TP2 - KnapSac

November 9, 2023

1 Questions

1. Faites varier le nombre d'objets, leurs poids, la capacité maximale du sac et le nombre de générations. Que constatez-vous ? Expliquez les causes.
2. Apporter les modifications nécessaires pour aboutir seulement à des solutions faisables, si elles existent. Voici les approches qu'il faut coder et comparer :

- Mettre une fitness négative proportionnelle au dépassement de la capacité du sac
- Vérifier lors de la génération des solutions, du croisement et de la mutation que ces solutions respectent la capacité du sac. Pour comparer les deux approches, générer 4 instances du problème et stocker les dans des fichiers csv. Comparer les deux approches en termes de résultats et de temps de calcul.

3. Nous souhaitons utiliser cet algorithme à la gestion du portefeuille. Pour des raisons de minimisation du risque, nous considérons que le client peut acheter plusieurs actions du même titre dans la mesure où il respecte la limite des 20% du budget total. L'achat de plusieurs actions ne concerne qu'un seul titre. A la suite des estimations des gains escomptés de chaque titre, il souhaite augmenter ses gains en respectant les limites de son budget et la contrainte de minimisation de risque. Adapter le programme à la gestion du portefeuille.
4. Choisir 4 instances des données du problème et stocker les dans des fichiers csv. Améliorer les performances de l'algorithme génétique sur ces quatre instances. Décrivez dans le rapport les améliorations réalisées et les motivations.

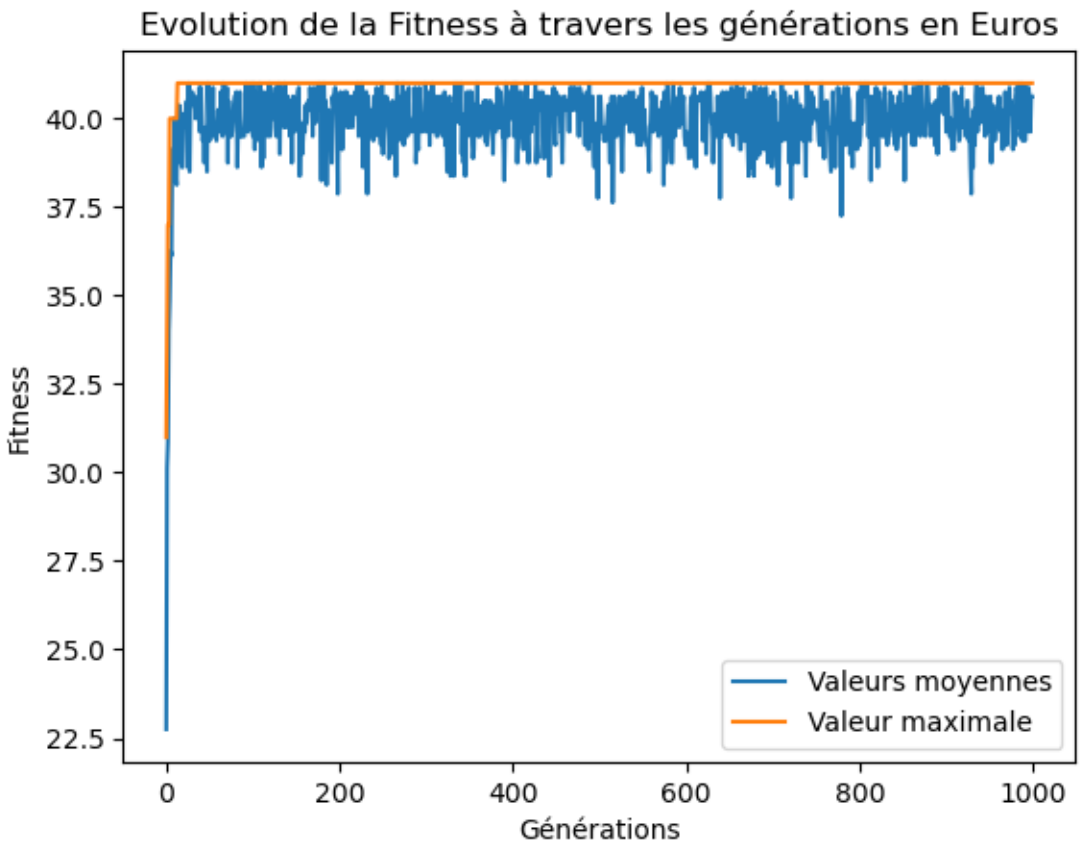
2 Faites varier le nombre d'objets, leurs poids, la capacité maximale du sac et le nombre de générations. Que constatez-vous ? Expliquez les causes.

2.1 Variations du nombre d'objets

[1]:
```
import Initiale_KnapSac as iks

# Données du problème générées aléatoirement
nombre_objets = 10  # Le nombre d'objets
capacite_max = 30  # La capacité du sac
poids_min = 1  # Le poids minimal d'un objet
poids_max = 5
valeur_min = 1  # La valeur minimale d'un objet
valeur_max = 10  # La valeur maximale d'un objet
```

```
# paramètres de l'algorithme génétique
nbr_generations = 1000  # nombre de générations
solutions_par_pop = 8

ID_objets, population_initiale, pop_size = iks.
 ↪fct_population_initiale(solutions_par_pop, nombre_objets)

iks.affichage(nbr_generations, capacite_max, nombre_objets, poids_min,␣
 ↪poids_max, valeur_min, valeur_max,
              ID_objets, population_initiale, pop_size)
```

