TP4 : NEAT

SINADINOVIC Marko

Semestre 6 : 2024-2025

Contents

1 Motivation

Dans le cadre du cours de Recherche Opérationnelle (Machine Learning), j'ai été amené à réaliser le TP numéro 4 sur la méthode NEAT (*NeuroEvolution of Augmenting Topologies*). Cette approche ne se limite pas à l'optimisation des poids et des biais d'un réseau de neurones, mais vise à adapter de manière dynamique sa topologie pour répondre efficacement à des problématiques qui peuvent être complexes. J'ai pris l'initiative de documenter mes travaux afin de consolider ma compréhension des concepts abordés.

2 Introduction

Le but de ce TP est d'étudier une technique d'évolution neuroévolutionnaire capable non seulement d'ajuster les poids et les biais des réseaux de neurones, mais également de modifier leur topologie. Pour ce faire, différentes étapes sont nécessaires : l'installation du module NEAT ainsi que des scripts associés (`evolve-feedforward.py` et `visualize.py`), suivie de nombreux tests visant à analyser les performances obtenues. Des paramétrages, tels que la modification du *compatibility threshold* ou de la taille minimale des espèces (*min species size*), permettent d'examiner l'impact des ajustements sur l'efficacité de l'algorithme, notamment en termes de convergence. Enfin, j'ai réalisé une étude sur les résultats graphiques obtenus ainsi que sur l'adaptation des solutions proposées à des cas plus complexes, comme par exemple la résolution du problème `A XOR B XOR C` en remplacement de `A XOR B`.

3 Exercice 1 - XOR

3.1 Question 1

En lançant le script `evolve-feedforward.py` une première fois, nous obtenons trois illustrations graphiques.

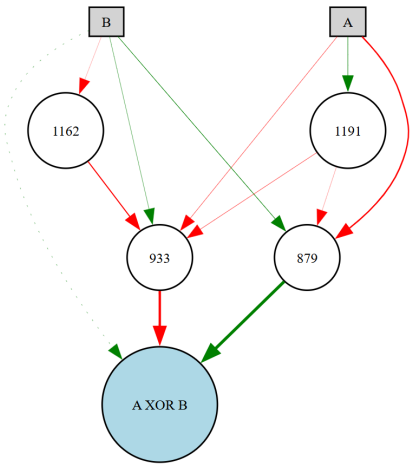


Figure 1: Diagramme de structure de réseau évolutif

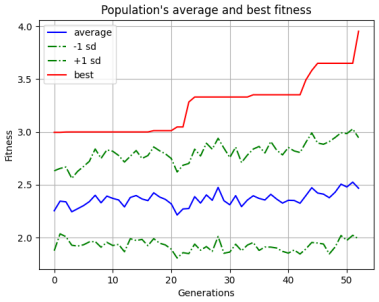


Figure 2: Convergence

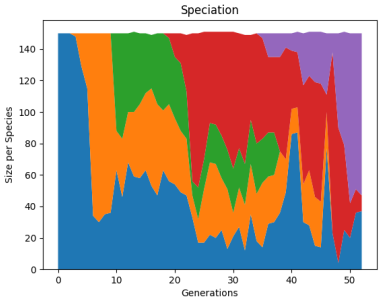


Figure 3: Taille des différentes espèces au fil des générations

La `Figure 1` est un diagramme de structure de réseau évolutif. Nous remarquons qu'il y a plusieurs nœuds/neurones (les cercles) : les neurones d'entrée, comme `A` et `B`, des neurones cachés entre l'entrée et la sortie, et enfin, un neurone de sortie, `A XOR B`. Entre chaque nœud, il y a des flèches qui représentent les connexions entre ces derniers ; en réalité, ce sont les poids entre chaque nœud. En somme, dans notre TP, ce graphe montre comment le réseau neuronal a été optimisé pour résoudre le problème XOR.

