvelle solution et de sauter abruptement à une autre solution, leur rôle est similaire à celui de la température dans le recuit simulé.

$$A_i = \alpha A_i \tag{5}$$

$$\tau_i = \tau_i^0(1 - e^{\gamma t}) \tag{6}$$

L'implémentation de l'algorithme des chauves-souris en Python se trouve dans l'annexe D.2 et l'annexe D.3, la pseudo-code est donné par:

Algorithm 1 Pseudo-code de l'Algorithme des chauves-souris
(n: nombre de générations, N: taille de la population, d: dimension, f: fonction de fitness)

Procédure ALGORITHME DES CHAUVES-SOURIS (n, N, d, f)
    Initialiser les positions x_i et les vitess v_i
    Initialiser $A_i \leftarrow A_0$ et $\tau_i \leftarrow 0$
    Pour $k = 1$ **à** n **faire**
        Générer des nouvelles solutions par Eqs. (1) $\sim$ (3)
        Pour $i = 1$ **à** N **faire**
            Si $rand(0, 1) > \tau_i$ **alors**
                Choisir une solution x_{old} parmi les meilleures solutions
                Générer une solution x_{new} autour de x_{old} utilisant Eq. (4)
                $\tilde{x}_i \leftarrow x_{new}$
            Si $rand(0, 1) < A_i$ **et** $f(\tilde{x}_i) > f(x_i)$ **alors**
                $x_i \leftarrow \tilde{x}_i$
                Mettre à jours A_i et τ_i selon Eqs. (5) et (6)
        Trier les x_i en fonction de $f(x_i)$

1.3 Amélioration

Pour améliorer la performance de l'algorithme des chauves-souris, en se référant aux [2] $\sim$ [8], les différentes modifications sont considérées. Seulement la version finale est présentée ici, pour le reste,

consulter l'annexe D.4 ∼ l'annexe D.6. (Les implémentations en python)

1.3.1 Mise à jour de vitesse

En s'inspirant des modifications de [5], une nouvelle mise à jour de vitesse est proposée.

$$v_{i,j} = \begin{cases} wv_{i,j} + f_i(x_{*1,j} - x_{i,j}) & \text{si} \quad rand_j(0,1) < \eta \\ wv_{i,j} + f_i(x_{*2,j} - x_{i,j}) & \text{sinon} \end{cases} \tag{7}$$

On privilige la capacité de l'exploration au début et au contraire celle de l'exploitation vers la fin, $w \in [0,1]$ est le poids de la vitesse individuelle, il diminue au cours de l'algorithme ($w_{init} > w_{final}$) :

$$w = \left(1 - \frac{iter}{iter_{max}}\right)^n (w_{init} - w_{final}) + w_{final} \tag{8}$$

D'ailleurs, pour maintenir la diversité de la population et éviter d'être piégé dans un extrêmum local, on introduit x_{*1} et x_{*2}. x_{*1} est le plus souvent la solution optimale connue pour l'instant (pobabilité de $\eta \in [0.5, 1]$), mais peut aussi être choisie suivant la loi normale, x_{*2} est toujours choisie suivant la loi normale. η contrôle également la probabilité qu'une dimension de la solution soit influencée par x_{*1} (qui est souvent meilleure) ou x_{*2}. Il accroît au cours du temps et obéit à l'équation:

$$\eta = \left(1 - \frac{iter}{iter_{max}}\right)^n (\eta_{init} - \eta_{final}) + \eta_{final} \tag{9}$$

1.3.2 Recherche locale

On implémente la recherche locale donnée dans [5] qui est à son tour inspiré par l'algorithme de l'optimisation des mauvaises herbes (*invasive weed optimization* [10]). Quelques candidats sont générés autour de x_{old} suivant une distribution normale.

$$x_{new,j} = x_{old,j} + N(0,\sigma)_j \bar{A} \tag{10}$$

Le nombre de candidat s est donné par (u le fitness) :

$$s = s_{min} + (s_{max} - s_{min}) \left(\frac{u_{old} - u_{worst}}{u_{best} - u_{worst}}\right) \tag{11}$$

L'écart type de la distribution décroît à chaque itération pour contrôler le pas de la recherche locale.

$$\sigma = \left(1 - \frac{iter}{iter_{max}}\right)^n (\sigma_{init} - \sigma_{final}) + \sigma_{final} \tag{12}$$

Enfin, x_{old} est déterminée grâce à la statistique de Maxwell-Bolzmann. L'énergie de la i^{eme} solution est $|u_{best} - u_i|$.

1.3.3 Opérateur de l'évolution différentielle

Suivant les conseils de [2], [6] et [7], le processus de l'évolution différentiel est introduit au sein de l'algorithme des chauves-souris. Quand $rand(0,1) < \tau_i$, on effectue un croisement différentiel selon `DE/current to best/1/bin` (une méthode de l'ED).

$$x_{new,j} = \begin{cases} x_{a,j} + F_1(x_{b,j} - x_{c,j}) + F_2(x_* - x_{a,j}) & \text{si} \quad rand_j(0,1) < Cr \ \text{ ou } \ j = j_r \\ x_{i,j} & \text{sinon} \end{cases} \tag{13}$$

x_a, x_b et x_c sont des individus distincts choisis aléatoirement dans la population, x_* est la meilleure solution connue pour l'instant, j_r est uniformément tiré dans $[\![1, d]\!]$ où d est la dimension, $Cr \in [0, 1]$ est le coefficient de croisement, et $F_1, F_2 \in [0, 2]$ sont appelés *differential weight*.

1.3.4 Mutation et d'autres détails

Toujours pour garantir la diversité des solutions, une mutation simple comme indiquée dans l'équation (14) (ρ le taux de mutation) est intégrée dans l'algorithme. Dans cette même optique, les conditions de mise à jour et d'acceptation des nouvelles solutions sont aussi légèrement modifiées.

$$x_{mutation,i,j} = \begin{cases} x_{i,j} + N_j(0, 0.5) & \text{si} \quad rand_j < \rho \\ x_{i,j} & \text{sinon} \end{cases} \tag{14}$$

Algorithm 2 Pseudo-code de l'Algorithme des chauves-souris Amélioré

Procédure ALGORITHME DES CHAUVES-SOURIS AMÉLIORÉ (n, N, d, f)
  Initialiser les positions x_i et les vitess v_i
  Initialiser $A_i \leftarrow A_0$ et $\tau_i \leftarrow 0$
  $T_0 \leftarrow u^0_{best}$
  Pour $k = 1$ **à** n **faire**
    Mettre à jours w, η, σ selon Eqs. (8), (9) et (12)
    $T \leftarrow \left(\frac{iter + 1}{iter_{max}}\right)^n T_0$
    Générer des nouvelles solutions selon Eqs. (1), (7) et (3)
    Pour $i = 1$ **à** N **faire**
      Si $rand(0, 1) < \max(A_i, 0.1)$ **alors**
        Si $rand(0, 1) > \tau_i$ **alors**
          Choisir la solution x_{old} utilisant la statistique de Maxwell-Boltzmann
          Calculer s selon Eq. (11)
          Générer s solutions autour de x_{old} selon Eq. (10)
          Choisir la x_{new} qui maximalise le fitness
        Sinon
          Générer x_{new} selon Eq. (13)
        Générer $x_{mutation,i}$ selon Eq. (14)
        Déterminer $\hat{x}_i$ parmi $\tilde{x}_i, x_{new}$ et $x_{muation,i}$ qui maximalise le fitness
        Si $f(\hat{x}_i) > f(x_i)$ **or** $rand(0, 1) < \eta/2$ **alors**
          $x_i \leftarrow \hat{x}_i$
          Mettre à jours A_i et τ_i selon Eqs. (5) et (6)
    Trier les x_i en fonction de $f(x_i)$
    Mettre à jours la meilleure solution connue pour l'instant x_*

2 Premier Résultat

Les fonctions du Benchmark sont les problèmes conçus en particulier pour tester les performance des algorithmes d'optimisation. Différentes fonctions possèdent des caractèristiques assez variées et correspondent souvent aux problèmes d'optimisation dans le monde réel.

Les fonctions utilisées dans mon travail sont présentées dans le **Tableau 1**. Les algorithmes sont entrainés afin de trouver le minimum global. Pour traiter le problème d'optimisation sous contraintes, il suffit de modifier la fonction de fitness telle qu'elle prend la valeur $-\infty$ quand les contraintes ne sont pas satisfaites.

2.1 Comparaisons des Différentes Modifications

BA, BA2, BA3, BA4, BA5 désignent réspectivement les algorithmes `BatAlgorithm2` (version originelle), `BA_v2`, `BA_v3`, `BA_DE` (l'opérateur ED) et `BA_DE_T_v2` (version finale) codés dans l'annexe D.3

TABLEAU 1: Fonctions du Benchmark utilisées
(C: caractéristique, U: unimodal, M: multimodal, E: espace de recherche, - : non fixé)

No.	Nom	Dim.	C	E	Définition de Fonction	f_{min}
1	Sphère	-	U	$[-10^3, 10^3]$	$f(x) = \sum_{i=1}^{d} x_i^2$	0
2	Zakharov	-	U	$[-10, 10]$	$f(x) = \sum_{i=1}^{d} x_i^2 + \left(\sum_{i=1}^{d} 0.5 i x_i^2\right)^2 + \left(\sum_{i=1}^{d} 0.5 i x_i^2\right)^4$	0
3	Rosenbrock	-	U	$[-15, 15]$	$f(x) = \sum_{i=1}^{d-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$	0
4	Ackley	-	M	$[-32, 32]$	$f(x) = -20 \exp\left(-0.2\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d} \cos(2\pi x_i)\right) + 20 + e$	0
5	Griewank	-	M	$[-600, 600]$	$f(x) = \frac{1}{4000}\sum_{i=1}^{d} x_i^2 - \prod_{i=1}^{d} \cos\left(\frac{x_i}{\sqrt{i}} + 1\right)$	0
6	Rastrigin	-	M	$[-5.12, 5.12]$	$f(x) = 10d + \sum_{i=1}^{d} [x_i^2 - 10\cos(2\pi x_i)]$	0
7	Schwefel	-	M	$[-500, 500]$	$f(x) = 418.9829d - \sum_{i=1}^{d} x_i \sin(\sqrt{\lvert x_i \rvert})$	≈ 0
8	Michalewicz	2 5 10	M	$[0, \pi]$	$f(x) = -\sum_{i=1}^{d} \sin(x_i) \sin^{20}\left(\frac{i x_i^2}{\pi}\right)$	-1.8013 -4.6876 -9.6602
9	Easom	2	M	$[-100, 100]$	$f(x) = (-1)^{d+1}\left(\prod_{i=1}^{d} \cos(x_i)\right) \exp\left[\sum_{i=1}^{d} (x_i - \pi)^2\right] + 1$	0
10	Dropwave	2	M	$[-5.12, 5.12]$	$f(x) = -\frac{1 + \cos\left(12\sqrt{\sum_{i=1}^{d} x_i^2}\right)}{(\sum_{i=1}^{d} 0.5 x_i^2) + 2} + 1$	0

$\sim$ l'annexe D.7. Ils sont testés par différentes fonctions du Benchmark. La taille de la population est fixé à $N = 50$, et les autres paramètres sont listés dans le **Tableau 2**. Les convergences graphiques se trouvent dans la **Figure 2**.

On observe pour presque toutes les fonctions (sauf Zakrarov) la supériorité de BA5 en termes de vitesse de convergence et qualité de solution vis-à-vis des autres améliorations. En particulier, la performance est énormément améliorée par rapport à l'algorithme des chauves-souris initial.

TABLEAU 2: Paramètres des différents algorithmes de chauves-souris ((*) : $0.6 \sim 1$ pour BA3)

Id	τ_0	A_0	γ	α	s	w	Cr	n	ρ	σ	η
	0.85	0.95	0.85	0.95	$0 \sim 5$	$0.9 \sim 0.2$	0.9	2	0.2	$1 \sim 0$	$0.5 \sim 0.8^{(*)}$

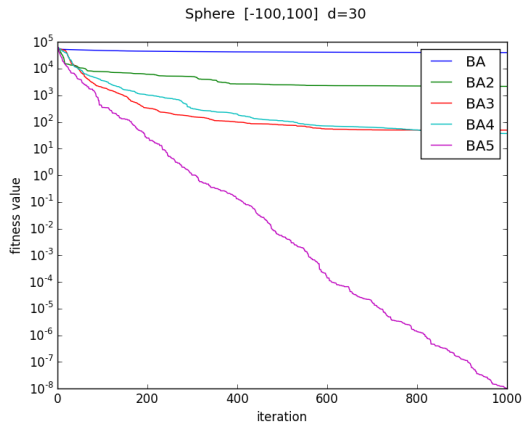
2.2 AC, AG, et ED

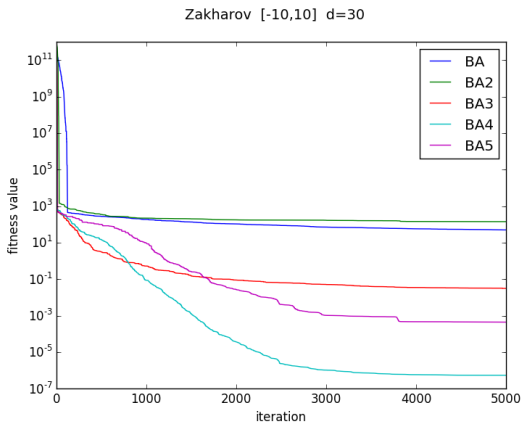
La version finale de l'algorithme des chauves-souris est aussi comparée avec l'algorithme génétique (AG) et l'algorithme à évolution différentielle (ED), on fixe $N = 50$. Pour l'AG, le croisement en un point est utilisé et l'élitisme est considéré, la probabilité de mutation et la proportion des élites sont prises égales à 0.1 et 0.05. On emploie directement la bibliothèque `pybrain` de python. Pour l'algorithme ED, on adopte la stratégie `DE/rand/1/bin` avec $Cr = 0.9$ et F généré uniformément de $[0.5, 1]$. Les paramètres pour l'ACA (algorithme des chauves-souris amélioré) sont comme avant.

Les algorithmes sont testés 30 fois sur chaque fonction donnée dans le **Tableau 1**, avec l'espace de recherche indiqué (chaque test comporte 1000 itérations) et les valeurs (min, max, moyen, écart-type, temps d'exécution) sont affichées dan le **Tableau 3**. Quelques convergences graphiques sont montrés dans la **Figure 3**.

Pour les fonctions 1, 4, 5, 9, 10, l'ACA obtient la meilleure solution dans tous les cas, et pour les fonction 2, 3 et 8, il possède aussi la mielleure performance pour des petites dimensions. D'ailleurs, la **Figure 3** suggère que pour les fonctions 2 ($d = 30$), 6 ($d = 10$) et 8 ($d = 10$), l'ACA est capable de trouver la meilleure solution parmi les trois à une longue échelle. Les temps d'exécution des trois algorithmes sont assez proches, ce qui est rassurant: la comparaison a un sens.

En même temps, l'ACA a souvent la meilleure vitesse de convergence. On remarque que malgré la vitesse de convergence faible de l'ED, la solution s'améliore toujours. D'aprés les expériences, si le nombre de générations n'est pas limité, l'ED a le plus de chance de trouver la solution optimale globale.