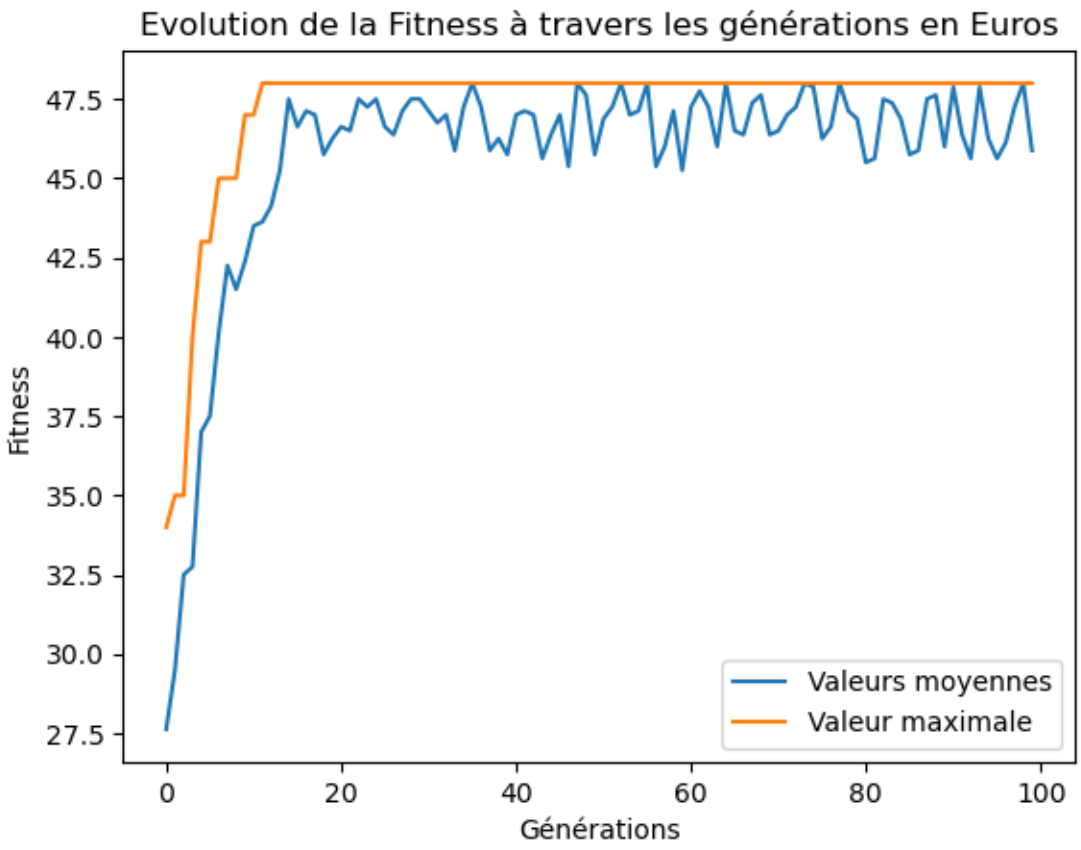
```
Taille de la population: (8, 10)

Voici la dernière génération de la population:
[[1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 0 1 1 1 1 1 1 1]
 [1 1 1 0 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]]

Fitness de la dernière génération:
[41 41 41 41 41 33 38 41]

La solution optimale est:
[array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])]
objets n° [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
(1000, 8)
Avec une valeur de 41 € et un poids de 27 kg
Les objets qui maximisent la valeur contenue dans le sac sans le déchirer :
0
1
2
3
4
5
6
7
8
9
```



[2]:
```
# Données du problème générées aléatoirement
nombre_objets = 10  # Le nombre d'objets
capacite_max = 30  # La capacité du sac
poids_min = 1  # Le poids minimal d'un objet
poids_max = 5
valeur_min = 1  # La valeur minimale d'un objet
valeur_max = 10  # La valeur maximale d'un objet

# paramètres de l'algorithme génétique
nbr_generations = 100  # nombre de générations
solutions_par_pop = 8

ID_objets, population_initiale, pop_size = iks.
 ↪fct_population_initiale(solutions_par_pop, nombre_objets)

iks.affichage(nbr_generations, capacite_max, nombre_objets, poids_min,␣
 ↪poids_max, valeur_min, valeur_max,
              ID_objets, population_initiale, pop_size)
```

```
Taille de la population: (8, 10)
```

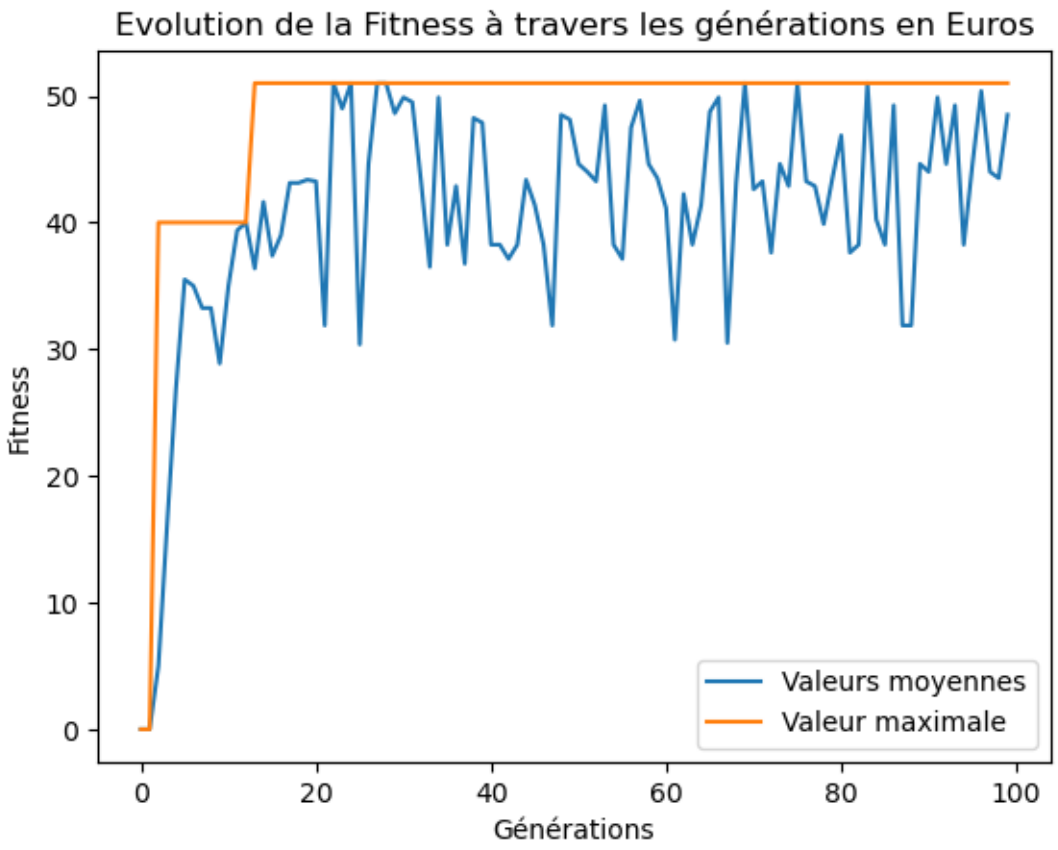
```
Voici la dernière génération de la population:
[[1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 0 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 0 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]]

Fitness de la dernière génération:
[48 48 48 48 47 48 44 48]

La solution optimale est:
[array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])]
objets n° [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
(100, 8)
Avec une valeur de 48 € et un poids de 29 kg
Les objets qui maximisent la valeur contenue dans le sac sans le déchirer :
0
1
2
3
4
5
6
7
8
9
```



[3]:
```
# Données du problème générées aléatoirement
nombre_objets = 10  # Le nombre d'objets
capacite_max = 35  # La capacité du sac
poids_min = 2  # Le poids minimal d'un objet
poids_max = 15
valeur_min = 5  # La valeur minimale d'un objet
valeur_max = 15  # La valeur maximale d'un objet

# paramètres de l'algorithme génétique
nbr_generations = 100  # nombre de générations
solutions_par_pop = 8

ID_objets, population_initiale, pop_size = iks.
 ↪fct_population_initiale(solutions_par_pop, nombre_objets)

iks.affichage(nbr_generations, capacite_max, nombre_objets, poids_min,␣
 ↪poids_max, valeur_min, valeur_max,
              ID_objets, population_initiale, pop_size)
```

```
Taille de la population: (8, 10)
```