La `Figure 2` représente un graphique de l'évolution de la performance du réseau neuronal au fil des générations. Sur l'axe horizontal, les générations successives, et sur l'axe vertical, les valeurs de fitness, la mesure de la performance. Un fitness élevé indique que le réseau neuronal est proche d'une solution optimale, alors qu'un fitness faible montre qu'il existe des erreurs importantes entre les sorties générées et les sorties attendues. Sur notre graphe, trois courbes sont visibles :

- Une courbe bleue montrant la fitness moyenne de la population à chaque génération. Elle reflète les performances globales des réseaux neuronaux au fil du temps. Notre moyenne est entre `2.25 et 2.52`, ce qui signifie que la performance moyenne de notre réseau de neurones est relativement faible au cours de l'évolution.

- Une courbe rouge représentant la meilleure fitness atteinte à chaque génération. Elle indique l'évolution de la solution optimale au problème XOR. Notre fitness débute à 3 et se termine vers 3.8 à la 50ème génération.

- Des bandes vertes pointillées illustrant l'écart-type, soit la dispersion des performances autour de la moyenne. Elles permettent d'observer la variabilité des résultats dans la population.

La `Figure 3` représente la taille des différentes espèces au fil des générations. Sur l'axe horizontal, nous avons les générations, allant de 0 à 50, et sur l'axe vertical, la taille par espèce avec des valeurs comprises entre 0 et 150. Différentes courbes de couleurs distinctes illustrent l'évolution de la population pour chaque espèce. Ce graphique met en évidence les variations de population au cours de l'évolution ; certaines espèces restent présentes en nombre, tandis que d'autres s'éteignent.

Si nous relançons le script comme demandé dans la question, nous obtenons les illustrations suivantes :

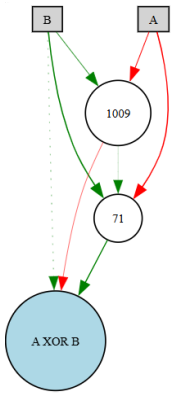


Figure 4: Diagramme de structure de réseau évolutif

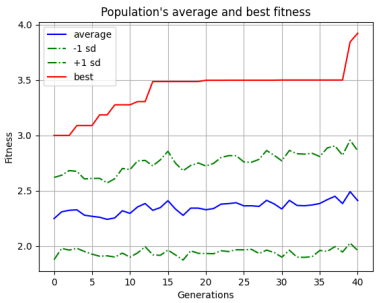


Figure 5: Convergence

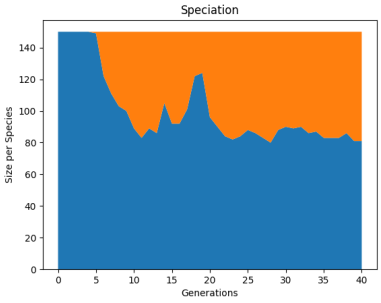


Figure 6: Taille des différentes espèces au fil des générations

Si nous relançons notre script `evolve-feedforward.py` plusieurs fois, par exemple à la 7ème fois, nous obtenons les illustrations suivantes :