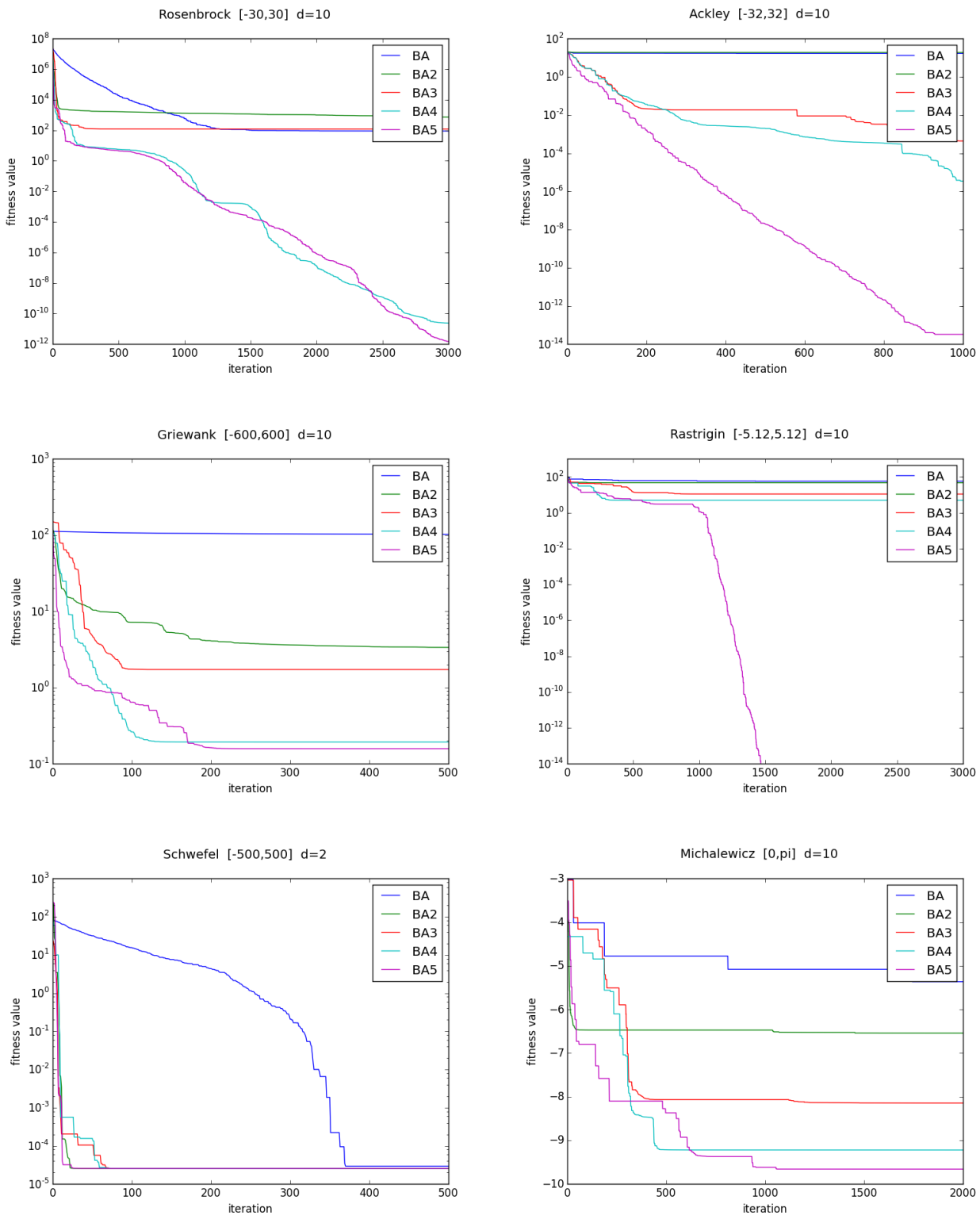


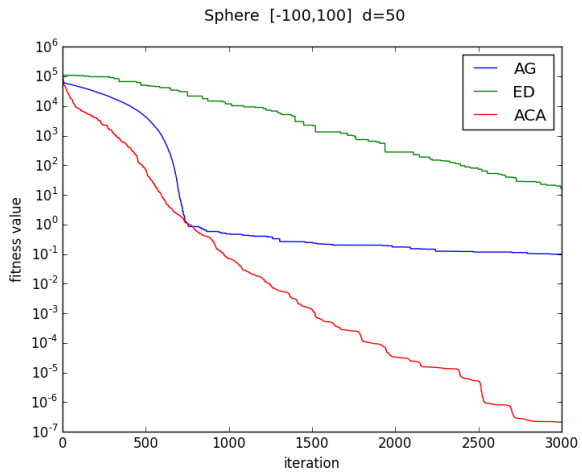
FIGURE 2: Convergence graphique des différentes modifications de l'algorithme des chauves-souris

Tableau 3: Résultats comaprés des AG, ED et ACA sur les fonctions numériques

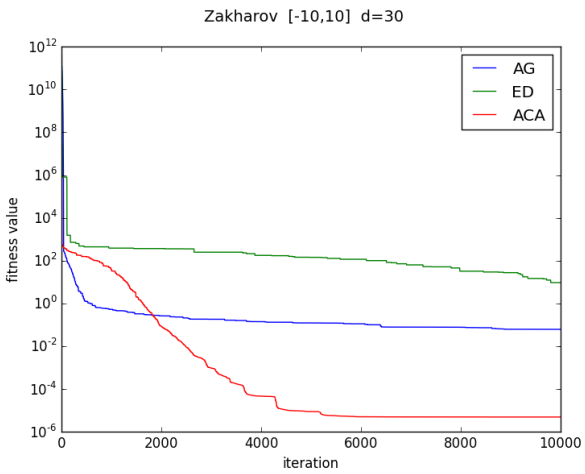
No.	Fonction	Dim.	Méthode	Min	Max	Moyen	Écart-type	Temps(s)
1	Sphère	10	AG	5.51E+02	2.34E+05	4.86E+04	4.97E+04	55.73
			ED	5.63E-13	1.46E-11	5.10E-12	3.99E-12	86.03
			ACA	**4.81E-32**	**3.51E-22**	**1.18E-23**	6.31E-23	79.89
		30	AG	6.67E+05	3.28E+06	2.04E+06	6.12E+05	95.80
			ED	6.29E+03	5.77E+04	2.28E+04	1.11E+04	153.61
			ACA	**8.77E-08**	**1.22E-02**	**2.22E-03**	2.87E-03	132.77
		50	AG	3.79E+06	7.56E+06	5.25E+06	8.66E+05	134.99
			ED	3.80E+05	2.01E+06	8.30E+05	3.56E+05	201.56
			ACA	**1.68E+00**	**2.20E+02**	**4.30E+01**	5.55E+01	187.04
2	Zakharov	10	AG	6.61E-04	1.17E-02	3.52E-03	2.48E-03	81.88
			ED	1.26E-07	2.10E-05	3.12E-06	4.50E-06	108.86
			ACA	**1.45E-18**	**1.03E-12**	**3.82E-14**	1.85E-13	94.43
		30	AG	**2.34E-01**	**8.87E-01**	**4.57E-01**	1.49E-01	137.48
			ED	2.68E+02	4.94E+02	3.83E+02	5.68E+01	135.91
			ACA	4.01E-01	6.39E+00	2.24E+00	1.28E+00	154.94
		50	AG	**2.93E+00**	**8.87E+00**	**6.48E+00**	1.20E+00	175.60
			ED	6.24E+02	4.63E+03	1.19E+03	8.30E+02	168.40
			ACA	9.33E+01	3.18E+02	1.87E+02	5.02E+01	214.83
3	Rosenbrock	10	AG	4.66E-03	1.20E+01	4.58E+00	3.25E+00	83.32
			ED	5.76E-04	1.13E-02	4.38E-03	2.96E-03	115.57
			ACA	**3.09E-04**	**4.01E+00**	**4.32E-01**	1.19E+00	99.00
		30	AG	**3.76E+00**	**1.85E+02**	**1.80E+01**	3.13E+01	174.12
			ED	6.55E+02	7.36E+03	2.81E+03	1.73E+03	218.99
			ACA	5.20E+00	1.95E+02	4.34E+01	3.95E+01	171.55
		50	AG	1.06E+02	3.39E+02	**1.75E+02**	4.81E+01	302.53
			ED	1.27E+05	1.07E+06	4.16E+05	2.10E+05	252.24
			ACA	**8.54E+01**	**3.21E+02**	1.91E+02	5.96E+01	240.02
4	Ackley	10	AG	1.02E-03	4.03E-03	2.28E-03	6.52E-04	73.82
			ED	5.14E-08	3.43E-07	1.32E-07	6.49E-08	104.25
			ACA	**7.55E-15**	**2.69E-12**	**1.14E-13**	4.80E-13	91.67
		30	AG	9.75E-02	1.87E+01	5.02E+00	6.88E+00	125.76
			ED	3.86E+00	6.63E+00	4.76E+00	6.69E-01	185.57
			ACA	**7.31E-05**	**2.41E+00**	**1.73E-01**	5.33E-01	147.05
		50	AG	1.76E+01	1.92E+01	1.84E+01	3.78E-01	177.18
			ED	1.07E+01	1.88E+01	1.42E+01	2.19E+00	204.01
			ACA	**2.99E+00**	**1.91E+01**	**7.50E+00**	4.31E+00	199.13
5	Griewank	10	AG	6.84E+00	6.26E+01	2.79E+01	1.26E+01	76.05
			ED	2.87E-01	**6.10E-01**	4.52E-01	7.53E-02	116.62
			ACA	**6.89E-02**	8.88E-01	**2.50E-01**	1.80E-01	86.41
		30	AG	5.47E+01	2.09E+02	1.20E+02	3.92E+01	128.43
			ED	1.57E+00	4.43E+00	2.88E+00	7.08E-01	182.36
			ACA	**1.04E-07**	**7.33E-02**	**2.37E-02**	2.11E-02	145.26
		50	AG	2.57E+02	5.06E+02	3.80E+02	7.10E+01	193.05
			ED	4.16E+01	1.31E+02	6.89E+01	2.38E+01	224.89
			ACA	**3.53E-02**	**2.30E-01**	**1.12E-01**	6.03E-02	197.83

Tableau 3: (continu)

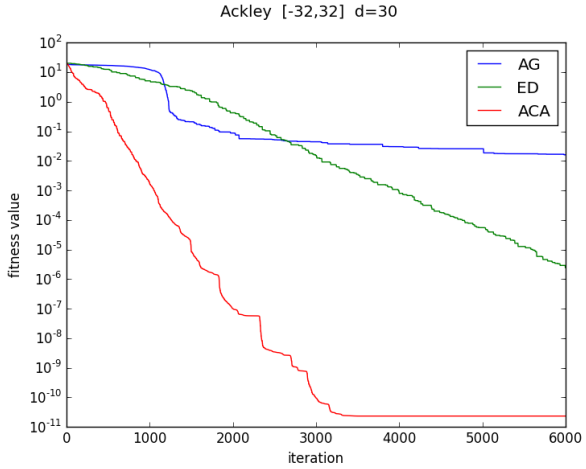
No.	Fonction	Dim.	Méthode	Min	Max	Moyen	Écart-type	Temps(s)
6	Rastrigin	10	AG	1.20E-04	**1.55E-03**	**5.72E-04**	4.23E-04	67.48
			ED	1.28E+01	3.83E+01	2.78E+01	5.29E+00	90.86
			ACA	**1.42E-14**	2.98E+00	9.07E-01	9.42E-01	91.16
		30	AG	**2.55E+00**	**1.27E+01**	**7.39E+00**	2.25E+00	118.20
			ED	2.23E+02	2.74E+02	2.45E+02	1.20E+01	131.66
			ACA	2.89E+01	1.06E+02	6.40E+01	1.85E+01	148.43
		50	AG	**7.31E+01**	**1.64E+02**	**1.18E+02**	2.17E+01	169.42
			ED	4.58E+02	6.01E+02	5.12E+02	3.62E+01	169.30
			ACA	1.19E+02	2.35E+02	1.70E+02	2.83E+01	203.18
7	Schwefel	2	AG	**2.55E-05**	2.17E+02	3.62E+01	8.09E+01	49.18
			ED	**2.55E-05**	**2.55E-05**	**2.55E-05**	0.00E+00	60.04
			ACA	**2.55E-05**	**2.55E-05**	**2.55E-05**	0.00E+00	67.39
		5	AG	**6.36E-05**	4.34E+02	1.30E+02	1.39E+02	51.82
			ED	**6.36E-05**	**6.36E-05**	**6.36E-05**	0.00E+00	69.65
			ACA	**6.36E-05**	5.72E+02	1.54E+02	1.56E+02	70.05
		10	AG	**1.27E-04**	**8.49E+02**	**4.16E+02**	2.75E+02	59.96
			ED	3.68E-03	1.73E+03	6.04E+02	4.94E+02	69.17
			ACA	2.17E+02	1.86E+03	8.17E+02	3.50E+02	88.31
8	Michalewicz	2	AG	**-1.80E+00**	**-1.80E+00**	**-1.80E+00**	3.00E-06	52.10
			ED	**-1.80E+00**	**-1.80E+00**	**-1.80E+00**	4.44E-16	61.68
			ACA	**-1.80E+00**	**-1.80E+00**	**-1.80E+00**	4.44E-16	62.54
		5	AG	**-4.69E+00**	**-4.69E+00**	**-4.69E+00**	7.70E-05	63.42
			ED	**-4.69E+00**	-4.65E+00	**-4.69E+00**	7.50E-03	80.17
			ACA	**-4.69E+00**	**-4.69E+00**	**-4.69E+00**	1.65E-15	75.97
		10	AG	**-9.66E+00**	**-9.66E+00**	**-9.66E+00**	7.97E-04	84.67
			ED	-8.09E+00	-6.33E+00	-7.24E+00	4.15E-01	88.97
			ACA	**-9.66E+00**	-9.17E+00	-9.51E+00	1.34E-01	95.85
9	Easom	2	AG	1.89E-08	1.00E+00	9.00E-01	3.00E-01	49.32
			ED	**0.00E+00**	**0.00E+00**	**0.00E+00**	0.00E+00	60.97
			ACA	**0.00E+00**	**0.00E+00**	**0.00E+00**	0.00E+00	61.24
10	Dropwave	2	AG	6.16E-09	6.38E-02	2.77E-02	3.16E-02	44.37
			ED	**0.00E+00**	**0.00E+00**	**0.00E+00**	0.00E+00	61.12
			ACA	**0.00E+00**	**0.00E+00**	**0.00E+00**	0.00E+00	60.18



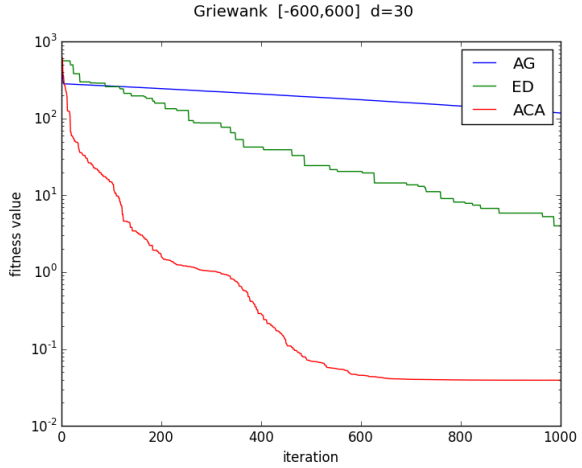
(a)



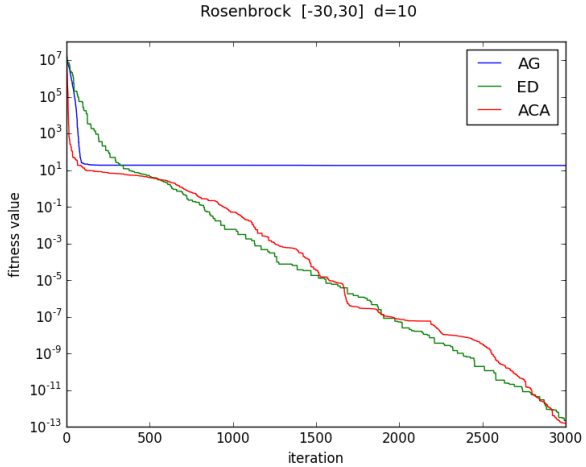
(b)



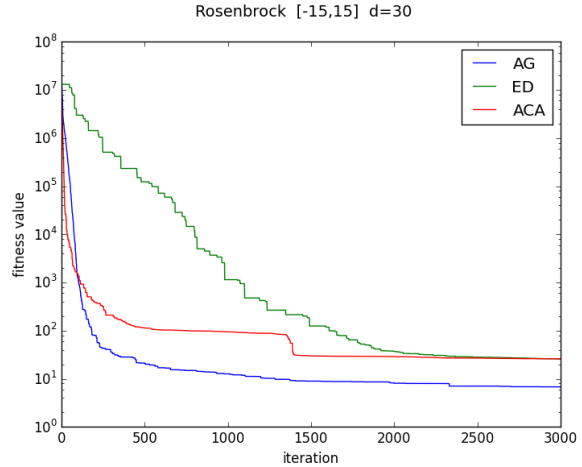
(c)



(d)

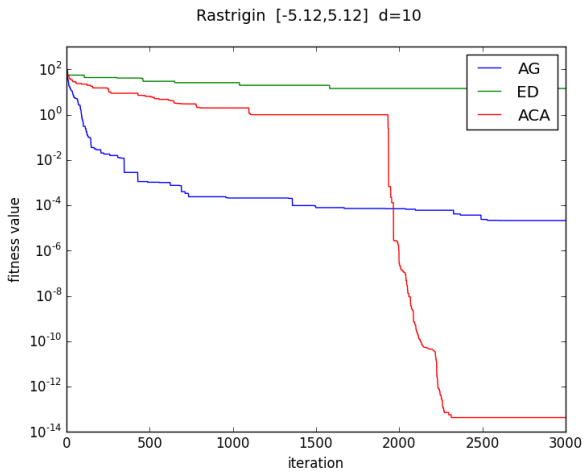


(e)