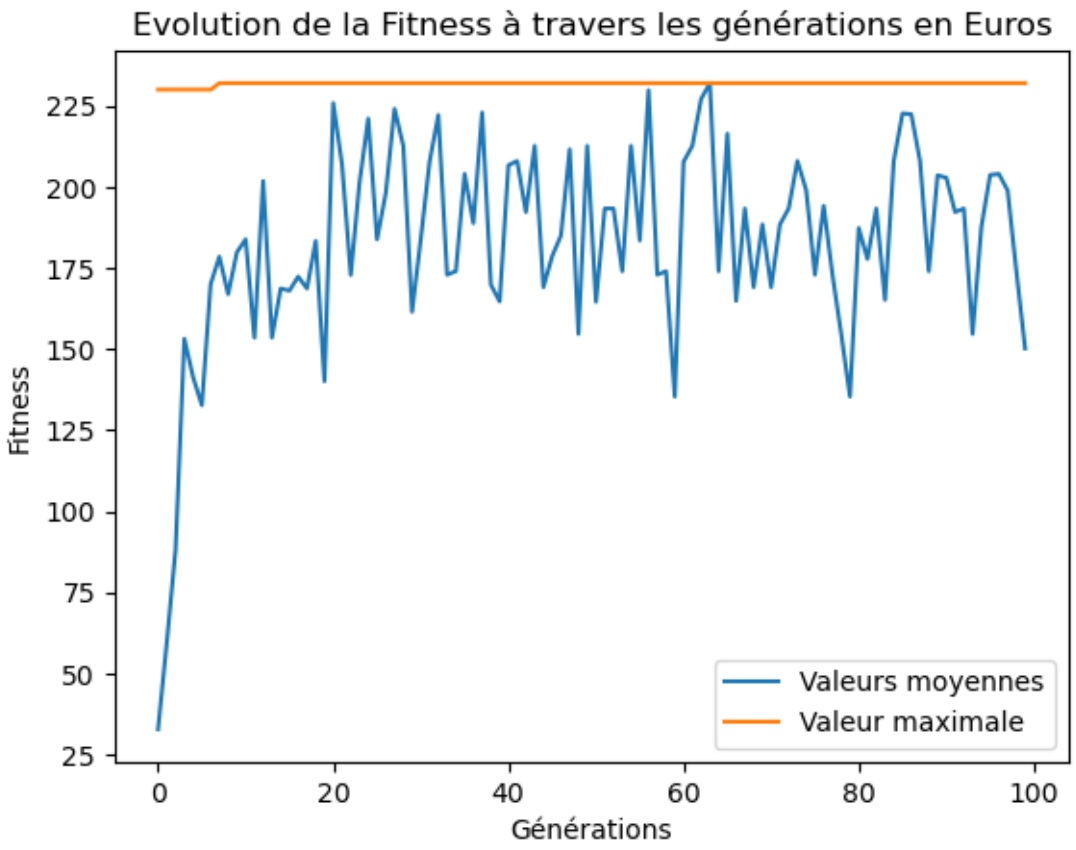
```
Voici la dernière génération de la population:
[[1 0 1 0 1 0 0 0 1 1]
 [1 0 1 0 1 0 0 0 1 1]
 [1 0 1 0 1 0 0 0 1 1]
 [1 0 1 0 1 0 0 0 1 1]
 [1 0 1 0 1 0 0 0 1 1]
 [1 0 1 0 1 0 0 0 1 1]
 [1 1 1 0 1 0 0 0 1 1]
 [1 0 0 0 1 0 0 0 1 1]]

Fitness de la dernière génération:
[51 51 51 51 51 51  0 39]

La solution optimale est:
[array([1, 0, 1, 0, 1, 0, 0, 0, 1, 1])]
objets n° [0, 2, 4, 8, 9]
(100, 8)
Avec une valeur de 51 € et un poids de 35 kg
Les objets qui maximisent la valeur contenue dans le sac sans le déchirer :
0
2
4
8
9
```



[4]:
```
# Données du problème générées aléatoirement
nombre_objets = 12  # Le nombre d'objets
capacite_max = 60  # La capacité du sac
poids_min = 10  # Le poids minimal d'un objet
poids_max = 25
valeur_min = 5  # La valeur minimale d'un objet
valeur_max = 60  # La valeur maximale d'un objet

# paramètres de l'algorithme génétique
nbr_generations = 100  # nombre de générations
solutions_par_pop = 12

ID_objets, population_initiale, pop_size = iks.
 ↪fct_population_initiale(solutions_par_pop, nombre_objets)

iks.affichage(nbr_generations, capacite_max, nombre_objets, poids_min,␣
 ↪poids_max, valeur_min, valeur_max,
              ID_objets, population_initiale, pop_size)
```

```
Taille de la population: (12, 12)
```

```
Voici la dernière génération de la population:
[[1 1 0 0 1 1 0 0 0 1 0 0]
 [1 1 0 0 1 1 0 0 0 1 0 0]
 [1 1 0 0 1 1 0 0 0 1 0 0]
 [1 1 0 0 1 1 0 0 0 1 0 0]
 [1 1 0 0 1 1 0 0 0 1 0 0]
 [1 1 0 0 1 1 0 0 0 1 0 0]
 [1 1 0 0 1 1 0 0 0 1 0 0]
 [1 1 0 0 1 1 0 0 0 1 0 0]
 [0 1 0 0 1 1 0 0 0 1 0 0]
 [1 1 0 0 1 1 1 0 0 1 0 0]
 [1 1 0 0 1 1 0 0 0 0 0 0]
 [1 1 0 0 1 1 0 0 0 1 0 0]]

Fitness de la dernière génération:
[232 232 232 232 232 232 232 232 182   0 173 232]

La solution optimale est:
[array([1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0])]
objets n° [0, 1, 4, 5, 9]
(100, 12)
Avec une valeur de 232 € et un poids de 57 kg
Les objets qui maximisent la valeur contenue dans le sac sans le déchirer :
0
1
4
5
9
```