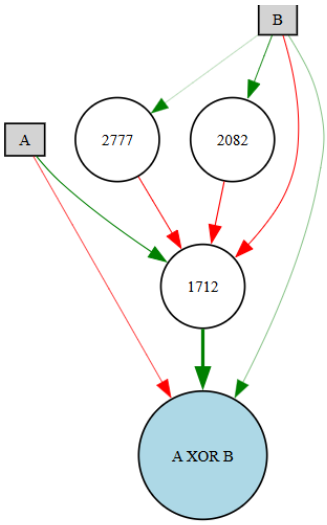


Figure 7: Diagramme de structure de réseau évolutif

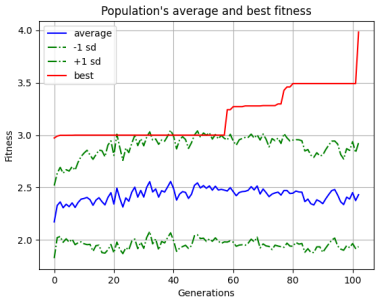


Figure 8: Convergence

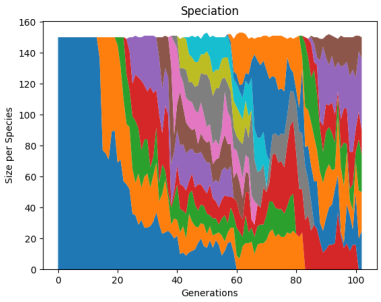


Figure 9: Taille des différentes espèces au fil des générations

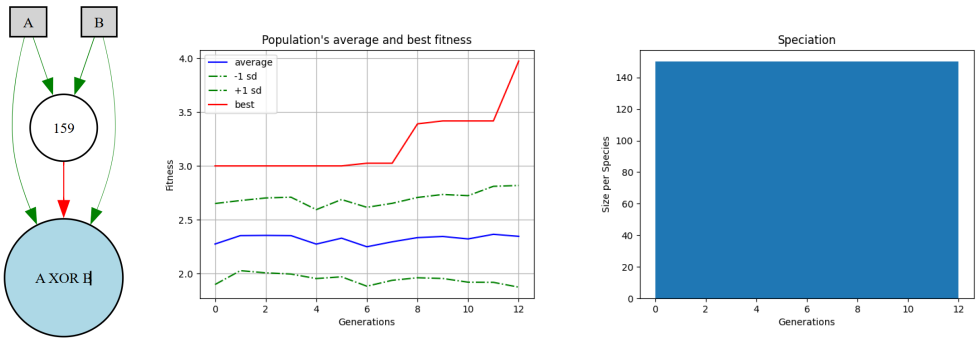
Nous remarquons qu'en relançant le script plusieurs fois, les résultats (les illustrations) sont différents. La variabilité des résultats est due à la génération aléatoire des génomes initiaux,

ce qui impacte considérablement "l'évolution", et à la mutation aléatoire appliquée au cours de notre processus. Nous verrons par la suite que les paramètres dans notre fichier de configuration `config-feedforward` jouent aussi un rôle crucial dans la variabilité des résultats et de la convergence du fitness.

3.2 Question 2

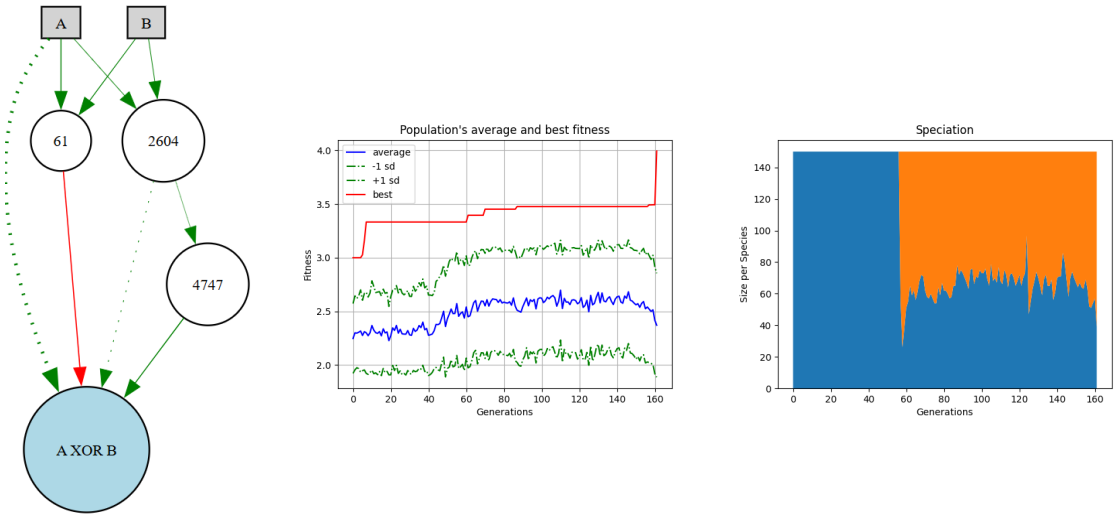
3.2.1 DefaultSpeciesSet

En augmentant la valeur de `compatibility-threshold` de `3` à `9`, les individus des espèces possèdent des génomes plus diversifiés, ce qui réduit le nombre d'espèces dans la population, et nous obtenons les illustrations suivantes :