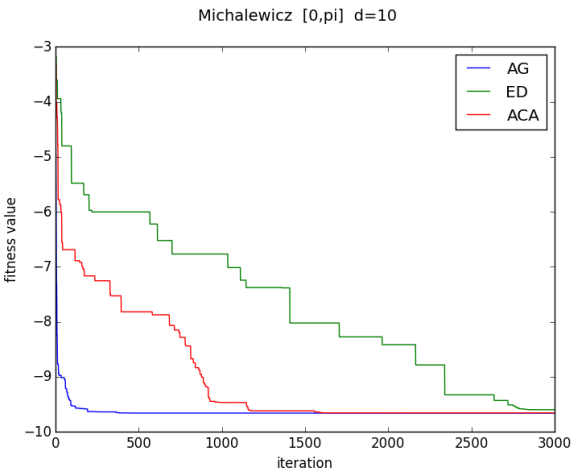


(f)

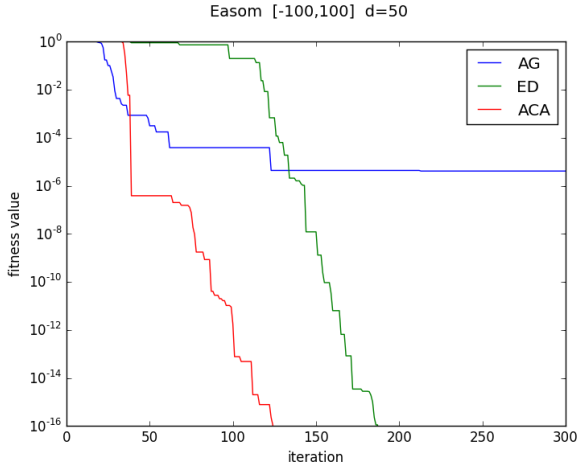
FIGURE 3: Convergence graphique des AG, ED, et ACA sur des différentes fonctions



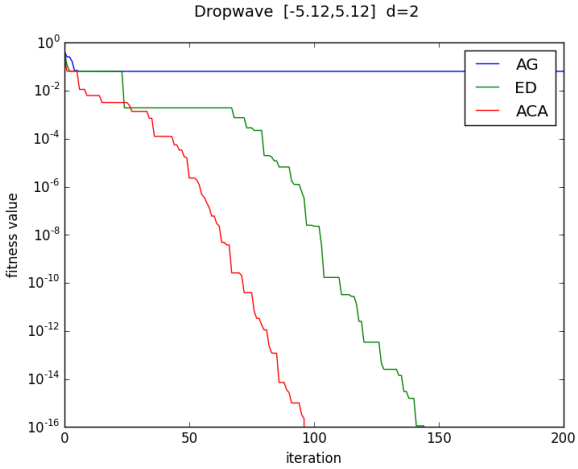
(g)



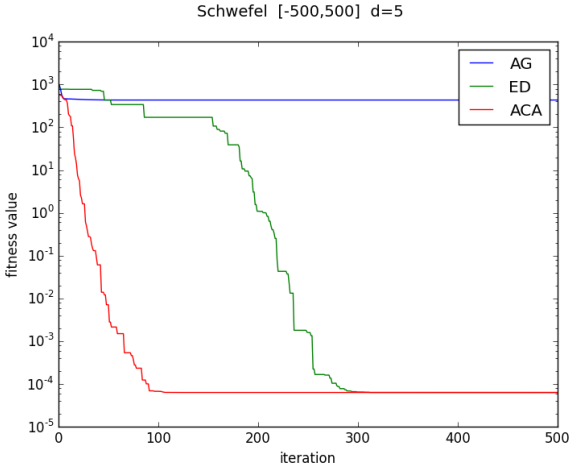
(h)



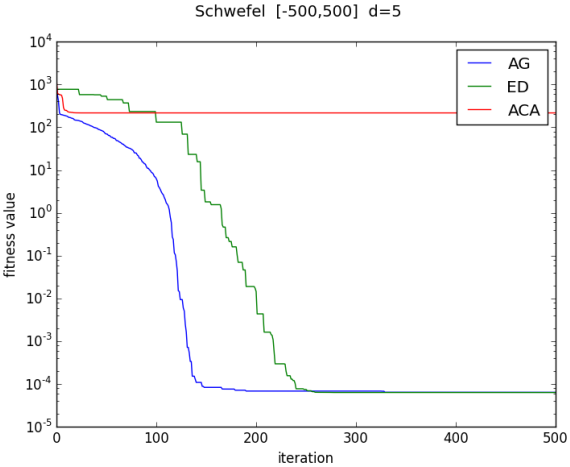
(i)



(j)



(k)



(l)

FIGURE 3: (Continu) Pour la fonction Schwefel ($d = 5$), deux cas se présentent

Autour des IA dans les Jeux Vidéo

1 Pong

Pong est un des premiers jeux vidéo d'arcade et le premier jeu vidéo d'arcade de sport. Chaque joueur s'affronte en déplaçant la raquette de haut en bas, de façon à garder la balle dans le terrain de jeu. (de *Wikipédia*)

1.1 Expérience

1.1.1 Implémentation de Bot

Le joueur artificiel, ou Bot, est construit grâce à un réseau de neurones non bouclé avec un biais. Le réseau posséde cinq entrées: la position de la balle (p_{bx}, p_{by}), la vitesse de la balle (v_{bx}, v_{by}) et la position de la raquette p_{ry}, une couche cachée, et une sortie s pour contrôler le mouvement de la raquette (vers le haut si $s > 0.5$, vers le bas si $s < -0.5$, ne fait rien si $s \in [-0.5, 0.5]$). On utilise la bibliothèque `pybrain` pour la réalisation du réseau de neurones.

1.1.2 Fonction de Fitness

Pour trouver les paramètres optimisés du réseau, le processus d'optimisation est lancé. Le jeu est joué et le temps total d'une partie (une partie termine quand un joueur pert 5 fois) sert de fonction de fitness. Et pour simplifier, seulement le mouvement de la raquette à gauche est considéré, la balle rebondit toujours quand elle rencontre la borne de droite. L'angle du rebond ne depend pas ici de l'endroit où la balle tape la raquette.

Le nombre des neurones cachés est fixé à 7 dans les expériences. Pour ne pas être influencé par des solutions mal évaluées (qui reçoivent par accident un fitness trop élevé), les fitness sont réévalués à chaque itération. On prend $N = 20$. Les autres paramètres sont inchangés.

1.2 Résultat

Quand le joueur est autorisé à déplacer la raquette pour chaque mouvement de balle, après suffisament des générations (ACA: 5, ED: 20, AG: 40), les trois algorithmes sont capables de développer une stratégie pour survivre. L'ACA apprend au bot à tout simplement suivre p_{by}, alors que les joueurs entrainés par l'ED et l'AG se cachent dans les coins dans la plupart du temps et bougent seulement quand la balle se rapproche. La **Figure 4** montre que le bot apprend le plus vite avec l'ACA.

Si l'action du joueur est effectuée seulement tous les deux mouvements de la balle, il ne peut plus suivre p_{by}, le problème devient donc plus délicat. Avec l'entrainement de l'ACA, le joueur artificiel est capable de prédire la trajectoire de la balle.

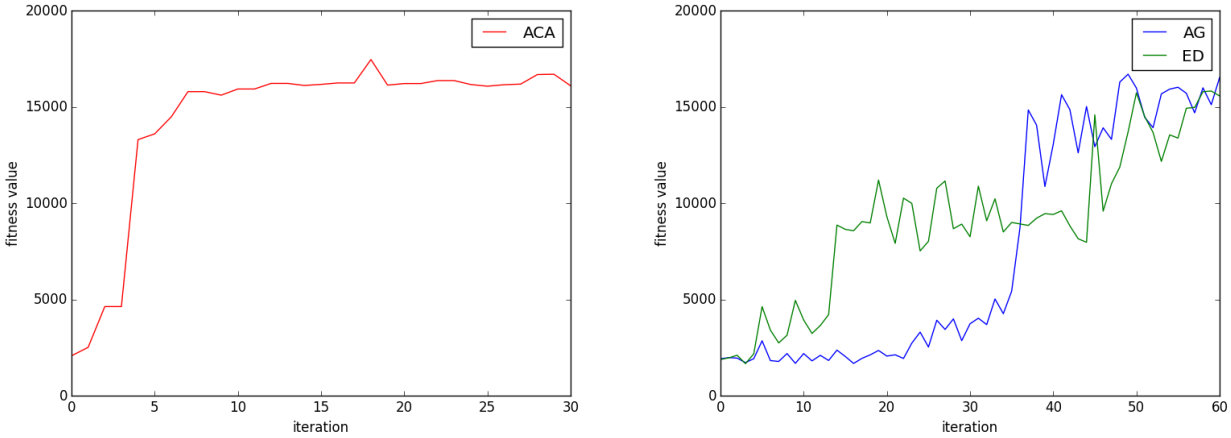


FIGURE 4: Les évolutions du fitness du bot entrainé par différentes méthodes, réaction du joueur pour chaque mouvement de la balle (≈15000 est déjà le fitness maximal)

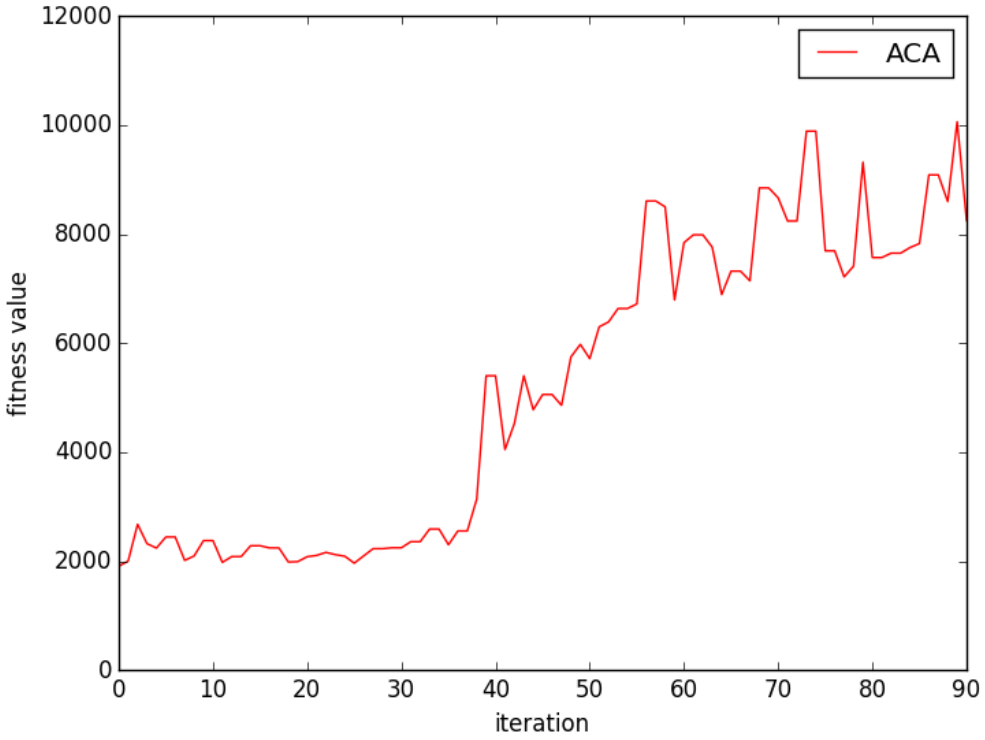


FIGURE 5: L'évolution du fitness du bot entrainé par l'ACA, réaction du joueur tous les deux mouvements de la balle **NB.** La variance des différents résultats est assez important

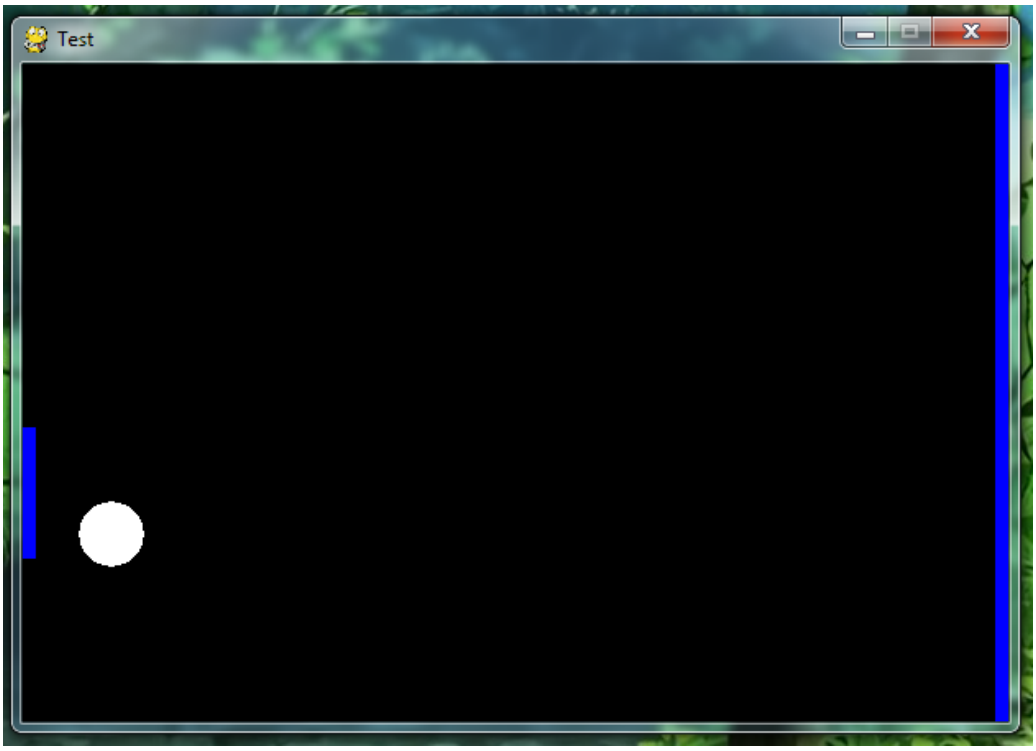


FIGURE 6: Le jeu de pong d'une seule raquette utilisé pour l'entrainenment

2 Tetris

Tetris est un célèbre jeu vidéo conçu par Alekseï Pajitnov en 1984. Le jeu se déroule sur une grille 2D où des pièces de différentes formes, les tétriminos, tombent depuis le haut jusqu'au bas. Le joueur doit déplacer chaque tétrimino pour réaliser des lignes complètes. Les lignes pleins alors disparaissent et le joueur peut de nouveau remplir les cases libérées. (de *Wikipédia*)

2.1 Expérience

2.1.1 Implémentation de Bot

Pour un état du jeu donné (configuration actuelle du mur et la pièce courante), le travail du joueur artificiel consiste à parcourir toutes les actions possibles (l'orientation et la position de la piéce), et en choisir celle qui est censée être la meilleure. Ainsi, une *fonction d'évaluation* F de l'action est nécessaire. Une évaluation est une combinaison des *fonctions de base* f_i. C'est-à-dire:

$$F = \sum_i p_i f_i$$

où les p_i sont les poids de chaque *fonction de base*. Les *fonctions de base* utilisées sont montrées dans le **Tableau 4**, en référence à [11], [12] et [14]. Le bot qui n'utilise que les six premières fonctions est dans la suite nommée B1, et ce qui les utilise toutes est nommé B2. On considère toujours un contrôleur à une pièce.

TABLEAU 4: *Fonctions de base* de Tetris

Fonction de base	**Id**	**Description**
Hauteur maximale	H	Hauteur maximale des colonnes du mur
Hauteur totale	Λ	Somme de hauteurs des colonnes
Lignes supprimées	C	Nombre de lignes complétées au dernier coup
Trous	T	Nombre de cases vides recouvertes par au moins une case plein
Différence intercolonne	D	Somme de différences de hauteurs entre deux colonnes adjacentes
Hauteur d'arrivée	ℓ	Hauteur de la position où la dernière pièce a été posée
Transitions de lignes	T_l	Nombre de transitions plein/vide ou vide/plein horizontales
Transitions de colonnes	T_c	Nombre de transitions plein/vide ou vide/plein verticales
Lignes avec trous	L_t	Nombre de lignes contenant au moins un trou
Profondeurs des trous	P	Nombre de cases pleins au-dessus d'au moins un trou

2.1.2 Fonction de Fitness

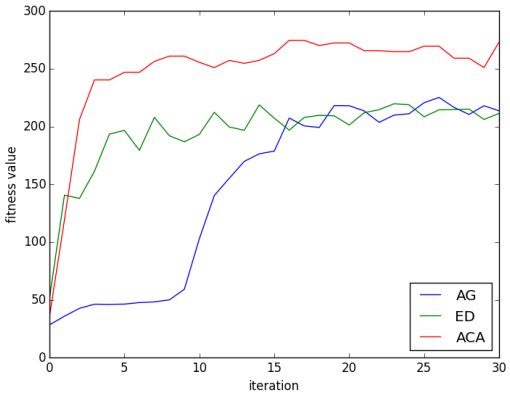
Notre but est de trouver la distribution optimale de poids des différentes f_i. Naturellement, le temps de déroulement de chaque partie peut être considéré comme le fitness d'une distribution, mais le caractère fortement aléatoire du jeu Tetris conduit à une variance importante des résultats, il vaut mieux prendre la moyenne sur plusieurs parties. Pour réduire le temps d'exécution, on peut soit (i) effectuer le jeu sur une grille plus petite (10×10 par exemple), soit (ii) lancer une partie qui ne comporte que les tétriminos Z et S. Malheureusement, comme montré par l'expérience et suggéré dans [11], [14], un joueur performant pour (i) ou (ii) n'est pas forcément le meilleur joueur pour un jeu normal. Les configurations des trois algorithmes sont comme dans la section précédente.