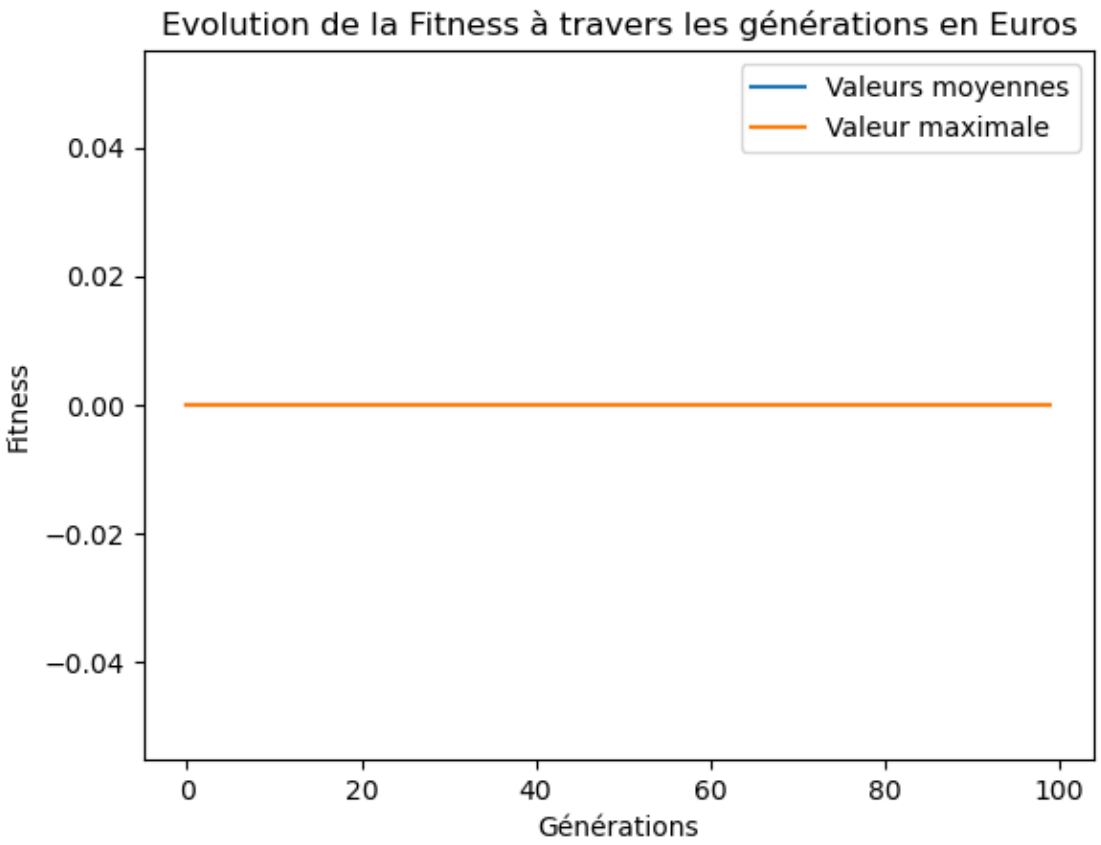


[5]:
```
# Données du problème générées aléatoirement
nombre_objets = 120  # Le nombre d'objets
capacite_max = 60  # La capacité du sac
poids_min = 10  # Le poids minimal d'un objet
poids_max = 25
valeur_min = 50  # La valeur minimale d'un objet
valeur_max = 60  # La valeur maximale d'un objet

# paramètres de l'algorithme génétique
nbr_generations = 100  # nombre de générations
solutions_par_pop = 12

ID_objets, population_initiale, pop_size = iks.
 ↪fct_population_initiale(solutions_par_pop, nombre_objets)

iks.affichage(nbr_generations, capacite_max, nombre_objets, poids_min,␣
 ↪poids_max, valeur_min, valeur_max,
              ID_objets, population_initiale, pop_size)
```

```
Taille de la population: (12, 120)
```

```
Voici la dernière génération de la population:
[[0 0 0 … 0 1 0]
 [0 0 1 … 0 1 1]
 [0 1 0 … 1 0 0]
 …
 [0 0 1 … 1 0 0]
 [1 0 0 … 0 0 1]
 [1 1 0 … 0 1 0]]

Fitness de la dernière génération:
[0 0 0 0 0 0 0 0 0 0 0 0]

La solution optimale est:
[array([0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
       1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0,
       1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0,
       1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0,
       1, 0, 0, 0, 1, 0, 1, 0, 1, 0])]
objets n° [3, 4, 5, 7, 8, 9, 13, 15, 17, 19, 21, 22, 24, 26, 27, 28, 29, 31, 32,
33, 36, 37, 38, 40, 50, 51, 52, 55, 57, 59, 62, 63, 64, 66, 67, 69, 72, 74, 75,
77, 79, 80, 83, 84, 85, 86, 88, 95, 97, 105, 108, 110, 114, 116, 118]
(100, 12)
Avec une valeur de 0 € et un poids de 991 kg
Les objets qui maximisent la valeur contenue dans le sac sans le déchirer :
3
4
5
7
8
9
13
15
17
19
21
22
24
26
27
28
29
31
32
33
36
37
```

38
40
50
51
52
55
57
59
62
63
64
66
67
69
72
74
75
77
79
80
83
84
85
86
88
95
97
105
108
110
114
116
118



2.1.1 Réponse :

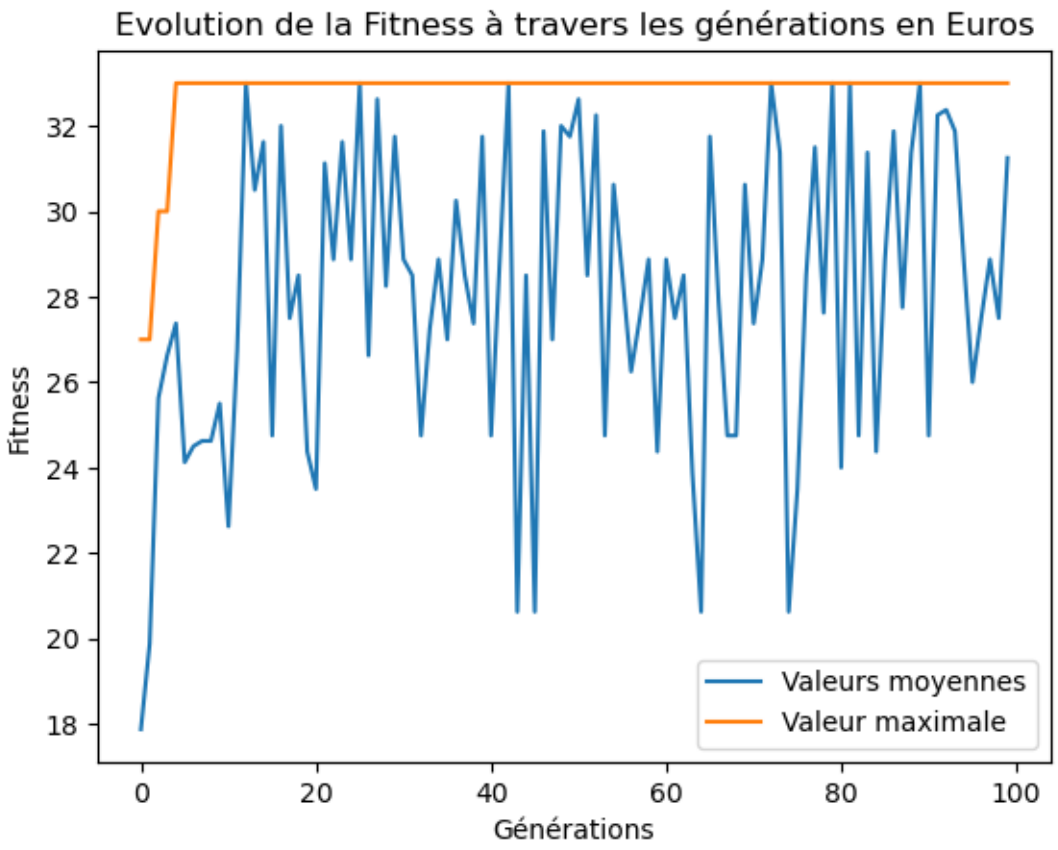
Nous pouvons observer qu'en quelques itérations, nous stagnons autour d'une valeur maximale. Après un certain moment, il n'y a plus rien qui marche. Ce moment arrive plus vite lorsque le nombre d'objets augmente.