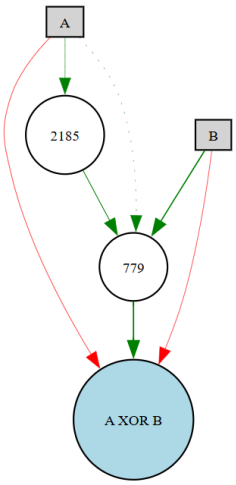


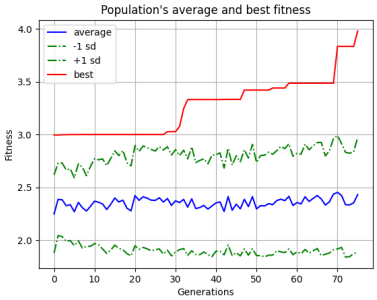
Nous remarquons qu'on obtient une espèce qui contient tous les individus avec des génomes différents. L'évolution est déséquilibrée et on ne peut pas exploiter pleinement les données. En ce qui concerne la convergence, elle est très rapide, mais comme on étudie un petit nombre de générations, pour des problèmes plus complexes cela peut poser des problèmes.

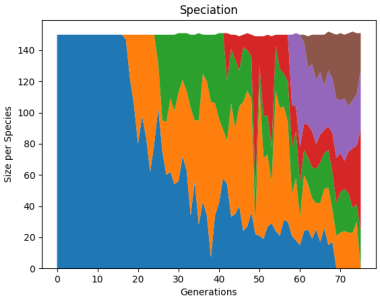
Alors qu'en augmentant la valeur de `compatibility-threshold` de `3` à `4`, nous obtenons les illustrations suivantes :



Si nous relançons notre programme avec la même valeur `compatibility-threshold = 4.0` :

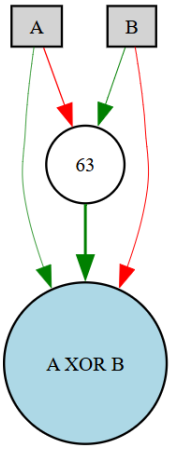


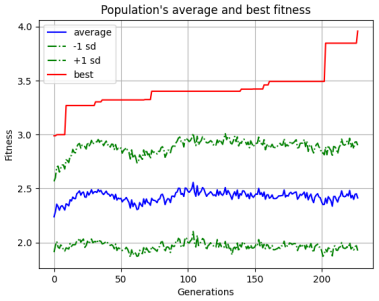


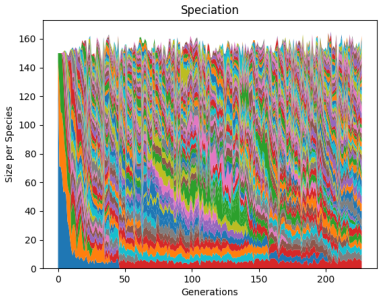


Si nous augmentons de manière modérée notre paramètre de 3 à 4, nous remarquons des stagnations à cause des différences entre les individus, mais aussi une convergence rapide à partir de la 30ème génération en moyenne. L'algorithme parvient à résoudre le problème XOR en peu de générations.

Si nous diminuons notre paramètre de 3 à 2, nous obtenons les illustrations suivantes :







Nous obtenons un très grand nombre d'espèces, car les individus de ces dernières ont des génomes moins diversifiés, ce qui augmente le nombre d'espèces dans la population. En ce qui concerne la convergence, nous remarquons des stagnations sur plusieurs dizaines de générations, donc une convergence très lente de la première à la 200ème génération, et une convergence rapide après la 201ème génération.

En somme, trop augmenter la valeur peut biaiser les résultats et la convergence pour des problèmes complexes. Si on l'augmente de manière modérée, nous pouvons bien exploiter les données et la convergence, et ainsi résoudre le problème XOR. Cependant, si nous diminuons notre paramètre, la convergence devient très lente et prend plusieurs centaines de générations pour converger.

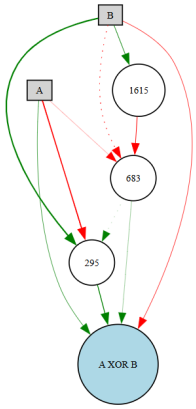
3.2.2 DefaultReproduction

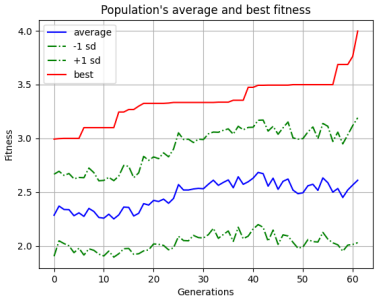
En ajoutant à notre fichier `config-feedforward` la ligne 4 suivante :