2.2 Résultat

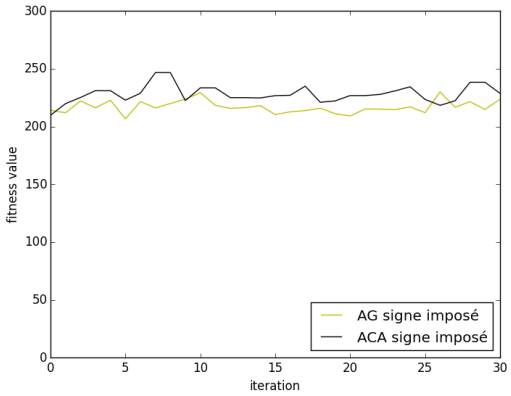
Les évolutions des bots entrainées sous conditions (i) et (ii) sont montrées dans la Figure 7. La comparaison de la Figure 7a et c exhibe une supériorité de B2 vis-à-vis de B1, ce qui nous confirme l'importance du choix des f_i. Curieusement, plusieurs expériences montrent qu'au lieu de favoriser la fonction C et pénaliser les autres, le bot posséde souvent aussi le poids positif pour les autres fonctions (la plusplart du temps H). La Figure 7b et d sont des manifestations du fait que le résultat ne devient pas forcément meilleur quand on impose les signes des p_i.

Si on enlève la condition (ii), après une trentaine de générations, le joueur arrive sans difficulté à réaliser quelques milliers de lignes sur une grille standard (10×22). (voir Figure 8 et Figure 9). Or au delà, c'est difficile d'avoir un meilleur résultat. Le caractère aléatoire du jeu et le temps important pour jouer chaque partie rend Tetris vraiment dur à apprendre - surtout sur un ordinateur normal. Une des meilleurs distributions de p_i obtenue (pour B2) est (dans l'ordre de Tableau 4):

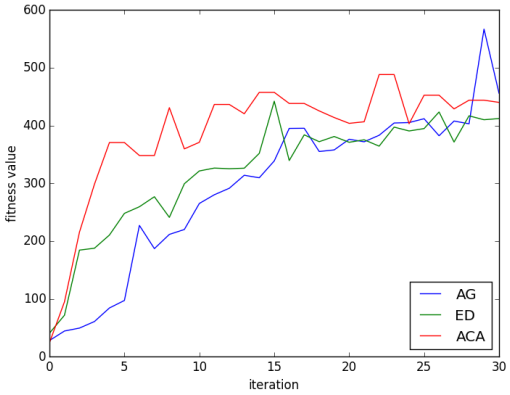
$$[3.75939374, 0.35630412, 15.31408412, -4.98850843, -1.502686,$$
$$-1.72665223, -1.74285002, -0.11057638, -4.60854415, -1.16647816]$$



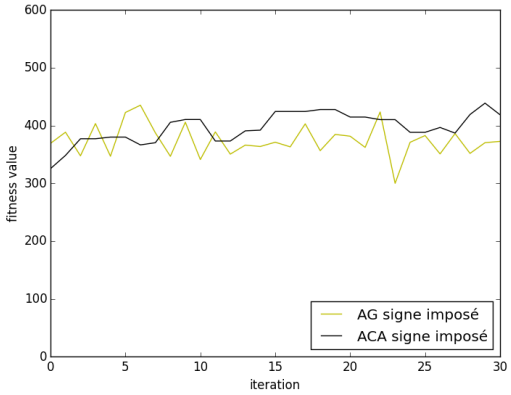
(a) B1, cas normal



(b) B1, signe imposé



(c) B2, cas normal



(d) B2, signe imposé

FIGURE 7: Les évolutions des bots entrainés sous conditions (i) et (ii)

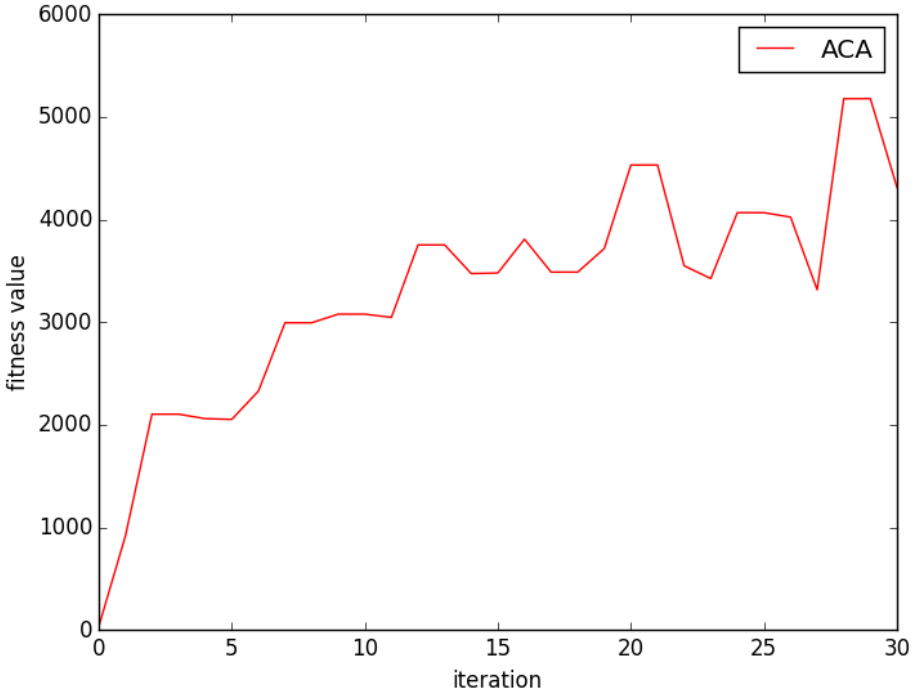


FIGURE 8: L'évolution de B2 entrainé par l'ACA sous condition (i)

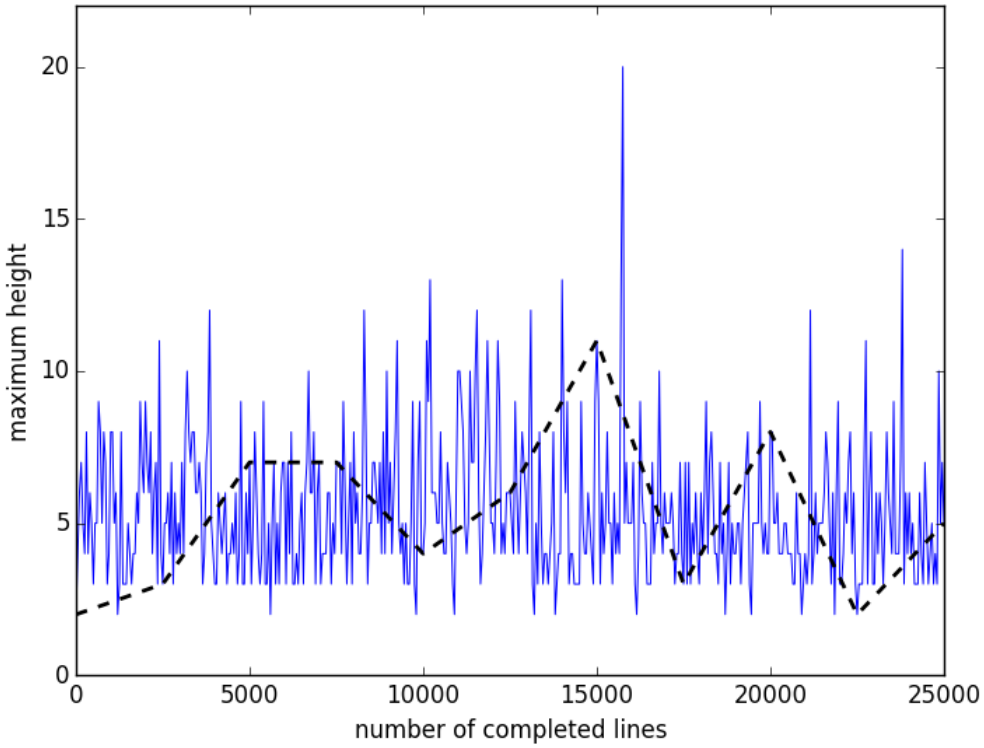


FIGURE 9: L'évolution de H au cours d'une partie de Tetris jouée sur une grille standard par un des bots entrainés (Les points sont marqués toutes les 50 lignes complétées)

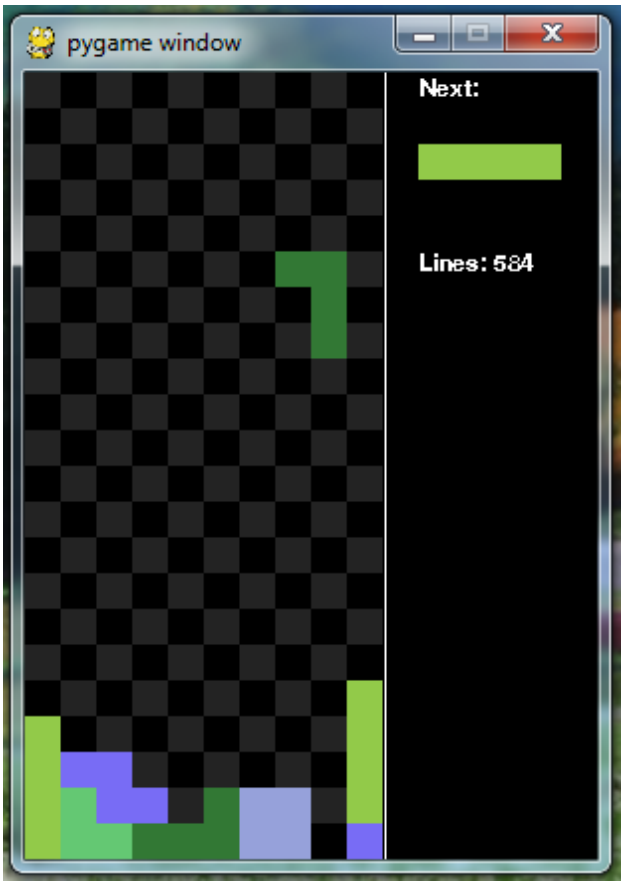


FIGURE 10: Une partie de Tetris jouée par un des bots entrainés

Annexe A. Bibliographie

[1] X.-S. Yang. A new metaheuristic Bat-inspired Algorithm, *Nature Inspired Cooperative Strategies for Optimization (NISCO 2010), Springer Press*. Vol. 284, 2010, p. 65-74.

[2] I. Jr. Fister, et al. A hybrid bat algorithm, *Elekrotehniški Vestnik*. Vol. 80, 2013, p. 1-7.

[3] S. Yılmaz, E.U. Kucuksille, Y. Cengiz. Modified bat algorithm, *Elektronika ir Elektrotechnika*. Vol. 20, No. 2, 2014, p. 71-78.

[4] Wasi Ul Kabir et al. A Novel Adaptive Bat Algorithm to Control Explorations and Exploitations for Continuous Optimization Problems, *International Journal of Computer Applications*. Vol. 94, No. 13, 2014, p. 15-20.

[5] Selim Yilmaz et Ecir U. Kucuksille. A new modification approach on bat algorithm for solving optimization problems, *Applied Soft Computing*. Vol. 28, 2015, p. 259-275.

[6] Adis Alihodzic et Milan Tuba. Improved Hybridized Bat Algorithm for Global Numerical Optimization. [document électronique]. 2014 UKSim-AMSS 16th International Conference on Computer Modelling and Simulation. http://ijssst.info/Vol-15/No-4/data/4923a057.pdf

[7] Jian Xie, Yongquan Zhou, Huan Chen. A novel bat algorithm based on differential operator and Lévy flights trajectory, *Computational Intelligence and Neuroscience*. [en ligne]. Vol. 2013, 2013. http://dx.doi.org/10.1155/2013/453812

[8] Zhao Guodong, Zhou Ying and Song Liya. Simulated Annealing Optimization Bat Algorithm in Service Migration Joining the Gauss Perturbation, *International Journal of Hybrid Information Technology*. [en ligne]. Vol. 8, No. 12, 2015, p. 47-62. http://dx.doi.org/10.14257/ijhit.2015.8.12.03

[9] Differential Evolution. [document électronique]. http://www.dii.unipd.it/~alotto/didattica/corsi/Elettrotecnica%20computazionale/DE.pdf

[10] A.R. Mehrabian et C. Lucas. A novel numerical optimization algorithm inspired from weed colonization, *Ecological Informatics*, Vol. 1, No. 4, 2006, p. 355-366.

[11] Amine Boumaza. How to design good Tetris players. [document électronique]. 2013. https://hal.inria.fr/hal-00926213 <hal-00926213>

[12] Jonas Balgaard Amundsen. A comparison of feature functions for Tetris strategies. 2014. [document électronique]. http://hdl.handle.net/11250/253729

[13] Applying Artificial Intelligence to Nintendo Tetris. [document électronique]. http://meatfighter.com/nintendotetrisai/#The_Algorithm

[14] Christophe Thiery, Bruno Scherrer. Construction d'un joueur artificiel pour Tetris, *Revue des Sciences et Technologies de l'Information- Série RIA: Revue d'Intelligence Artificielle, Lavoisier, 2009, Modélisation et décision pour les jeux*. Vol. 23, p.387-407. http://ria.revuesonline.com/article.jsp?articleId=13195. <10.3166/ria.23.387-407>. <inria-00418922>

[15] COLLECTIF. Métaheuristiques. Eyrolles, «Algorithmes», 2014, 515 pages.