2.2 Même poids, même nombre d'objets, mais valeur aléatoire

[6]:
```
# Données du problème générées aléatoirement
nombre_objets = 10  # Le nombre d'objets
capacite_max = 30  # La capacité du sac
poids_min = 5  # Le poids minimal d'un objet
poids_max = 5
valeur_min = 1  # La valeur minimale d'un objet
valeur_max = 10  # La valeur maximale d'un objet

# paramètres de l'algorithme génétique
nbr_generations = 100  # nombre de générations
solutions_par_pop = 8
```

```
ID_objets, population_initiale, pop_size = iks.
 ↪fct_population_initiale(solutions_par_pop, nombre_objets)

iks.affichage(nbr_generations, capacite_max, nombre_objets, poids_min,␣
 ↪poids_max, valeur_min, valeur_max,
              ID_objets, population_initiale, pop_size)
```

```
Taille de la population: (8, 10)

Voici la dernière génération de la population:
[[0 1 1 1 1 1 0 1 0 0]
 [0 1 1 1 1 1 0 1 0 0]
 [0 1 1 1 1 1 0 1 0 0]
 [0 1 1 1 1 1 0 1 0 0]
 [0 1 1 1 1 1 0 1 0 0]
 [0 1 1 1 1 1 0 1 1 0]
 [0 1 1 1 1 1 0 1 0 0]
 [0 1 1 1 0 1 0 1 0 0]]

Fitness de la dernière génération:
[33 33 33 33 33  0 33 26]

La solution optimale est:
[array([0, 1, 1, 1, 1, 1, 0, 1, 0, 0])]
objets n° [1, 2, 3, 4, 5, 7]
(100, 8)
Avec une valeur de 33 € et un poids de 30.0 kg
Les objets qui maximisent la valeur contenue dans le sac sans le déchirer :
1
2
3
4
5
7
```

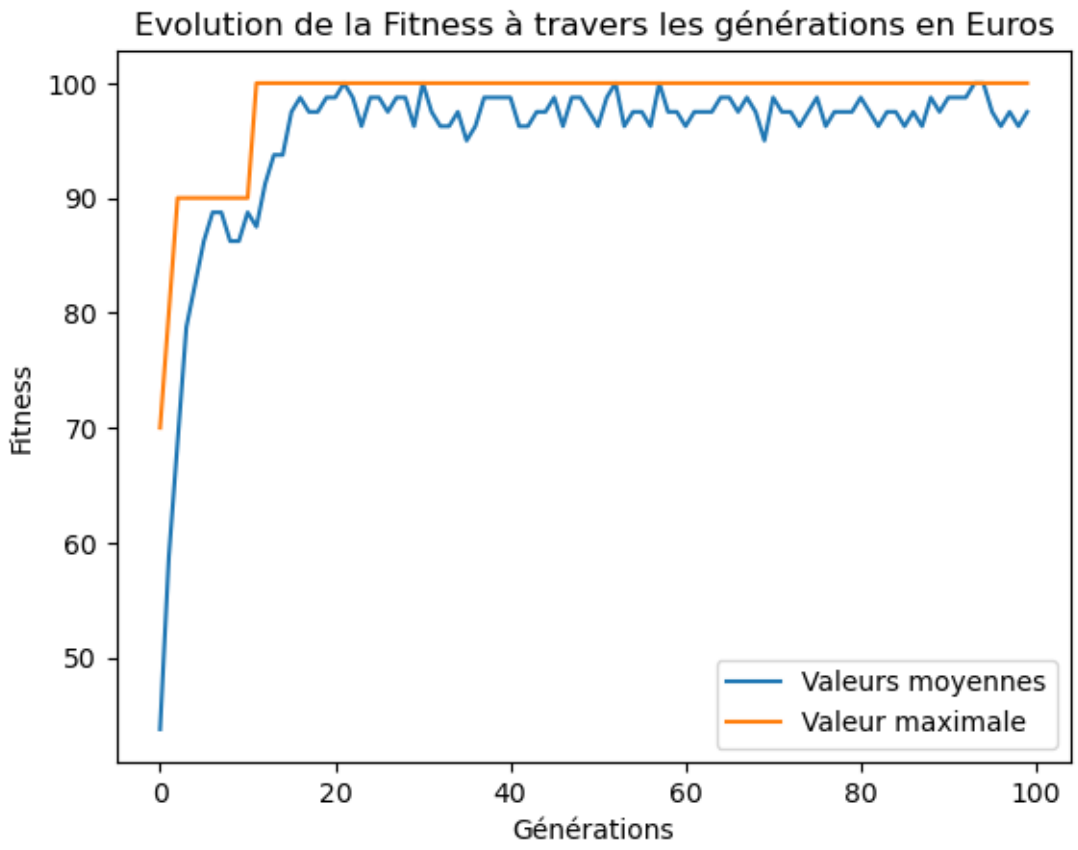


2.3 Même valeur, mais poids aléatoire

[7]:
```
# Données du problème générées aléatoirement
nombre_objets = 10  # Le nombre d'objets
capacite_max = 30  # La capacité du sac
poids_min = 1  # Le poids minimal d'un objet
poids_max = 5
valeur_min = 10  # La valeur minimale d'un objet
valeur_max = 10  # La valeur maximale d'un objet

# paramètres de l'algorithme génétique
nbr_generations = 100  # nombre de générations
solutions_par_pop = 8

ID_objets, population_initiale, pop_size = iks.
 ↪fct_population_initiale(solutions_par_pop, nombre_objets)

iks.affichage(nbr_generations, capacite_max, nombre_objets, poids_min,␣
 ↪poids_max, valeur_min, valeur_max,
```

```
                ID_objets, population_initiale, pop_size)
```

```
Taille de la population: (8, 10)

Voici la dernière génération de la population:
[[1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 0 1 1 1]]

Fitness de la dernière génération:
[100 100 100 100 100 100 100  90]

La solution optimale est:
[array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])]
objets n° [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
(100, 8)
Avec une valeur de 100 € et un poids de 26 kg
Les objets qui maximisent la valeur contenue dans le sac sans le déchirer :
0
1
2
3
4
5
6
7
8
9
```



2.3.1 Réponse :

Nous remarquons que l'algorithme évolue vers une solution locale optimale en quelques itérations, il y arrive en généralement 3 ou 4 paliers et après il stagne.

3 Apporter les modifications nécessaires pour aboutir seulement à des solutions faisables, si elles existent. Voici les approches qu'il faut coder et comparer :

```
- Mettre une fitness négative proportionnelle au dépassement de la capacité du sac
- Vérifier lors de la génération des solutions, du croisement et de la mutation que ces soluti
```

Pour comparer les deux approches, générer 4 instances du problème et stocker les dans des fichiers csv. Comparer les deux approches en termes de résultats et de temps de calcul.