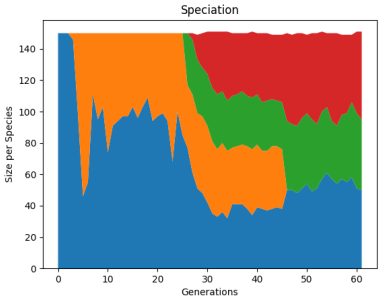
```
1 #DefaultReproduction
2 elitism             = 2
3 survival_threshold  = 0.2
4 min_species_size    = x
```

Listing 1: Notre modification

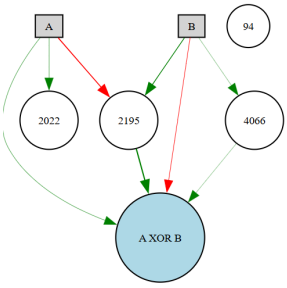
Nous influons sur le comportement des espèces après reproduction, ce qui influe sur nos résultats. Étudions cela de plus près en utilisant plusieurs valeurs pour ce paramètre. Nous obtenons les illustrations suivantes pour `x = 2` :

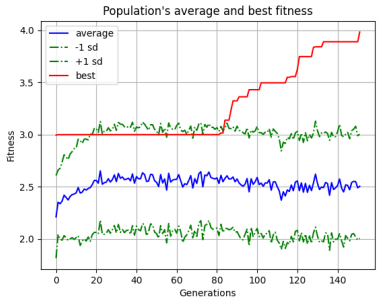


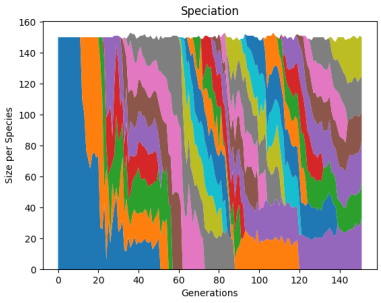




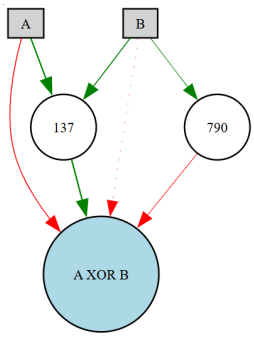
Nous obtenons les illustrations suivantes pour `x = 6`

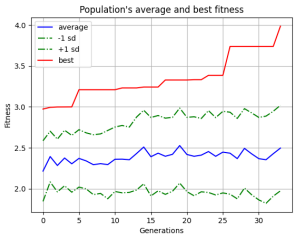


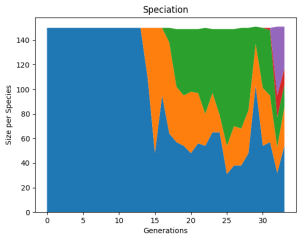




Nous obtenons les illustrations suivantes pour `x = 8`

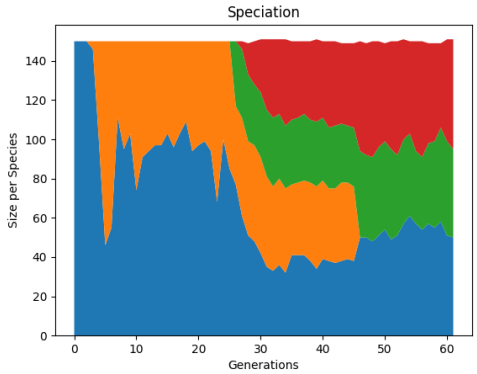




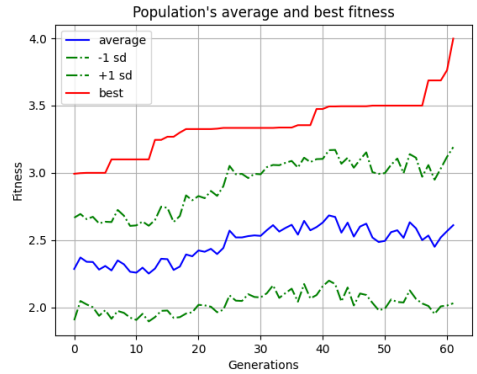


En analysant ces graphiques, nous remarquons que lorsque notre paramètre est assez faible, par exemple 2, le nombre d'itérations avant convergence est plus important comparé à lorsqu'il est élevé. En effet, les espèces doivent contenir plus d'individus pour éviter l'extinction, ce qui a pour effet d'accélérer la convergence.