Annexe B. utilities.py

```
def setAllArgs(obj, argdict):
    for n in list(argdict.keys()):
        if hasattr(obj, n):
            setattr(obj, n, argdict[n])
        else:
            print('Warning: parameter name', n, 'not found!')


def modifyf(f,rayon):
    def res(arr):
        for x in arr:
            if abs(x)>rayon:
                return -float("inf")
        return f(arr)
    return res
```

Annexe C. Ajout de pybrain/optimization/populationbased/ga.py

```
class GA_new(GA):
    rayon = 1
    constraint = False
    mutationProb = 0.1
    elitism = True
    eliteProportion = 0.05

    def __init__(self, evaluator = None, **kwargs):
        super().__init__(evaluator = evaluator, **kwargs)
        if self.constraint:
            self.setEvaluator(modifyf(evaluator,self.rayon))

    def initPopulation(self):
        self.currentpop = [array([self.rayon*uniform(-1,1) for _ in
            range(self.numParameters)]) for _ in range(self.populationSize)]
```


Annexe D. Algorithme des chauves-souris

D.1 bat_algorithms/__init__.py

```
from .bat import *
from .bat_algorithm import *
from .bat_algorithm_v2 import *
from .bat_algorithm_v3 import *
from .BA_DE import *
from .BA_DE_variant import *
```

D.2 bat_algorithms/bat.py

```
from numpy import array
from random import random,uniform
import math

class Bat(object):

    def __init__(self, idd, dimension, max_loudness= 1, r= 1):
        self.id = idd
        self.rayon = r
        self.position = array([self.rayon*uniform(-1,1) for _ in range(dimension)])
        self.new_position = array([self.rayon*random() for _ in range(dimension)])
        self.velocity = array([self.rayon*random() for _ in range(dimension)])
        self.pulse_rate0 = 0.85
        self.pulse_rate = 0.85
        self.loudness = max_loudness
        self.times = 0

    def getfitness(self, evaluator):
        self.fitness = evaluator(self.position)
        return self.fitness

    def update_loudness(self):
        alpha = 0.95
        self.loudness *= alpha

    def update_pulse_rate(self):
        gamma = 0.85
        self.times += 1
        self.pulse_rate = (1-math.exp(-gamma*self.times))*self.pulse_rate0

    def update_pulse_rate2(self,itera,n):
        self.pulse_rate = self.pulse_rate0 * (itera/n)**3

    def __repr__(self):
        s = "Bat number: " + str(self.id) + (
            "\nBat current position: " + str(self.position) +
            "\nBat current vilocity: " + str(self.velocity) +
            "\nBat current fitness: " + str(self.fitness))
        return s
```

D.3 bat_algorithms/bat_algorithm.py

```
"""
Version originelle:
BatAlgorithm version dans l'article
Mais BatAlgorithm2 semble plus logique
"""

from bat_algorithms.bat import Bat
from random import random, randint, uniform, gauss
from numpy import array, mean
from utilities import setAllArgs, modifyf
import math


class BatAlgorithm(object):

    min_frequency = 0
    max_frequency = 1

    # La précision du résultat depend largement de max_loudness
    max_loudness = 0.9
    gamma = 0.85
    rayon = 1

    best_proportion = 0.1

    storeallbestfitness = False
    constraint = False
    minimalize = False


def __init__(self, size, dimension, evaluator, **kwargs):

    self.population_size = size
    self.dimension = dimension
    setAllArgs(self, kwargs)
    self.epsilon = -1 if self.minimalize else 1
    if self.constraint:
        self.evaluator = modifyf(lambda x: self.epsilon*evaluator(x),self.rayon)
    else:
        self.evaluator = lambda x: self.epsilon*evaluator(x)
    self.init_bat()


def init_bat(self):

    self.bats_list = [Bat(i, self.dimension, self.max_loudness, self.rayon)
                      for i in range(self.population_size)]
    self.bats_pulse_frequency = [0] * self.population_size
    self.update_fitness()
    if self.storeallbestfitness:
        self.bestfitnesses=[self.epsilon*self.bats_list[0].fitness]


def init_optimize(self):
```



```
    for bat in self.bats_list:
        bat.loudness = self.max_loudness
        bat.pulse_rate = 0.85
        if mean(abs(bat.velocity)) < self.rayon/20:
            bat.velocity = array([v+gauss(0, self.rayon/2) for v in bat.velocity])



def optimize(self, number_of_iterations=1):

    self.init_optimize()

    for it in range(number_of_iterations):

        self.update_bats()
        self.average_loudness = mean([bat.loudness for bat in self.bats_list])

        for i,bat in enumerate(self.bats_list):

            if random() * self.max_loudness < bat.loudness:

                if random() > bat.pulse_rate:
                    choix = randint(0, int(self.population_size*self.best_proportion)-1)
                    target = self.bats_list[choix]
                    n_pos = self.localsearch(target)
                    if n_pos is not None:
                        bat.new_position = n_pos

                newfitness = self.evaluator(bat.new_position)
                if newfitness >= bat.fitness:
                    bat.position = bat.new_position
                    bat.fitness = newfitness
                    bat.update_loudness()
                    bat.pulse_rate = (1-math.exp(-self.gamma*it))*bat.pulse_rate0

        self.bats_list.sort(key = lambda bat: bat.fitness, reverse= True)

        if self.storeallbestfitness:
            self.bestfitnesses.append(self.epsilon*self.bats_list[0].fitness)

    return self.bats_list[0].position, self.epsilon*self.bats_list[0].fitness


def localsearch(self,target):
    decalage = array([
        self.average_loudness*uniform(-0.5,0.5) for _ in range(self.dimension)])
    return target.position + decalage


def update_bats(self):
    self.update_frequency()
    self.update_velocity()
    self.update_position()

def update_fitness(self):
    for bat in self.bats_list:
```

```
            bat.getfitness(self.evaluator)
        self.bats_list.sort(key = lambda bat: bat.fitness, reverse= True)

    def update_frequency(self):
        for i in range(self.population_size):
            value = self.min_frequency + random()*(
                   self.max_frequency - self.min_frequency)
            self.bats_pulse_frequency[i] = value

    def update_velocity(self):
        for i,bat in enumerate(self.bats_list):
            new_velocity = bat.velocity + (
                bat.position - (self.bats_list[0].position)) * self.bats_pulse_frequency[i]
            bat.velocity = new_velocity

    def update_position(self):
        for bat in self.bats_list:
            bat.new_position = bat.position + bat.velocity


'''
Il semble beaucoup plus raisonable d'écrire update_velocity comme ça,
effectivement ça marche mieux
'''

class BatAlgorithm2(BatAlgorithm):
    def update_velocity(self):
        for i,bat in enumerate(self.bats_list):
            new_velocity = bat.velocity - (
                bat.position - (self.bats_list[0].position)) * self.bats_pulse_frequency[i]
            bat.velocity = new_velocity


'''
Si on enlève directement update_bats (il ne reste que donc local_search)
Pour certain loudness les résultats restent indiférrent (0.1)
Pour d'autres ça change beaucoup (10,0.01)
Curieusement, même la première version marche mieux alors que
les bats ne bougent dans la bonne direction
Raison possible: on a tout simplement un mouvement d'une échelle correcte
(quand loudness ~ 0.1, l'échelle de local_search fournit les bons résultats)
Et c'est encore mieux si on indique une direction qui est la bonne (version 2)
'''

class BatAlgorithm3(BatAlgorithm):
    def update_bats(self):
        pass

class BatAlgorithm4(BatAlgorithm2):
    def localsearch(self,target):
        pass

"""
Problème de l'algorithme, souvent difficile d'avoir une bonne précision de
la solution, l'échelle de chaque pas est déterminée par le loudness
"""
```


D.4 bat_algorithms/bat_algorithm_v2.py

```
"""
On considére ici la modification adoptée dans
A Novel Adaptive Bat Algorithm to Control Explorations and Exploitations
    for Continuous Optimization Problems
La partie 3.2
- Adaptive Mutation Step Size
- 'Rechenbergs 1/5 mutation rule
"""

from bat_algorithms.bat_algorithm import BatAlgorithm2
from random import uniform, random, randint, gauss
from numpy import array, mean

class BA_v2(BatAlgorithm2):

    def init_optimize(self):
        super().init_optimize
        self.mutation = 0
        self.sigma = 1


    def optimize(self, number_of_iterations=1):

        self.init_optimize()

        for it in range(number_of_iterations):

self.aver_v = mean([bat.velocity for bat in self.bats_list])
self.update_bats()
self.average_loudness = mean([bat.loudness for bat in self.bats_list])

for i,bat in enumerate(self.bats_list):

    if random() > bat.pulse_rate:
        self.mutation += 1
        choix = randint(0, int(self.population_size*self.best_proportion)-1)
        target = self.bats_list[choix]
        n_pos = self.localsearch(target)
        if n_pos is not None:
            bat.new_position = n_pos

    if random() * self.max_loudness < bat.loudness:
        newfitness = self.evaluator(bat.new_position)
        if newfitness >= bat.fitness:
            bat.position = bat.new_position
            bat.fitness = newfitness
            bat.update_loudness()

self.bats_list.sort(key = lambda bat: bat.fitness, reverse= True)

if self.storeallbestfitness:
    self.bestfitnesses.append(self.epsilon*self.bats_list[0].fitness)

if self.mutation > (it+1)*self.population_size*0.2:
```

```
            for bat in self.bats_list:
                bat.pulse_rate /= gauss(0.85,0.01)
            self.sigma -= 0.01

        elif self.mutation < (it+1)*self.population_size*0.2:
            for bat in self.bats_list:
                bat.pulse_rate *= gauss(0.85,0.01)
            self.sigma += 0.01

    return self.bats_list[0].position, self.epsilon*self.bats_list[0].fitness

'''
def localsearch(self,target):
    decalage = array([
        self.average_loudness*gauss(0,self.sigma) for _ in range(self.dimension)])
    return target.position + decalage
  '''

def update_velocity(self):
    for i,bat in enumerate(self.bats_list):
        new_velocity = uniform(0.1,1)*bat.velocity - (
            bat.position - (self.bats_list[0].position)) * self.bats_pulse_frequency[i]
        bat.velocity = new_velocity


"""
Contradictoirement, avec quelques expériences, on trouve que:
1. L'emploie de 'Rechenbergs 1/5 mutation rule semble totalement indifférent
    dans les résultats
2. Le nouveau local search détériore le performance
3. Par contre, l'idée d'ajouter le facteur uniform(0.1,1) avant bat.velocity
    dans update_velocity améliore beaucoup le performance,
    mais il faut pas non plus enlever complètement le terme bat.velocity
"""
```


D.5 bat_algorithms/bat_algorithm_v3.py

```
"""
Reference:
A new modification approach on bat algorithm for solving optimization problems
"""

from bat_algorithms.bat_algorithm import BatAlgorithm
from random import random, randint, uniform, gauss
from numpy import array, mean
import math



class BA_v3(BatAlgorithm):

    n = 2
    thetainit = 0.6
    winit = 0.9
    wfin = 0.2
    smin = 0
    smax = 5
    sigfin = 0
    siginit = 1
    max_loudness = 0.95

def optimize(self, number_of_iterations=1):

    self.init_optimize()

    for it in range(number_of_iterations):

        self.theta1 = 1+ (self.thetainit-1)* (1-it/number_of_iterations)**self.n
        self.theta2 = 1- self.theta1
        self.w = self.wfin+ (self.winit-self.wfin)* (1-it/number_of_iterations)**self.n
        self.sigma = self.sigfin+ (
            self.siginit-self.sigfin)* (1-it/number_of_iterations)**self.n
        self.update_bats()
        self.average_loudness = mean([bat.loudness for bat in self.bats_list])

        for i,bat in enumerate(self.bats_list):

            if random() * self.max_loudness < bat.loudness:

                if random() > bat.pulse_rate:
                    choix = randint(0, int(self.population_size*self.best_proportion)-1)
                    target = self.bats_list[choix]
                    n_pos = self.localsearch(target)
                    if n_pos is not None:
                        bat.new_position = n_pos

                newfitness = self.evaluator(bat.new_position)
                if newfitness >= bat.fitness:
                    bat.position = bat.new_position
```

```
                bat.fitness = newfitness
                bat.update_loudness()
                bat.pulse_rate = (1-math.exp(-self.gamma*it))*bat.pulse_rate0


        self.bats_list.sort(key = lambda bat: bat.fitness, reverse= True)

        if self.storeallbestfitness:
            self.bestfitnesses.append(self.epsilon*self.bats_list[0].fitness)


    return self.bats_list[0].position, self.epsilon*self.bats_list[0].fitness



# IS1, IS2
def update_velocity(self):
    best_position = self.bats_list[0].position
    choix = int(min(abs(gauss(0,0.5)),0.9)*self.population_size)
    autre_position = self.bats_list[choix].position

    for i,bat in enumerate(self.bats_list):
        new_velocity = self.w* bat.velocity + (
                    self.theta1* (best_position-bat.position) +
                    self.theta2* (autre_position-bat.position))*(
                    self.bats_pulse_frequency[i])
        bat.velocity = new_velocity
```



```
'''
Il faut dire que dans la version précédente, en enlèvant localsearch,
la fonction devient beaucoup moins efficace
C'est moins vraie ici parce que update_velocity a été bcp améliorée
Comme d'habitude, loudness joue un rôle important dans la capacité de localsearch
'''

# IS3

def localsearch(self,target):

    best_fitness = self.bats_list[0].fitness
    worst_fitness = self.bats_list[-1].fitness
    if best_fitness == worst_fitness:
        return

    s = self.smin+ (self.smax-self.smin)*(
        (target.fitness-worst_fitness)/(best_fitness-worst_fitness))

    seeds = []
    for _ in range(int(s)):
        decalage = [self.average_loudness*gauss(0,self.sigma) for _ in range(self.dimension)
            ]
        decalage = array(decalage)
        seed = target.position + decalage
        seeds.append(seed)
```



```
        if seeds != []:
            best_seed = max(seeds, key= lambda x: self.evaluator(x))
            return best_seed



class BA_explor(BA_v3):
    def localsearch(self,target):
        pass



"""
Cette fois ci, les deux modifications se trouvent assez utiles
Et la solution est bcp améliorée
Mais pour la plupart des fonctions multimodales,
comme Ackley([-32,32]), Griewank([-600,600]), Michalewicz([0,pi]),
ça reste peu satifisfaisant,
contrairement à ce qui est dit dans l'article
"""
```

D.6 bat_algorithms/BA_DE.py

```
"""
Reference:

A Novel Bat Algorithm Based on Differential Operator and
Lévy Flights Trajectory
Simulated Annealing Optimization Bat Algorithm in Service
Migration Joining the Gauss Perturbation
"""

from bat_algorithms.bat_algorithm_v3 import BA_v3
import math
from numpy import zeros,array,mean,seterr,isinf
from numpy.random import choice as nu_ch
from random import randint, sample, random, uniform, gauss



class BA_DE(BA_v3):

    Cr = 0.9
    n = 2
    thetainit = 0.5
    winit = 0.9
    wfin = 0.2
    smin = 0
    smax = 5
    sigfin = 0
    siginit = 1
    max_loudness = 0.95


    def __init__(self, size, dimension, evaluator, **kwargs):
        super().__init__(size, dimension, evaluator, **kwargs)
        self.choisir_target = self.choisir_target_rand

    '''
    On stocke désormais la meilleure solution jamais rencontrée car on accepte
    maintenant éventuellement les solutions moins bonnes (voir change_bat)
    '''
    def init_bat(self):
        super().init_bat()
        if self.storeallbestfitness:
            self.currentbf=[self.epsilon*self.bats_list[0].fitness]
        self.best_evaluated = self.bats_list[0].position
        self.best_evaluation = self.bats_list[0].fitness


    # Changement principal: l'ajoute de l'opérateur de DE
    def optimize(self, number_of_iterations=1):

        self.init_optimize()
        self.n_it = number_of_iterations
```



```
        for it in range(self.n_it):

            self.init_iteration(it)
            self.update_bats()
            self.has_changed = False


            for i,bat in enumerate(self.bats_list):


                if self.update_condition(bat):

                    if random() > bat.pulse_rate:
                        target = self.choisir_target()
                        n_pos = self.localsearch(bat,target)
                    else:
                        n_pos = self.DE(bat)

                    self.change_bat(bat,n_pos)

            self.bats_list.sort(key = lambda bat: bat.fitness, reverse= True)

            if self.storeallbestfitness:
                self.bestfitnesses.append(self.epsilon*self.best_evaluation)
                self.currentbf.append(self.epsilon*self.bats_list[0].fitness)


        return self.best_evaluated, self.epsilon*self.best_evaluation, self.epsilon*self.
                bats_list[0].fitness


    # Inchangé
    def update_velocity(self):

        autre_position = self.choisir_target_gauss().position

        for i,bat in enumerate(self.bats_list):
            new_velocity = self.w* bat.velocity + (
                        self.theta1* (self.best_evaluated-bat.position) +
                        self.theta2* (autre_position-bat.position))*(
                        self.bats_pulse_frequency[i])
            bat.velocity = new_velocity


    # On remarque que theta1 varie de 0.5 à 0.8
    def init_iteration(self,it):

        self.theta1 = 0.8+ (self.thetainit-0.8)* (1-it/self.n_it)**self.n
        self.theta2 = 1- self.theta1
        self.w = self.wfin+ (self.winit-self.wfin)* (1-it/self.n_it)**self.n
        self.sigma = self.sigfin+ (
            self.siginit-self.sigfin)* (1-it/self.n_it)**self.n
        self.average_loudness = mean([bat.loudness for bat in self.bats_list])

    # NB: avec au moin une probabilité de 0.1 pour chercher les nouvelles solutions
    def update_condition(self,bat):
```

```
    return random() * self.max_loudness < max(0.1,bat.loudness)


def choisir_target_rand(self):
    choix = randint(0, int(self.population_size*self.best_proportion)-1)
    target = self.bats_list[choix]
    return target

def choisir_target_gauss(self):
    choix = int(min(abs(gauss(0,0.5)),0.9)*self.population_size)
    target = self.bats_list[choix]
    return target

# Inchangé
def localsearch(self,bat,target):
    return super().localsearch(target)


# DE/rand/1/bin
def DE(self,bat):

    a,b,c = sample(self.bats_list,3)
    pa, pb, pc = a.position, b.position, c.position
    jr = randint(0,self.dimension-1)
    n_pos = zeros(self.dimension)

    for j in range(self.dimension):
        if random() < self.Cr or j == jr:
            n_pos[j] = pc[j] + uniform(0.5,1)*(pa[j]-pb[j])
        else:
            n_pos[j] = bat.position[j]

    return n_pos


'''
Pour ne pas être piégé dans une extréma locale,
on accepte aussi de temps en temps une solution qui n'est pas meilleure
La mise à jour de pluse rate est aussi légèrement modifiée
'''
def change_bat(self,bat,n_pos):

    newfit1 = self.evaluator(bat.new_position)
    newfit2 = -float("inf") if n_pos is None else self.evaluator(n_pos)
    tmp = list(zip([bat.new_position,n_pos],[newfit1,newfit2]))
    tmp.sort(key=lambda x: x[1], reverse = True)

    if not isinf(tmp[0][1]) and (tmp[0][1] > bat.fitness
        or random() * self.max_loudness < bat.loudness*0.5):

        bat.position = tmp[0][0]
        bat.fitness = tmp[0][1]
        bat.update_loudness()
        bat.update_pulse_rate()
        self.has_changed = True
```



```
        if tmp[0][1] > self.best_evaluation:
            self.best_evaluation = tmp[0][1]
            self.best_evaluated = tmp[0][0]



def around(x):
    return x if x > 1e-299 else 0

def Boltzmann(energies,T):

    energies -= energies[0]
    prob = array([around(math.exp(-E/T)) for E in energies])
    Z = sum(prob)
    ans = prob/Z
    return ans



'''
Pour la température
                Ackley Griewank Rosenbrock Michalewicz Zakharov
fixé X O @ X @
croissante O O @ O @
décroissante O X @ O @
'''

class BA_DE_T(BA_DE):


    def __init__(self, size, dimension, evaluator, **kwargs):
        super().__init__(size, dimension, evaluator, **kwargs)
        self.choisir_target = self.choisir_target_boltzmann


    def init_optimize(self):

        for bat in self.bats_list:
            bat.loudness = self.max_loudness
            bat.pulse_rate = 0.85

        self.T0 = abs(self.best_evaluation)
        self.has_changed = True


    def init_iteration(self,it):

        super().init_iteration(it)
        self.T = self.T0 * ((it+1)/self.n_it)**self.n
        #self.T = self.T0 * 1.2**it
        #self.T = self.T0 *0.9**it
        if self.has_changed:
            self.energies = array([-b.fitness+self.best_evaluation for b in self.bats_list])
            self.prob_vect = Boltzmann(self.energies,self.T)
```

```
    def choisir_target_boltzmann(self):
        pv = self.prob_vect if random()<self.theta1 else None
        target = nu_ch(self.bats_list, p=pv)
        return target


"""
De façon générale, cette version de l'algorithme batte toutes les versions précédentes

Dans BA_DE_T
Le choix de choisir_target_boltzmann à la place de choisir_target_gauss
peut être favorable (Ackley), mais en fait le plus souvent indéffirent
"""
```