[8]:
```
import Faisable_KnapSac as fks
import numpy as np
import os

# Créez le dossier 'actions' s'il n'existe pas
```

```python
if not os.path.exists('./populations'):
    os.makedirs('./populations')

# Création de 4 instances du problème

# Données du problème générées aléatoirement
nombre_objets = 40  # Le nombre d'objets
poids_min = 1  # Le poids minimal d'un objet
poids_max = 10
valeur_min = 1  # La valeur minimale d'un objet
valeur_max = 10  # La valeur maximale d'un objet

# paramètres de l'algorithme génétique
solutions_par_pop = 15

for i in range(4):
    _, population_initiale, _ = fks.fct_population_initiale(solutions_par_pop,␣
 ↪nombre_objets)

    nom_fichier = f'./populations/population_{i + 1}.csv'
    np.savetxt(nom_fichier, population_initiale, delimiter=",", fmt='%d')

    if poids_min == poids_max:
        poids = poids_min * np.ones(nombre_objets)
    else:
        poids = np.random.randint(poids_min, poids_max,
                                  size=nombre_objets)  # Poids des objets␣
 ↪générés aléatoirement entre 1kg et 15kg
    nom_poids = f'./populations/poids_{i + 1}.csv'
    np.savetxt(nom_poids, poids, delimiter=",", fmt='%d')
    if valeur_max == valeur_min:
        valeur = valeur_max * np.ones(nombre_objets)
    else:
        valeur = np.random.randint(valeur_min, valeur_max,
                                   size=nombre_objets)  # Valeurs des objets␣
 ↪générées aléatoirement entre 50€ et 350€
    nom_valeur = f'./populations/valeur_{i + 1}.csv'
    np.savetxt(nom_valeur, valeur, delimiter=",", fmt='%d')

    print(f'Taille de la population: {pop_size}')
    # print(f'Population Initiale: \n{population_initiale}')
    print(f'ID des objets: {ID_objets}')
```

```
Taille de la population: (8, 10)
ID des objets: [0 1 2 3 4 5 6 7 8 9]
Taille de la population: (8, 10)
ID des objets: [0 1 2 3 4 5 6 7 8 9]
```

```
Taille de la population: (8, 10)
ID des objets: [0 1 2 3 4 5 6 7 8 9]
Taille de la population: (8, 10)
ID des objets: [0 1 2 3 4 5 6 7 8 9]
```

Une fois les valeurs générées, nous pouvons les utiliser pour tester les deux approches.

3.1 Utilisation du code initial pour utiliser la fitness négative

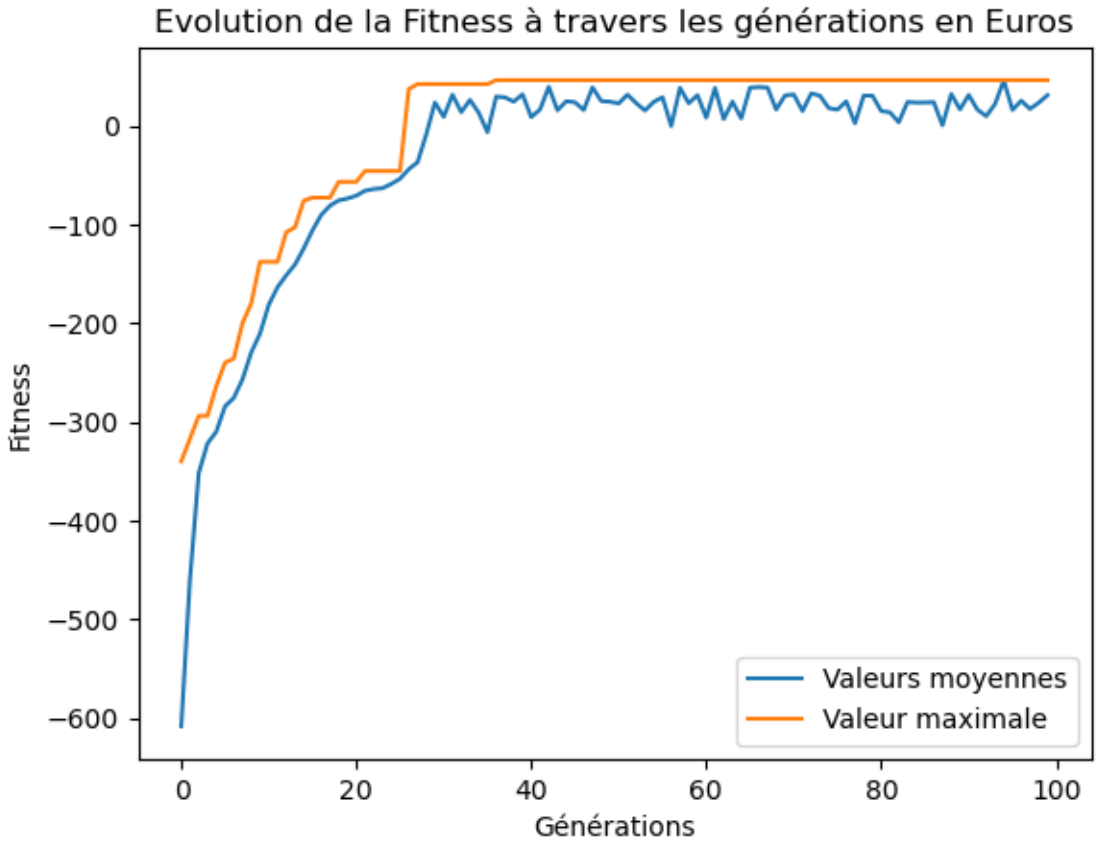
[9]:
```python
import Faisable_KnapSac as fks
import numpy as np

nbr_generations = 100  # nombre de générations
capacite_max = 20  # La capacité du sac

for i in range(4):
    # Charger la matrice depuis le fichier CSV
    nom_fichier = f'./populations/population_{i + 1}.csv'
    population_initiale = np.loadtxt(nom_fichier, delimiter=",")
    nom_poids = f'./populations/poids_{i + 1}.csv'
    poids = np.loadtxt(nom_poids, delimiter=",")
    nom_valeur = f'./populations/valeur_{i + 1}.csv'
    valeur = np.loadtxt(nom_valeur, delimiter=",")

    # Convertir en int si nécessaire
    population_initiale = population_initiale.astype(int)
    ID_objets = array = np.arange(0, len(population_initiale[0]))
    pop_size = (len(population_initiale), len(ID_objets))

    #print(f"En utilisant la correction {i+1}")
    #fks.affichage(nbr_generations, capacite_max, poids, valeur, ID_objets,␣
↪population_initiale, pop_size, "0")
    print(f"En utilisant la fitness négative {i + 1}")
    fks.affichage(nbr_generations, capacite_max, poids, valeur, ID_objets,␣
↪population_initiale, pop_size, "négatif")
```

```
En utilisant la fitness négative 1

Fitness de la dernière génération:
[ 43  43  43  43  43  43  43  43 -61 -54  43  43  43  43  43]

La solution optimale est:
[array([1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0,
       0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])]
objets n° [0, 12, 15, 20, 26, 27, 28]
(100, 15)
Avec une valeur de 43 € et un poids de 20.0 kg
Les objets qui maximisent la valeur contenue dans le sac sans le déchirer :
```