En ce qui concerne l'extinction d'une espèce après une stagnation trop importante, nous pouvons nous référer aux illustrations. Nous avons obtenu ce graphique pour x = 2 :



Nous remarquons qu'après la 44ème génération, l'espèce orange disparaît ; elle n'apparaît plus, ce qui coïncide avec la stagnation entre la 17ème et la 38ème générations constatée sur le graphique suivant :



Enfin, le nombre d'individus par espèce reste toujours supérieur ou égal à notre paramètre tant qu'une espèce n'a pas dépassé sa stagnation. En effet, si nous étudions le fichier de configuration `config-feedforward`, nous remarquons les lignes suivantes :

```
#DefaultStagnation
species_fitness_func = max
max_stagnation       = 20
species_elitism      = 2
```

Listing 2: Paramètres de stagnation

Tant qu'une espèce ne dépasse pas le seuil maximal de stagnation défini à la ligne 3 de `Listing 3`, l'algorithme lui alloue le nombre minimal d'individus, qui est notre paramètre :

```
min_species_size    = 2
```

Si l'espèce dépasse la limite de stagnation et qu'elle n'a pas le nombre d'individus minimal requis, elle s'éteint.

3.3 Question 3

En définissant nos paramètres de la manière suivante :

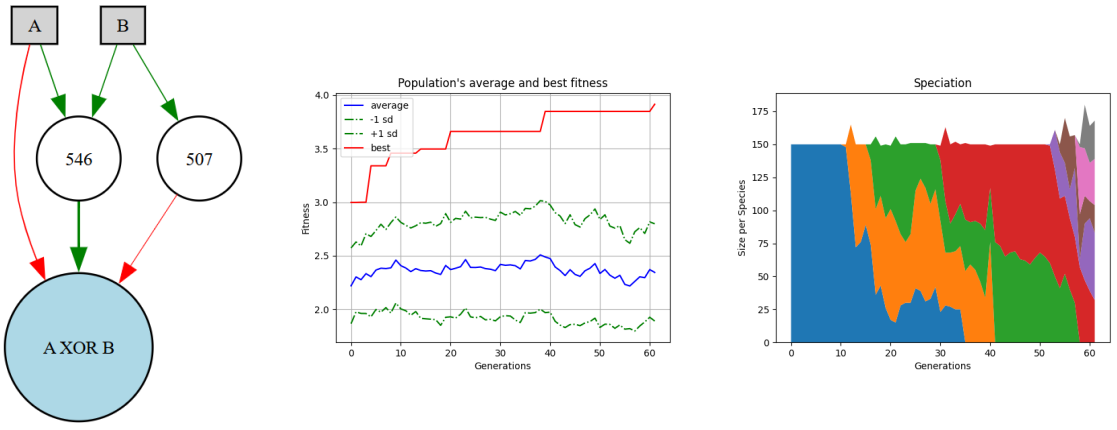
```
#DefaultReproduction
elitism            = 2
survival_threshold = 0.2
min_species_size   = 32
```

Listing 3: Paramètre de reproduction

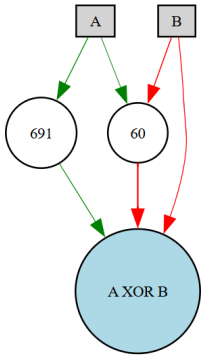
```
#DefaultStagnation
species_fitness_func = max
max_stagnation       = 20
species_elitism      = 2
```