D.7 bat_algorithms/BA_DE_variant.py

```
"""
Il y a trois modifications importantes
1. Cherche une nouvelle solution tant que random > max(0.1,self.loudness)
   (pour continuer à chercher des nouvelles solutions,
    mais ensuite greedy marche encore mieux)
2. L'emploie de mutation (c'est nécessaire, sinon le 1 ne sert pas forcément)
3. La nouvelle update_velocite s'inspirant de DE

NB: le 1 est maintenant aussi codé dans BA_DE

Les deux premières nous permettent de trouver la minima globale à une longue échelle
(environ 1250 générations à la place de 200 générations)
Surtout utiles pour Michalewicz, Rastrigin, Rosenbrock
Il ne nous reste que Griewank
"""

from bat_algorithms.BA_DE import BA_DE, BA_DE_T, Boltzmann
import math
import time
from numpy import zeros,array,array_equal,isinf
from numpy.random import choice as nu_ch
from random import randint, sample, random, uniform, gauss, choice
from copy import copy


class BA_DE_T_v2(BA_DE_T):

    Cr = 0.9
    n = 2
    thetainit = 0.5
    max_loudness = 0.95
    c = 340


    def __init__(self, size, dimension, evaluator, **kwargs):
        super().__init__(size, dimension, evaluator, **kwargs)
        self.choisir_target = self.choisir_target_boltzmann


    '''
    Ca a effectivement un effet positif dans la plupart de cas
    (et pas négligeable)
    '''
    def update_velocity(self):

        for i,bat in enumerate(self.bats_list):

            if random() < self.theta1:
                bonne_position = self.best_evaluated
            else:
                bonne_position = self.choisir_target_gauss().position

            autre_position = self.choisir_target_gauss().position
```

```
        add_velocity = zeros(self.dimension)
        for j in range(self.dimension):
            if random()<self.theta1:
                add_velocity[j] = bonne_position[j] - bat.position[j]
            else:
                add_velocity[j] = autre_position[j] - bat.position[j]

        bat.velocity = self.w* bat.velocity + add_velocity*self.bats_pulse_frequency[i]

'''
Greedy converge plus rapidement (générations)
Mais ça prend 4 fois plus de temps
(Problème: pourqoui ne pas greedy depuis le début,
 quel est le rôle de cette comparaison dans l'algorithme originel,
 vois pas trop...)
'''
def update_condition(self,bat):
    return random() * self.max_loudness < max(0.1,bat.loudness)


'''
Le terme uniform(0.5,1)*(self.best_evaluated[j]-pc[j])
augmente le vitess de convergence (utilisons simplement converge_trace pour voir)
Et il ne se révele pas de conduire à une convergence prématurée (effet négligeable)
'''
def DE(self,bat):

    a,b,c = sample(self.bats_list,3)
    pa, pb, pc = a.position, b.position, c.position
    jr = randint(0,self.dimension-1)
    n_pos = zeros(self.dimension)

    for j in range(self.dimension):
        if random() < self.Cr or j == jr:
            n_pos[j] = pc[j] + uniform(0.5,1)*(pa[j]-pb[j]) +(
                uniform(0.5,1)*(self.best_evaluated[j]-pc[j]))
        else:
            n_pos[j] = bat.position[j]

    return n_pos


# On introduit la mutation
def change_bat(self,bat,n_pos):

    m = self.mutate(bat)
    newfit1 = self.evaluator(bat.new_position)
    newfit2 = -float("inf") if n_pos is None else self.evaluator(n_pos)
    newfit3 = -float("inf")
    newfit3 = self.evaluator(m)
    tmp = list(zip([bat.new_position,n_pos,m,bat.position],[newfit1,newfit2,newfit3,bat.
        fitness]))
    tmp.sort(key=lambda x: x[1], reverse = True)


    if not isinf(tmp[0][1]) and (tmp[0][1] > bat.fitness
```



```
        or random() < self.theta1/2):

        bat.position = tmp[0][0]
        bat.fitness = tmp[0][1]
        bat.update_loudness()
        bat.update_pulse_rate()
        self.has_changed = True

        if tmp[0][1] > self.best_evaluation:
            self.best_evaluation = tmp[0][1]
            self.best_evaluated = tmp[0][0]

    def mutate(self,bat):
        v = copy(bat.position)
        for i in range(self.dimension):
            if random() < 0.2:
                v[i] += gauss(0,0.5)
        return v


class BA_DE_T_greedy(BA_DE_T_v2):

    def update_condition(self,bat):
        return 1


'''
Voici la version finale, adapté pour l'entrainement des bots
Comme le fitness d'un individu peut varie beaucoup et depend fortement
de la chance, les fitness sont reévaluer à chaque fois
Ainsi on ne peut que prendre la meilleure solution à chaque instant
pour bien guider afin d'éviter d'être égaré par un candidat mal évalué
On imprime à chaque itération quelques informations nécessaire pour l'expérience
'''
class BA_DE_T_v3(BA_DE_T_v2):

    def __init__(self, size, dimension, evaluator, **kwargs):
        super().__init__(size, dimension, evaluator, **kwargs)
        self.t0 = time.time()

    def update_bats(self):
        self.update_frequency()
        self.update_velocity()
        self.update_position()
        self.update_fitness()

    def init_iteration(self,it):
        super().init_iteration(it)
        print(it)
        print(time.time()-self.t0)
        print(self.best_evaluation,self.best_evaluated)
        self.t0 = time.time()
        self.best_evaluated = self.bats_list[0].position
        self.best_evaluation = self.bats_list[0].fitness
```

Annexe E. differential_evolution/DE_rand_1_bin.py

```
"""
DE/rand/1/bin

Reference:
Improved Hybridized Bat Algorithm for Global Numerical Optimization
DE
"""

from random import random,sample,uniform,randint
from numpy import array,zeros
from utilities import setAllArgs, modifyf



class DE(object):

    # Differential weight
    F = 0.8
    # Crossover probability
    Cr = 0.9
    rayon = 1

    minimalize = False
    storeallgenerations = False
    storeallbestfitness = False
    constraint = False


def __init__(self, size, dimension, evaluator, **kwargs):

    self.population_size = size
    self.dimension = dimension
    setAllArgs(self, kwargs)
    self.epsilon = -1 if self.minimalize else 1
    if self.constraint:
        self.evaluator = modifyf(lambda x: self.epsilon*evaluator(x),self.rayon)
    else:
        self.evaluator = lambda x: self.epsilon*evaluator(x)

    self.currentpop = [array([uniform(-1,1)*self.rayon
        for _ in range(self.dimension)]) for _ in range(self.population_size)]
    self.fitnesses = [self.evaluator(x) for x in self.currentpop]

    if self.storeallgenerations:
        self.generations = [self.currentpop]
    if self.storeallbestfitness:
        self.bestfitnesses = [self.epsilon*max(self.fitnesses)]


def optimize(self, number_of_iterations=1):

    for _ in range(number_of_iterations):

        self.init_iter()
```



```
        for (i,xi) in enumerate(self.currentpop):

            a,b,c = sample(range(self.population_size),3)
            xa, xb, xc = self.currentpop[a], self.currentpop[b], self.currentpop[c]
            jr = randint(0,self.dimension-1)
            v = zeros(self.dimension)

            for j in range(self.dimension):
                if random() < self.Cr or j == jr:
                    v[j] = xc[j] + uniform(0.5,1)*(xa[j]-xb[j])
                else:
                    v[j] = xi[j]

            newfitness = self.evaluator(v)
            if newfitness > self.fitnesses[i]:
                self.currentpop[i] = v
                self.fitnesses[i] = newfitness

        if self.storeallgenerations:
            self.generations.append(self.currentpop)
        if self.storeallbestfitness:
            self.bestfitnesses.append(self.epsilon*max(self.fitnesses))

    best_indice = max(list(range(self.population_size)), key= lambda i:self.fitnesses[i])
    best_value = self.epsilon * self.fitnesses[best_indice]

    return self.currentpop[best_indice], best_value


    def init_iter(self):
        pass


# adpaté pour l'entrainement des bots
class DE_adapte(DE):

    def init_iter(self):
        self.fitnesses = [self.evaluator(x) for x in self.currentpop]
```

Annexe F. Tests des algorithmes

F.1 evalu_functions/numerical_functions.py

```
from numpy import mean, std
from math import *
import time


# U

def Sphere(arr):
    return sum([x**2 for x in arr])

def Zakharov(arr):
    square_sum = sum([x**2 for x in arr])
    ix_sum = sum([(i+1)*x for (i,x) in enumerate(arr)])
    return square_sum + (0.5*ix_sum)**2 + (0.5*ix_sum)**4

def Rosenbrock(arr):
    return sum([100*(arr[i+1]-arr[i]**2)**2 + (arr[i]-1)**2 for i in range(len(arr)-1)])


# M

def Ackley(arr):
    d = len(arr)
    square_sum = sum([x**2 for x in arr])
    cos_sum = sum([cos(2*pi*x) for x in arr])
    return -20*exp(-0.2*sqrt(square_sum/d))-exp(cos_sum/d)+20+exp(1)

def Griewank(arr):
    produit = 1
    for (i,x) in enumerate(arr):
        produit *= cos(x/sqrt(i+1))
    return sum([x**2 for x in arr])/4000 - produit + 1

def Rastrigin(arr):
    d = len(arr)
    return 10*d + sum([x**2-10*cos(2*pi*x) for x in arr])

def Schwefel(arr):
    d = len(arr)
    try:
        return 418.9829*d-sum([x*sin(sqrt(abs(x))) for x in arr])
    except ValueError:
        print(arr)

def Salomon(arr):
    square_sum = sum([x**2 for x in arr])
    return -cos(2*pi*sqrt(square_sum)) + 0.1*sqrt(square_sum) + 1

# Testé acec d= 2,5,10
def Michalewicz(arr):
    arr = [x+pi/2 for x in arr]
    penal = 0
```



```
    for x in arr:
        if x<0 or x>pi:
            penal += 20
    return -sum([sin(x)*(sin((i+1)*x**2/pi))**20 for (i,x) in enumerate(arr)]) + penal

# Testé avec d=2, +1 comparé avec le vesion normale
def Easom(arr):
    produit = 1
    for x in arr:
        produit *= cos(x)
    sq_sum = sum([(x-pi)**2 for x in arr])
    return -(-1)**len(arr)*produit*exp(-sq_sum)+1

# Testé avec d=2, +1 comparé avec le vesion normale
def Dropwave(arr):
    square_sum = sum([x**2 for x in arr])
    return -(1+cos(12*sqrt(square_sum)))/(square_sum/2+2)+1



'''
temps fixé
'''
def test(f, ba, **kwargs):
    d, N, t = kwargs.get('d',10), kwargs.get('N',50), kwargs.get('t',30)
    n, r = kwargs.get('n',1000), kwargs.get('r',1)
    const = kwargs.get('const',True)
    times = 0
    fits = []
    t0 = time.time()
    while time.time()-t0 < t:
        bat_a = ba(N,d,f, rayon= r, minimalize= True, constraint= const)
        fitness = bat_a.optimize(n)[1]
        fits.append(fitness)
        times += 1
    return min(fits),max(fits),mean(fits),std(fits),times



def test_ga(f, ga, **kwargs):
    d, N, t = kwargs.get('d',10), kwargs.get('N',50), kwargs.get('t',30)
    n, r = kwargs.get('n',1000), kwargs.get('r',1)
    const = kwargs.get('const',True)
    times = 0
    fits = []
    t0 = time.time()
    while time.time()-t0 < t:
        g = ga(lambda x: -f(x), numParameters= d, populationSize= N, rayon= r, constraint=
            const)
        fitness = -g.learn(n)[1]
        fits.append(fitness)
        times += 1
    return min(fits),max(fits),mean(fits),std(fits),times
```

```
'''
nombre de boucles exécutées fixé
'''
def test2(f, ba, **kwargs):
    d, N, times = kwargs.get('d',10), kwargs.get('N',50), kwargs.get('times',30)
    n, r = kwargs.get('n',1000), kwargs.get('r',1)
    const = kwargs.get('const',True)
    fits = []
    t0 = time.time()
    for _ in range(times):
        bat_a = ba(N,d,f, rayon= r, minimalize= True, constraint= const)
        fitness = bat_a.optimize(n)[1]
        fits.append(fitness)
    return min(fits),max(fits),mean(fits),std(fits),time.time()-t0



def test_ga2(f, ga, **kwargs):
    d, N, times = kwargs.get('d',10), kwargs.get('N',50), kwargs.get('times',30)
    n, r = kwargs.get('n',1000), kwargs.get('r',1)
    const = kwargs.get('const',True)
    fits = []
    t0 = time.time()
    for _ in range(times):
        g = ga(lambda x: -f(x), numParameters= d, populationSize= N, rayon= r, constraint=
            const)
        fitness = -g.learn(n)[1]
        fits.append(fitness)
    return min(fits),max(fits),mean(fits),std(fits),time.time()-t0



def t_value(x1,x2):
    m1, m2, sd1, sd2, n1, n2 = x1[2], x2[2], x1[3], x2[3], x1[4], x2[4]
    return (m1-m2)/sqrt(sd1**2/(n1-1)+sd2**2/(n2-1))
```


F.2 evalu_functions/converge.py

```
from matplotlib import pyplot as plt
import time
import numpy as np

def trace_converge(f, algo, **kwargs):

    d, N = kwargs.get('d',10), kwargs.get('N',50)
    n, r = kwargs.get('n',1000), kwargs.get('r',1)
    const = kwargs.get('const',True)
    log = kwargs.get('log',True)
    second = kwargs.get('second',False)
    title, courbe = kwargs.get('title', None), kwargs.get('courbe',None)

    plt.xlabel("iteration", fontsize=12)
    plt.ylabel("fitness value", fontsize=12)
    if title is not None:
        plt.suptitle(title, fontsize=14)

    t0 = time.time()
    al = algo(N,d,f,rayon=r,minimalize=True,storeallbestfitness=True, constraint=const)
    al.optimize(n)
    gene = list(range(n+1))
    if log:
        plt.yscale('log')
    plt.plot(gene,al.bestfitnesses, label = courbe)
    if second:
        plt.plot(gene,al.currentbf)
    if courbe is not None:
        plt.legend()
    plt.show()
    return time.time() - t0

def trace_converge_ga(f, algo, **kwargs):

    d, N = kwargs.get('d',10), kwargs.get('N',50)
    n, r = kwargs.get('n',1000), kwargs.get('r',1)
    const = kwargs.get('const',True)
    log = kwargs.get('log',True)
    courbe = kwargs.get('courbe',None)

    t0 = time.time()
    al = algo(lambda x: -f(x), numParameters= d, populationSize= N, rayon= r, constraint=
        const)
    bestfounds = []
    for _ in range(n+1):
        bestfounds.append(-al.learn(0)[1])
    gene = list(range(n+1))
    if log:
        plt.yscale('log')
    plt.plot(gene,bestfounds,label = courbe)
    if courbe is not None:
        plt.legend()
    plt.show()
    return time.time() - t0
```