0
12
15
20
26
27
28

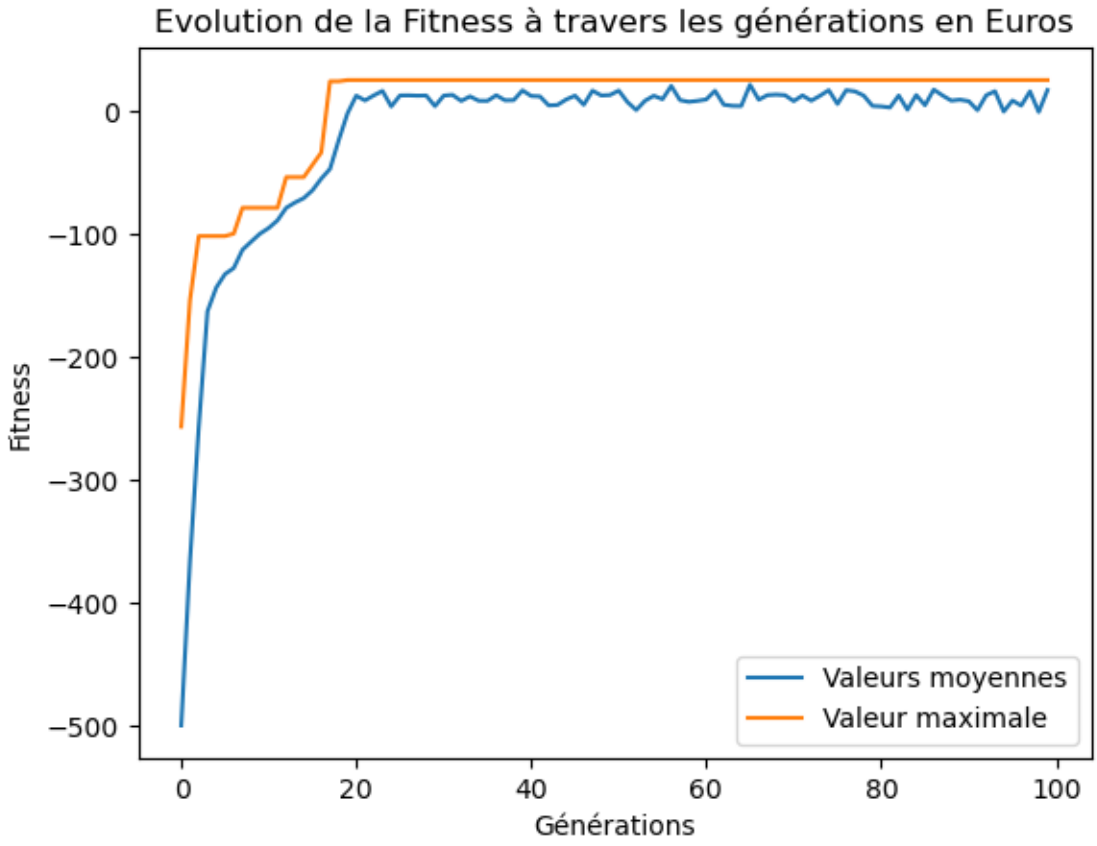

```
En utilisant la fitness négative 2

Fitness de la dernière génération:
[ 46  46  46  46  46  46  46 -60  46  46  46 -64 -77 -78  46]

La solution optimale est:
[array([0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
       1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1])]
objets n° [4, 5, 8, 16, 22, 27, 34, 39]
(100, 15)
Avec une valeur de 46 € et un poids de 20.0 kg
Les objets qui maximisent la valeur contenue dans le sac sans le déchirer :
4
```

5
8
16
22
27
34
39



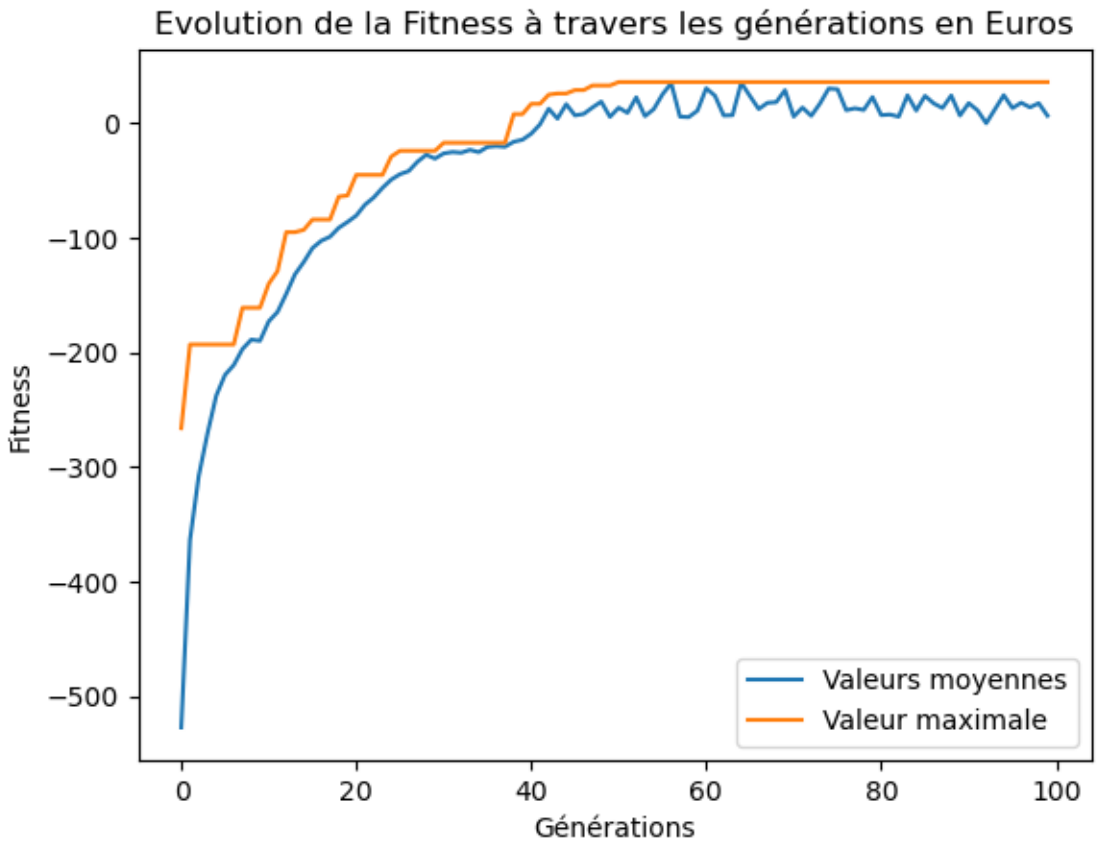

```
En utilisant la fitness négative 3

Fitness de la dernière génération:
[ 25  25  25  25  25  25  25  25  25  25 -39  25 -35  25 -33]

La solution optimale est:
[array([0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
       1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])]
objets n° [1, 8, 20, 21, 22, 39]
(100, 15)
Avec une valeur de 25 € et un poids de 20.0 kg
Les objets qui maximisent la valeur contenue dans le sac sans le déchirer :
1
```