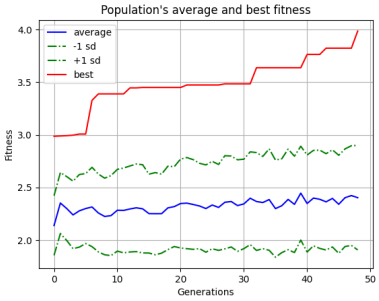
Listing 4: Paramètres de stagnation

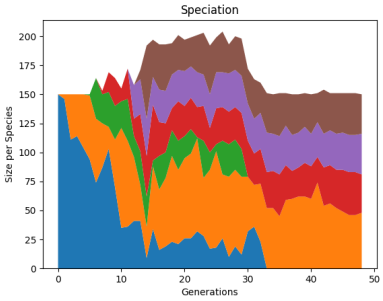
Nous obtenons les graphiques suivant :



Ou bien :



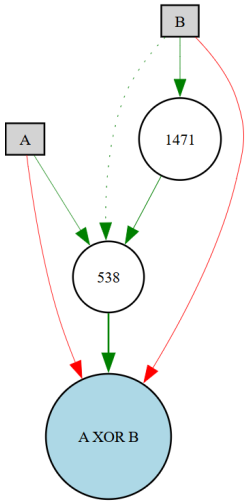


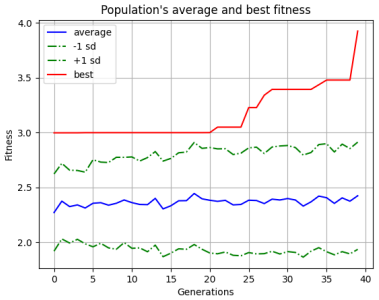


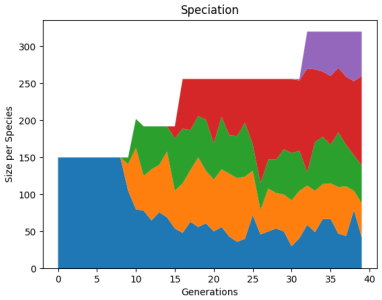
En définissant nos paramètres de la manière suivante :

```
#DefaultReproduction
elitism             = 2
survival_threshold  = 0.2
min_species_size    = 64
```

Listing 5: Paramètre de reproduction







Nous remarquons qu'il existe de nombreuses stagnations sur les graphiques de convergence. Si nous étudions de plus près la convergence, nous voyons clairement que la courbe du meilleur fitness atteint la valeur maximale de 4, mais cette convergence est très lente avec beaucoup de stagnations. Cette tendance est due à la réduction de la diversité génétique à cause de notre valeur élevée de notre paramètre. On alloue plus de ressources à nos espèces les plus présentes en termes d'individus, en omettant les autres espèces (peut-être prometteuses), réduisant ainsi les solutions. En somme, même si le graphe de l'individu le plus adapté est proche de la valeur optimale, il peut entraîner des performances sous-optimales pour des problèmes complexes. Dans notre problème comme le XOR, nous pouvons nous satisfaire de cette convergence, mais en optimisant notre valeur (en choisissant une valeur plus modérée comme dans les questions précédentes), on peut obtenir une "meilleure" convergence.

3.4 Question 4

Pour adapter notre script d'optimisation à résoudre le problème `A XOR B XOR C`, dans un premier temps, nous allons modifier les variables `xor inputs` et `xor outputs` en augmentant le nombre d'inputs à 3 de la manière suivante :

```
xor_inputs = [
    (0.0, 0.0, 0.0), (0.0, 0.0, 1.0),
    (0.0, 1.0, 0.0), (0.0, 1.0, 1.0),
    (1.0, 0.0, 0.0), (1.0, 0.0, 1.0),
    (1.0, 1.0, 0.0), (1.0, 1.0, 1.0)]


xor_outputs = [
    (0.0,), (1.0,),
    (1.0,), (0.0,),
    (1.0,), (0.0,),
    (0.0,), (1.0,)]
```

Listing 6: Modification inputs et outputs

Ensuite, nous allons adapter la fonction `eval genomes` à nos entrées :

```
def eval_genomes(genomes, config):
    for genome_id, genome in genomes:
        genome.fitness = len(xor_inputs) #Notre modification
        net = neat.nn.FeedForwardNetwork.create(genome, config)
        for xi, xo in zip(xor_inputs, xor_outputs):
            output = net.activate(xi)
            genome.fitness -= (output[0] - xo[0]) ** 2
```