Annexe G. Pong

G.1 pong/pong_game.py

```
from random import randint
from utilities import setAllArgs


class pong(object):

    WIDTH = 600
    HEIGHT = 400
    BALL_RADIUS = 20
    PAD_WIDTH = 8


    def __init__(self, PAD_HEIGHT= 80, **kwargs):
        self.PAD_HEIGHT = PAD_HEIGHT
        self.HALF_PAD_HEIGHT = self.PAD_HEIGHT / 2
        setAllArgs(self, kwargs)
        self.init_game()

    def init_game(self):
        self.score1, self.score2 = 0,0
        self.success_1, self.success_2 = 0,0
        self.restart()


def init_ball(self,right,up):
    self.ball_pos = [self.WIDTH/2, self.HEIGHT/2]
    self.ball_vel = [randint(2,5) for _ in range(2)]
    self.directionx = 1 if right else -1
    self.directiony = 1 if up else -1

def init_paddle(self):
    self.paddle1_vel = 5
    self.paddle2_vel = 5
    self.paddle1_pos = [0, self.HEIGHT/2-self.PAD_HEIGHT/2]
    self.paddle2_pos = [self.WIDTH-self.PAD_WIDTH, self.HEIGHT/2-self.PAD_HEIGHT/2]


def restart(self):
    self.init_paddle()
    self.init_ball(randint(0,1),randint(0,1))



def update_ball_pos(self):

    self.check_collision1()
    self.check_collision2()

    if self.ball_pos[1]>self.HEIGHT-self.BALL_RADIUS or (
        self.ball_pos[1] < self.BALL_RADIUS):
        self.directiony *= -1
```



```
    self.ball_pos[0] += self.ball_vel[0]*self.directionx
    self.ball_pos[1] += self.ball_vel[1]*self.directiony

    if self.ball_pos[0]<0:
        self.score2 +=1
        self.restart()
    elif self.ball_pos[0]>self.WIDTH:
        self.score1 +=1
        self.restart()


def check_collision2(self):

    if self.ball_pos[0] > self.WIDTH - self.BALL_RADIUS - self.PAD_WIDTH and (
        abs((self.paddle2_pos[1]+self.HALF_PAD_HEIGHT)-self.ball_pos[1]) < (
        self.HALF_PAD_HEIGHT+self.BALL_RADIUS)):

        self.directionx *= -1 #change direction
        self.ball_vel[0] += 1 # speed up ball in x
        self.ball_vel[1] += 1 # speed up ball in y
        self.paddle1_vel += 1 # speed up paddle1
        self.paddle2_vel += 1 # speed up paddle2
        self.success_2 += 1

def check_collision1(self):

    if self.ball_pos[0] < self.BALL_RADIUS + self.PAD_WIDTH and (
        abs((self.paddle1_pos[1]+self.HALF_PAD_HEIGHT)-self.ball_pos[1]) < (
        self.HALF_PAD_HEIGHT+self.BALL_RADIUS)):

        self.directionx *= -1 #change direction
        self.ball_vel[0] += 1 # speed up ball in x
        self.ball_vel[1] += 1 # speed up ball in y
        self.paddle1_vel += 1 # speed up paddle1
        self.paddle2_vel += 1 # speed up paddle2
        self.success_1 +=1


def update_pos_paddle1(self,n):

    if n <= -0.5:
        self.paddle1_pos[1] += self.paddle1_vel
        if self.paddle1_pos[1] > self.HEIGHT-self.PAD_HEIGHT:
            self.paddle1_pos[1] = self.HEIGHT-self.PAD_HEIGHT
    if n >= 0.5:
        self.paddle1_pos[1] -= self.paddle1_vel
        if self.paddle1_pos[1] < 0:
            self.paddle1_pos[1] = 0

def update_pos_paddle2(self,n):

    if n <= -0.5:
        self.paddle2_pos[1] += self.paddle2_vel
        if self.paddle2_pos[1] > self.HEIGHT-self.PAD_HEIGHT:
            self.paddle2_pos[1] = self.HEIGHT-self.PAD_HEIGHT
    if n >= 0.5:
```

```
            self.paddle2_pos[1] -= self.paddle2_vel
            if self.paddle2_pos[1] < 0:
                self.paddle2_pos[1] = 0


class pong_seul(pong):

    def check_collision2(self):

        if self.ball_pos[0] > self.WIDTH - self.BALL_RADIUS - self.PAD_WIDTH:

            self.directionx *= -1
            self.ball_vel[0] += 1
            self.ball_vel[1] += 1
            self.paddle1_vel += 1
```

G.2 pong/pong_bot.py

```
from pybrain.tools.shortcuts import buildNetwork

def bot_simple(pon):

    posy = pon.paddle1_pos[1]
    vel = pon.ball_vel
    pos = pon.ball_pos
    pp,bvx,bvy,bpx,bpy = posy,vel[0]*pon.directionx,vel[1]*pon.directiony,pos[0],pos[1]

    if bpy > pp + pon.HALF_PAD_HEIGHT:
        return -1
    elif bpy < pp + pon.HALF_PAD_HEIGHT:
        return 1
    return 0


def bot_neuron(hide,params):

    assert len(params) == 7*hide+1

    n = buildNetwork(5,hide,1)
    n._setParameters(params)

    def player(pon):
        posy = pon.paddle1_pos[1]
        vel = pon.ball_vel
        pos = pon.ball_pos
        return n.activate([posy,vel[0]*pon.directionx,vel[1]*pon.directiony,pos[0],pos[1]])

    return player
```


G.3 pong/pong_evalu.py

```
from pong.pong_game import *
import pygame


class pong_evalu_seul(pong_seul):

    def evalu(self,bot,interval= 1):

        cmp, cmp2, res = 0,0,0

        up = 250
        down = 150

        self.init_game()

        while max(self.score1,self.score2)<=5:

            posy = self.paddle1_pos[1]
            vel = self.ball_vel
            pos = self.ball_pos

            old_suc = self.success_1
            old_sc2 = self.score2
            self.update_ball_pos()
            if cmp % interval == 0:
                res1 = bot(self)
                self.update_pos_paddle1(res1)
            cmp +=1


            if down <= posy and posy <= up:
                cmp2 +=1
            else:
                cmp2 = 0
                up = posy + 50
                down = posy - 50

            if self.score2 == old_sc2+1:
                cmp2 = 0
            if self.success_1 == old_suc+1:
                res += cmp2
                cmp2 = 0

        return cmp



class pong_display_seul(pong_seul):

    def display(self,bot,interval= 1):

        black = (0, 0, 0)
        white = (255, 255, 255)
        blue = (0, 0, 255)
```

```
size = [self.WIDTH, self.HEIGHT]
screen = pygame.display.set_mode(size)

pygame.display.set_caption('Test')

clock = pygame.time.Clock()

cmp = 0
done = False
self.init_game()

while done == False:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True

    screen.fill(black)

    self.update_ball_pos()
    if cmp % interval == 0:
        res1 = bot(self)
        self.update_pos_paddle1(res1)


    ball_x = int(self.ball_pos[0])
    ball_y = int(self.ball_pos[1])

    pygame.draw.circle(screen, white, [ball_x, ball_y], self.BALL_RADIUS)
    pygame.draw.rect(screen, blue, [self.paddle1_pos[0],self.paddle1_pos[1],
                                    self.PAD_WIDTH,self.PAD_HEIGHT])
    pygame.draw.rect(screen, blue, [self.paddle2_pos[0],0,
                                    self.PAD_WIDTH,self.HEIGHT])

    pygame.display.flip()
    clock.tick(30)
    cmp += 1

pygame.quit ()
```


G.4 pong/pong_training.py

```
from pong.pong_bot import *
from pong.pong_evalu import *
from pybrain.optimization.populationbased.ga import GA_new
from differential_evolution.DE_rand_1_bin import DE_adapte
import time

# réseaux neuron (5,7,1), interval = 1, PAD_HEIGHT, BALL_RADIUS normaux
def training1(param):
    return pong_evalu_seul().evalu(bot_neuron(7,param))

# interval = 2
def training2(param):
    return pong_evalu_seul().evalu(bot_neuron(7,param),2)

'''
Pour obtenir le résultat de l'entrainement par ga
Pas emballé dans une fonction ou une classe car c'est plus pratique comme ça

GAa = GA_new(training1, numParameters= 50, populationSize= 20,
  mutationProb= 0.2, elitism= True, _eliteSize= 1)
'''

'''
fichier = open("ga_30.txt","a")
for i in range(31):
    t0 = time.time()
    print(i+30)
    fichier.write(str(i+30)+"\n\n")
    res = GAa.learn(0)
    print(res)
    fichier.write(str(res)+"\n\n")
    i0 = max(range(20), key= lambda j:GAa.fitnesses[j])
    print((GAa.currentpop[i0],GAa.fitnesses[i0]))
    fichier.write(str((GAa.currentpop[i0],GAa.fitnesses[i0]))+"\n\n")
    print(time.time()-t0)
    fichier.write(str(time.time()-t0)+"\n\n\n")
fichier.close()


DEa = DE_adapte(20,50,training1)

fichier = open("DE_30.txt","a")
for i in range(31):
    t0 = time.time()
    print(30)
    fichier.write(str(30)+"\n\n")
    res = DEa.optimize(1)
    print(res)
    fichier.write(str(res)+"\n\n")
    print(time.time()-t0)
    fichier.write(str(time.time()-t0)+"\n\n\n")
fichier.close()
'''
```

Annexe H. Tetris

H.1 tetris/tetris_game/tetris_game_only.py

```
from random import randrange as rand
from random import randint


# Define the shapes of the single parts
tetris_shapes = [
    [[1, 1, 1],
     [0, 1, 0]],

    [[0, 2, 2],
     [2, 2, 0]],

    [[3, 3, 0],
     [0, 3, 3]],

    [[4, 0, 0],
     [4, 4, 4]],

    [[0, 0, 5],
     [5, 5, 5]],

    [[6, 6, 6, 6]],

    [[7, 7],
     [7, 7]]
]


def rotate_clockwise(shape):
    return [ [ shape[y][x]
        for y in range(len(shape)) ]
        for x in range(len(shape[0]) - 1, -1, -1) ]

def join_matrixes(mat1, mat2, mat2_off):
    off_x, off_y = mat2_off
    for cy, row in enumerate(mat2):
        for cx, val in enumerate(row):
            mat1[cy+off_y-1][cx+off_x] += val
    return mat1


def remove_matrixes(mat1, mat2, mat2_off):
    off_x, off_y = mat2_off
    for cy, row in enumerate(mat2):
        for cx, val in enumerate(row):
            if val:
                mat1[cy+off_y-1][cx+off_x] = 0
    return mat1


class grille(object):
```



```
def __init__(self, cols=10, rows=22):
    self.cols = cols
    self.rows = rows
    self.board = self.new_board()

def check_collision(self,shape,offset):
    off_x, off_y = offset
    for cy, row in enumerate(shape):
        for cx, cell in enumerate(row):
            try:
                if cell and self.board[ cy + off_y ][ cx + off_x ]:
                    return True
            except IndexError:
                    return True
    return False

def remove_rows(self,rows):
    newboard = self.new_board()
    del_rows = []
    rows.reverse()
    decalage = len(rows)
    for y in range(decalage,self.rows):
        while rows != [] and y-decalage == rows[-1]:
            del_rows.append((rows[-1],self.board[rows[-1]]))
            rows.pop()
            decalage -= 1
        newboard[y] = self.board[y-decalage]
    while rows != [] and y+1-decalage == rows[-1]:
            del_rows.append((rows[-1],self.board[rows[-1]]))
            rows.pop()
            decalage -= 1
    self.board = newboard
    return del_rows

def add_rows(self,rows):
    oldboard = self.new_board()
    rows.reverse()
    decalage = len(rows)
    y = 0
    while y < self.rows:
        while rows != [] and y == rows[-1][0]:
            oldboard[y] = rows[-1][1]
            y += 1
            decalage -= 1
            rows.pop()
        oldboard[y] = self.board[y+decalage]
        y += 1
    self.board = oldboard

def new_board(self):
    board = [ [ 0 for x in range(self.cols) ] for y in range(self.rows) ]
    board += [[ 1 for x in range(self.cols)]]
    return board
```

```
class TetrisApp1(object):

    def __init__(self, cols=10, rows=22):
        self.cols = cols
        self.rows = rows
        self.grille = grille(cols,rows)
        self.next_stone = tetris_shapes[rand(len(tetris_shapes))]
        self.init_game()

    def new_stone(self):
        self.stone = self.next_stone[:]
        self.next_stone = tetris_shapes[rand(len(tetris_shapes))]
        self.stone_x = int(self.cols / 2 - len(self.stone[0])/2)
        self.stone_y = 0

        if self.grille.check_collision(self.stone,
                        (self.stone_x, self.stone_y)):
            self.gameover = True

    def init_game(self):
        self.grille.board = self.grille.new_board()
        self.heights = []
        self.new_stone()
        self.lines = 0

    def add_cl_lines(self, n):
        self.lines += n
        self.heights.extend([self.height_max()]*n)

def move(self, delta_x):
    if not self.gameover:
        new_x = self.stone_x + delta_x
        if new_x < 0:
            new_x = 0
        if new_x > self.cols - len(self.stone[0]):
            new_x = self.cols - len(self.stone[0])
        if not self.grille.check_collision(
            self.stone,(new_x, self.stone_y)):
            self.stone_x = new_x

def drop(self):
    if not self.gameover:
        self.stone_y += 1
        if self.grille.check_collision(self.stone,
                                       (self.stone_x, self.stone_y)):
            self.grille.board = join_matrixes(
                self.grille.board,
                self.stone,
                (self.stone_x, self.stone_y))
            self.new_stone()
            cleared_rows = []
            for i in range(self.rows):
                row = self.grille.board[i]
                if 0 not in row:
                    cleared_rows.append(i)
            if cleared_rows != []:
```



```
                self.add_cl_lines(len(cleared_rows))
                self.grille.remove_rows(cleared_rows)
            return True
    return False

def insta_drop(self):
    if not self.gameover:
        while(not self.drop()):
            pass

def rotate_stone(self):
    if not self.gameover:
        new_stone = rotate_clockwise(self.stone)
        if not self.grille.check_collision(new_stone,
                                           (self.stone_x, self.stone_y)):
            self.stone = new_stone


def transformer(self):
    bo_trans = [[self.grille.board[j][i] for j in range(self.rows+1)] for i in range(self.
        cols)]
    return bo_trans

def set_height_colonne(self):
    bo_trans = self.transformer()
    self.h_c = [self.rows]*self.cols
    for x,col in enumerate(bo_trans):
        y = 0
        while col[y] == 0:
            y += 1
            self.h_c[x] -= 1

    def height_max(self):
        self.set_height_colonne()
        return max(self.h_c)


class TetrisApp_z(TetrisApp1):

    def __init__(self, cols=10, rows=22):
        self.cols = cols
        self.rows = rows
        self.grille = grille(cols,rows)
        self.next_stone = tetris_shapes[randint(1,2)]
        self.init_game()

    def new_stone(self):
        self.stone = self.next_stone[:]
        self.next_stone = tetris_shapes[randint(1,2)]
        self.stone_x = int(self.cols / 2 - len(self.stone[0])/2)
        self.stone_y = 0

        if self.grille.check_collision(self.stone,
                        (self.stone_x, self.stone_y)):
            self.gameover = True
```

```
class TetrisApp_z_ordre(TetrisApp1):

    def __init__(self, cols=10, rows=22):
        self.cols = cols
        self.rows = rows
        self.grille = grille(cols,rows)
        self.shape = 0
        self.next_stone = tetris_shapes[self.shape%2+1]
        self.init_game()

    def new_stone(self):
        self.stone = self.next_stone[:]
        self.shape += 1
        self.next_stone = tetris_shapes[self.shape%2+1]
        self.stone_x = int(self.cols / 2 - len(self.stone[0])/2)
        self.stone_y = 0

        if self.grille.check_collision(self.stone,
                        (self.stone_x, self.stone_y)):
            self.gameover = True



class TetrisApp_dans_lordre(TetrisApp1):

    def __init__(self, cols=10, rows=22):
        self.cols = cols
        self.rows = rows
        self.grille = grille(cols,rows)
        self.shape = 0
        self.next_stone = tetris_shapes[self.shape]
        self.init_game()

    def new_stone(self):
        self.stone = self.next_stone[:]
        self.shape += 1
        self.next_stone = tetris_shapes[self.shape%(len(tetris_shapes))]
        self.stone_x = int(self.cols / 2 - len(self.stone[0])/2)
        self.stone_y = 0

        if self.grille.check_collision(self.stone,
                        (self.stone_x, self.stone_y)):
            self.gameover = True
```

H.2 tetris/tetris_AI/__init__.py

```
from .tetris_bot import *
from .tetris_evalu import *
from .tetris_scorer import *
```


H.3 tetris/tetris_AI/tetris_scorer.py

```
"""
Reference:

A comparison of feature functions for Tetris strategies
https://luckytoilet.wordpress.com/2011/05/27/coding-a-tetris-ai-using-a-genetic-algorithm/
https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/
"""

from tetris.tetris_game.tetris_game_only import *

class boardscorer(object):

    def __init__(self,param,board,cleared,land_h):
        self.rows = len(board)-1
        self.cols = len(board[0])
        self.param = param
        self.board_nor = board
        self.board = self.transformer(board)
        self.set_height_colonne()
        self.clear_lines = cleared
        self.landing_height = land_h

    def transformer(self,board):
        bo_trans = [[board[j][i] for j in range(len(board))] for i in range(len(board[0]))]
        return bo_trans

    def set_height_colonne(self):
        self.h_c = [self.rows]*self.cols
        for x,col in enumerate(self.board):
            y = 0
            while col[y] == 0:
                y += 1
                self.h_c[x] -= 1

    def calcul_some_values(self):
        self.bl = 0
        self.col_trans = 0
        self.rwh = set()
        self.holes = 0
        for x in range(self.cols):
            flag = False
            col = self.board[x]
            for y in range(self.rows-1,self.rows-self.h_c[x]-1,-1):
                if col[y] == 0:
                    flag = True
                    self.holes += 1
                    self.rwh.add(y)
                elif flag:
                    self.bl += 1
                if col[y]^col[y-1]:
                    self.col_trans += 1
            self.col_trans -= 1

    def height(self):
```

```
    return max(self.h_c)

def aggregate_height(self):
    return sum(self.h_c)

def pseudo_holes(self):
    return self.holes

def bumpiness(self):
    return sum([abs(self.h_c[i+1]-self.h_c[i]) for i in range(self.cols-1)])

def blockade(self):
    return self.bl
    '''
    bl = 0
    for x in range(self.cols):
        flag = False
        col = self.board[x]
        for y in range(self.rows-1,self.rows-self.h_c[x]-1,-1):
            if col[y] == 0:
                flag = True
            elif flag:
                bl += 1
    return bl
    '''

def row_transitions(self):
    row_trans = 0
    for row in self.board_nor[:-1]:
        if row.count(0) < len(row):
            for x in range(self.cols-1):
                if row[x]^row[x+1]:
                    row_trans += 1
            if row[0] == 0:
                row_trans += 1
            if row[self.cols-1] == 0:
                row_trans += 1
    return row_trans

def colonne_transitions(self):
    return self.col_trans
    '''
    col_trans = 0
    for col in self.board:
        for i in range(self.rows):
            if col[i] ^ col[i+1]:
                col_trans += 1
        col_trans -= 1
    return col_trans
    '''

def rows_with_holes(self):
    return len(self.rwh)
    '''
    rwh = set()
    for x in range(self.cols):
```



```
        for y in range(self.rows-1,self.rows-self.h_c[x]-1,-1):
            if self.board[x][y] == 0:
                rwh.add(y)
    return len(rwh)
    '''

def score(self):
    score = 0
    self.calcul_some_values()
    score += self.param[0]*self.height()
    score += self.param[1]*self.aggregate_height()
    score += self.param[2]*self.clear_lines
    score += self.param[3]*self.pseudo_holes()
    score += self.param[4]*self.bumpiness()
    score += self.param[5]*self.blockade()
    score += self.param[6]*self.landing_height
    score += self.param[7]*self.row_transitions()
    score += self.param[8]*self.colonne_transitions()
    score += self.param[9]*self.rows_with_holes()
    '''
    score += -abs(self.param[0]*self.height())
    score += -abs(self.param[1]*self.aggregate_height())
    score += abs(self.param[2]*self.clear_lines)
    score += -abs(self.param[3]*self.pseudo_holes())
    score += -abs(self.param[4]*self.bumpiness())
    score += -abs(self.param[5]*self.blockade())
    score += -abs(self.param[6]*self.landing_height)
    score += -abs(self.param[7]*self.row_transitions())
    score += -abs(self.param[8]*self.colonne_transitions())
    score += -abs(self.param[9]*self.rows_with_holes())
    '''
    return score


class scorer_simple(boardscorer):

    def pseudo_holes(self):
        return sum([self.h_c[i]-(self.rows-self.board[i].count(0)) for i in range(self.cols)])

    def score(self):
        score = 0
        score += self.param[0]*self.height()
        score += self.param[1]*self.aggregate_height()
        score += self.param[2]*self.clear_lines
        score += self.param[3]*self.pseudo_holes()
        score += self.param[4]*self.bumpiness()
        score += self.param[5]*self.landing_height
        '''
        score += -abs(self.param[0])*self.height()
        score += -abs(self.param[1])*self.aggregate_height()
        score += abs(self.param[2])*self.clear_lines
        score += -abs(self.param[3])*self.pseudo_holes()
        score += -abs(self.param[4])*self.bumpiness()
        score += -abs(self.param[5])*self.landing_height
        '''
        return score
```

H.4 tetris/tetris_AI/tetris_bot.py

```
from tetris.tetris_game.tetris_game_only import *
from tetris.tetris_AI.tetris_scorer import *

class bot(object):

    def __init__(self,param,taille):
        self.cols, self.rows = taille
        self.move = 0
        self.rotate = 0
        self.param = param

    def testdrop(self,grille0,stone,x):
        for y in range(self.rows+1):
            if grille0.check_collision(stone,(x,y)):
                if y == 0:
                    return False
                landing_h = self.rows-y-len(stone)+1
                grille0.board = join_matrixes(grille0.board,stone,(x,y))
                cleared_rows = []
                for i in range(self.rows):
                    row = grille0.board[i]
                    if 0 not in row:
                        cleared_rows.append(i)
                if cleared_rows != []:
                    has_cleared = grille0.remove_rows(cleared_rows)
                else:
                    has_cleared = []
                break
        def change_back():
            if has_cleared != []:
                grille0.add_rows(has_cleared)
            grille0.board = remove_matrixes(grille0.board,stone,(x,y))
        return grille0,len(cleared_rows),landing_h,change_back

    def scorer(self,board,clear,landing_h):
        sc = boardscorer(self.param,board,clear,landing_h)
        return sc.score()

    # pl est True quand un nouveau stone arrive
    def play(self,grille0,stone,x,y,ne,pl):
        if pl:
            maxscore = float("-inf")
            maxcouple = None
            for r in range(4):
                for c in range(self.cols):
                    res = self.testdrop(grille0,stone,c)
                    if res:
                        newscore = self.scorer(res[0].board,res[1],res[2])
                        if newscore > maxscore:
                            maxscore = newscore
                            maxcouple = (r,c)
                        res[3]()
                stone = rotate_clockwise(stone)
            self.rotate = maxcouple[0]
            self.move = maxcouple[1] - x
        if self.move < 0:
            self.move += 1
            return "LEFT"
        elif self.move >0 and (
             self.rotate == 0 or self.cols-x > max(len(stone[0]),len(stone))):
            self.move -= 1
            return "RIGHT"
        elif self.rotate != 0:
            self.rotate -= 1
            return "UP"
        else:
            return "DOWN"

class bot_simple_scorer(bot):
    def scorer(self,board,clear,landing_h):
        sc = scorer_simple(self.param,board,clear,landing_h)
        return sc.score()


class bot_2_pieces(bot):

    def play(self,grille0,stone,x,y,ne,pl):
        if pl:
            maxscore = float("-inf")
            maxcouple = None
            for r1 in range(4):
                for c1 in range(self.cols):
                    res = self.testdrop(grille0,stone,c1)
                    if res:
                        for r2 in range(4):
                            for c2 in range(self.cols):
                                res2 = self.testdrop(res[0],ne,c2)
                                if res2:
                                    newscore = self.scorer(res2[0].board,res[1]+res2[1],res[2])
                                    if newscore > maxscore:
                                        maxscore = newscore
                                        maxcouple = (r1,c1)
                                    res2[3]()
                                ne = rotate_clockwise(ne)
                            res[3]()
                    stone = rotate_clockwise(stone)
            self.rotate = maxcouple[0]
            self.move = maxcouple[1] - x
        if self.move < 0:
            self.move += 1
            return "LEFT"
        elif self.move >0 and (
             self.rotate == 0 or self.cols-x > max(len(stone[0]),len(stone))):
            self.move -= 1
            return "RIGHT"
        elif self.rotate != 0:
            self.rotate -= 1
            return "UP"
        else:
            return "DOWN"
```

H.5 tetris/tetris_AI/tetris_evalu.py

```
from tetris.tetris_game.tetris_game_only import *
import pygame
import time
from random import randint

class Tetris_evalu(TetrisApp1):

    # 5 actions pour chaque drop
    def evalu(self,bot,param):
        key_actions = {
            'LEFT': lambda:self.move(-1),
            'RIGHT': lambda:self.move(+1),
            'DOWN': self.drop,
             'UP': self.rotate_stone,
             None: lambda:None
        }

        bot0 = bot(param,(self.cols,self.rows))
        self.gameover = False
        self.init_game()
        cmp = 0
        play = True
        t = time.time()
        oldline = 0

        while not self.gameover:
            action = bot0.play(self.grille,self.stone,
                               self.stone_x,self.stone_y,self.next_stone,play)
            play = False
            if key_actions[action]():
                play = True
            if cmp%5 == 0:
                if self.drop():
                    play = True
            cmp += 1
            if self.lines%1000 == 0 and self.lines != oldline:
                print(self.lines,time.time()-t)
                oldline = self.lines
        return cmp,self.lines,time.time()-t


cell_size = 18
maxfps =  30

colors = [
(0, 0, 0 ),
(255, 85, 85),
(100, 200, 115),
(120, 108, 245),
(255, 140, 50 ),
(50, 120, 52 ),
(146, 202, 73 ),
(150, 161, 218 ),
(35, 35, 35)]
```



```
class Tetris_display(TetrisApp1):

    def init_display(self):
        pygame.init()
        self.width = cell_size*(self.cols+6)
        self.height = cell_size*self.rows
        self.rlim = cell_size*self.cols
        self.bground_grid = [[ 8 if x%2==y%2 else 0 for x in range(self.cols)]
                              for y in range(self.rows)]
        self.default_font = pygame.font.Font(pygame.font.get_default_font(), 12)
        self.screen = pygame.display.set_mode((self.width, self.height))
        self.pause = False

    def disp_msg(self, msg, topleft):
        x,y = topleft
        for line in msg.splitlines():
            self.screen.blit(
                self.default_font.render(line,False,(255,255,255),(0,0,0)),(x,y))
            y += 14

    def draw_matrix(self, matrix, offset):
        off_x, off_y = offset
        for y, row in enumerate(matrix):
            for x, val in enumerate(row):
                if val:
                    pygame.draw.rect(
                        self.screen,
                        colors[val],
                        pygame.Rect(
                            (off_x+x)*cell_size,
                            (off_y+y)*cell_size, cell_size, cell_size),0)

def fpause(self):
    self.pause = not self.pause

def start_game(self):
    if self.gameover:
        self.init_game()
        self.gameover = False

def display(self,bot,param):
    key_actions = {
        'LEFT': lambda:self.move(-1),
        'RIGHT': lambda:self.move(+1),
        'DOWN': self.drop,
         'UP': self.rotate_stone,
         None: lambda:None
    }

    self.init_display()
    self.init_game()
    self.gameover = False
    dont_burn_my_cpu = pygame.time.Clock()
    bot0 = bot(param,(self.cols,self.rows))
    cmp = 0
```

```
done = False
play = True

while not done:

    if not self.gameover and not self.pause:
        self.screen.fill((0,0,0))
        pygame.draw.line(self.screen,
            (255,255,255),(self.rlim+1, 0),(self.rlim+1, self.height-1))
        self.disp_msg("Next:", (self.rlim+cell_size,2))
        self.disp_msg("Lines: %d" % self.lines,(self.rlim+cell_size, cell_size*5))
        self.draw_matrix(self.bground_grid, (0,0))
        self.draw_matrix(self.grille.board, (0,0))
        self.draw_matrix(self.stone, (self.stone_x, self.stone_y))
        self.draw_matrix(self.next_stone, (self.cols+1,2))
        pygame.display.update()

        action = bot0.play(self.grille,self.stone,
                        self.stone_x,self.stone_y,self.next_stone,play)
        play = False
        if key_actions[action]():
            play = True
        if cmp%5 == 0:
            if self.drop():
                play = True
        cmp += 1

    for event in pygame.event.get():
            if event.type == pygame.QUIT:
                done = True
            elif event.type == pygame.KEYDOWN:
                if event.key == eval("pygame.K_SPACE"):
                    self.start_game()
                if event.key == eval("pygame.K_p"):
                    self.fpause()
        dont_burn_my_cpu.tick(maxfps)

    pygame.display.quit()



class Tetris_evalu_z(TetrisApp_z,Tetris_evalu):
    pass

class Tetris_display_z(TetrisApp_z,Tetris_display):
    pass

class Tetris_display_z_ordre(TetrisApp_z_ordre,Tetris_display):
    pass

class Tetris_evalu_dans_lordre(TetrisApp_dans_lordre,Tetris_evalu):
    pass

class Tetris_display_dans_lordre(TetrisApp_dans_lordre,Tetris_display):
    pass
```


H.6 tetris/tetris_training/tetris_training_functions.py

```
from pybrain.optimization.populationbased.ga import GA,GA_new
from differential_evolution.DE_rand_1_bin import DE_adapte
from bat_algorithms import *
from tetris.tetris_AI import *
from numpy import mean
import time

# training z, bot_simple, 15 fois puis moyenner, revoie cmp
def training1(cols,rows):
    def evaluz_s(param):
        res = [Tetris_evalu_z(cols,rows).evalu(bot_simple_scorer,param)[0] for _ in range(15)]
        return mean(res)
    return evaluz_s

# training normal, bot_simple
def training1_n(cols,rows):
    def evalu_s(param):
        res = [Tetris_evalu(cols,rows).evalu(bot_simple_scorer,param)[0] for _ in range(15)]
        return mean(res)
    return evalu_s


# training z, bot
def training2(cols,rows):
    def evaluz_s(param):
        res = [Tetris_evalu_z(cols,rows).evalu(bot,param)[0] for _ in range(15)]
        return mean(res)
    return evaluz_s

# training normal, bot
def training2_n(cols,rows):
    def evalu_s(param):
        res = [Tetris_evalu(cols,rows).evalu(bot,param)[0] for _ in range(8)]
        return mean(res)
    return evalu_s


'''
Pour obtenir quelques résultats
'''

'''
GAs = GA_new(training1(10,10), numParameters= 6, populationSize= 20,
            mutationProb=0.2, elitism= True, _eliteSize= 1)

for i in range(31):
    t0 = time.time()
    print(i)
    print(GAs.learn(0))
    i = max(range(20), key= lambda j:GAs.fitnesses[j])
    print((GAs.currentpop[i],GAs.fitnesses[i]))
    print(time.time()-t0)
'''
```

```
'''
GA1 = GA_new(training2(10,10), numParameters= 10, populationSize= 20,
          mutationProb=0.2, elitism= True, _eliteSize= 1)

for i in range(31):
    t0 = time.time()
    print(i)
    print(GA1.learn(0))
    i = max(range(20), key= lambda j:GA1.fitnesses[j])
    print((GA1.currentpop[i],GA1.fitnesses[i]))
    print(time.time()-t0)
'''

'''
DEs = DE_adapte(20,10,training2(10,10))

for i in range(31):
    t0 = time.time()
    print(i)
    print(DEs.optimize(1))
    print(time.time()-t0)
'''

'''
DE1 = DE_adapte(20,6,training1(10,10))

for i in range(31):
    t0 = time.time()
    print(i)
    print(DE1.optimize(1))
    print(time.time()-t0)
'''
```