Listing 7: Modification eval genomes

Nous incluons le troisième input `C` et modifions notre output de la manière suivante pour visualiser les résultats :

```
    node_names = {-1: 'A', -2: 'B', -3: 'C', 0: 'A XOR B XOR C'}
    visualize.draw_net(config, winner, True, node_names=node_names)
    visualize.plot_stats(stats, ylog=False, view=True)
    visualize.plot_species(stats, view=True)
```

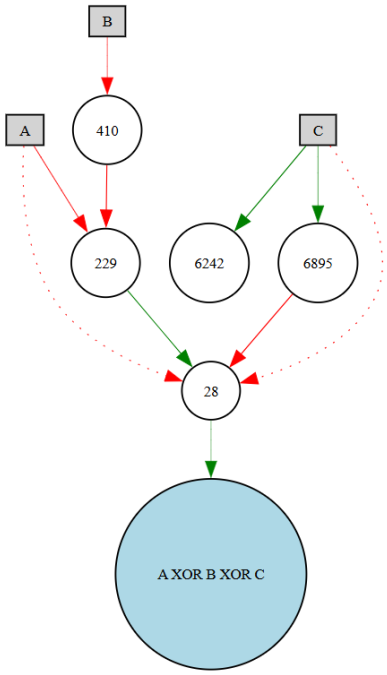
Listing 8: Modification visualisation des sorties

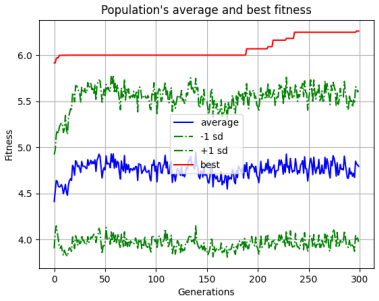
Enfin, nous modifions notre fichier de configuration pour que notre réseau prenne en charge 3 inputs et nous augmentons le `fitness-threshold` :

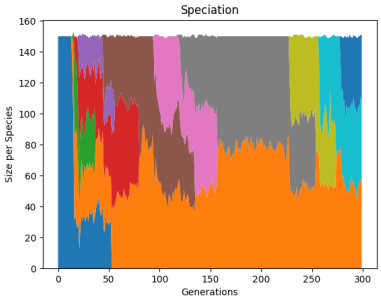
```
#NEAT
fitness_criterion     = max
fitness_threshold     = 7

# network parameters
num_hidden              = 0
num_inputs              = 3
num_outputs             = 1
```

Listing 9: Modification fichier de configuration

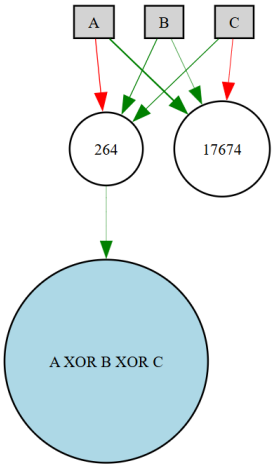
Maintenant que nous avons modifié le script d'optimisation de sorte à résoudre le problème `A XOR B XOR C`, en le lançant, nous obtenons les graphes suivants :

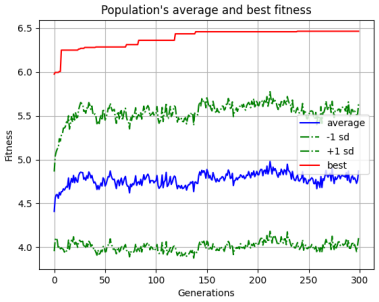


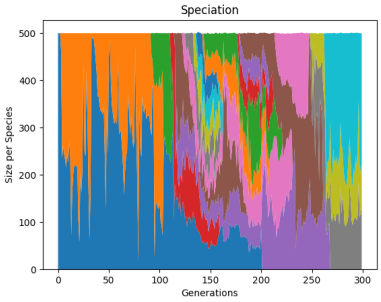




Si nous modifions le fichier de configuration en augmentant le nombre d'individus dans chaque espèce et en ajoutant des fonctions d'activation comme `tanh` :







Enfin, si nous modifions notre fichier de configuration en mettant 2 couches cachées :

```
# network parameters
num_hidden              = 2
num_inputs              = 3
num_outputs             = 1
```

Listing 10: Modification fichier de configuration

Nous obtenons ces graphes :