8
20
21
22
39




```
En utilisant la fitness négative 4

Fitness de la dernière génération:
[ 36  36  36  36  36  36  36  36 -52 -59 -54 -49  36 -59  36]

La solution optimale est:
[array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1,
       0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0])]
objets n° [11, 15, 19, 21, 23, 34]
(100, 15)
Avec une valeur de 36 € et un poids de 20.0 kg
Les objets qui maximisent la valeur contenue dans le sac sans le déchirer :
11
15
19
```

21
23
34

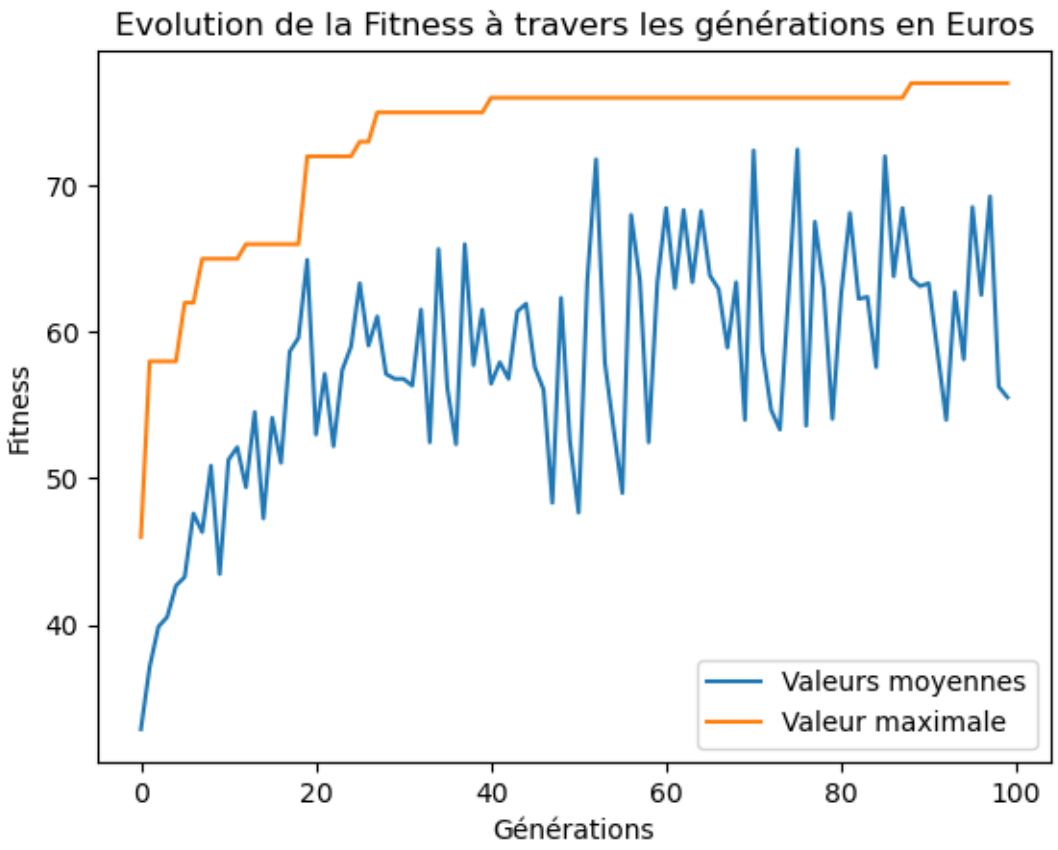


3.2 Utilisation du code initial modifié pour vérifier la faisabilité des solutions (capacité du sac)

[10]:
```python
import Faisable_KnapSac as fks
import numpy as np

nbr_generations = 100  # nombre de générations
capacite_max = 30  # La capacité du sac

for i in range(4):
    # Charger la matrice depuis le fichier CSV
    nom_fichier = f'./populations/population_{i + 1}.csv'
    population_initiale = np.loadtxt(nom_fichier, delimiter=",")
    nom_poids = f'./populations/poids_{i + 1}.csv'
    poids = np.loadtxt(nom_poids, delimiter=",")
    nom_valeur = f'./populations/valeur_{i + 1}.csv'
```

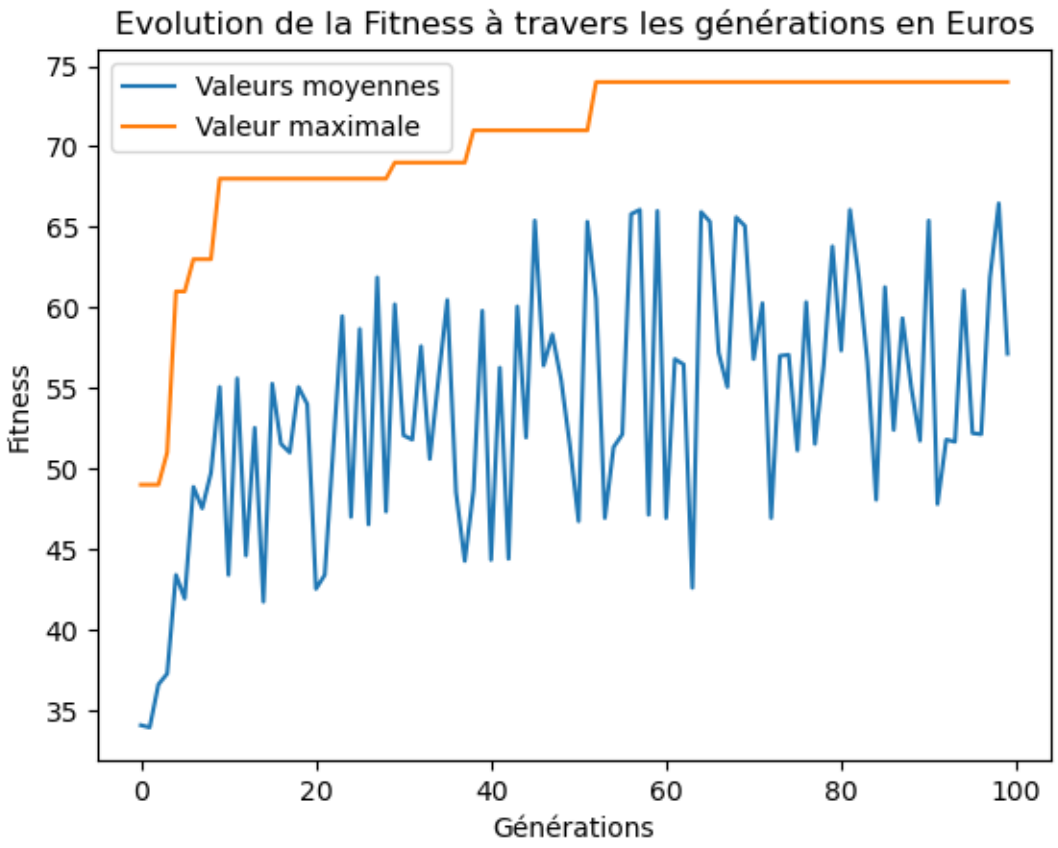
```
    valeur = np.loadtxt(nom_valeur, delimiter=",")

    # Convertir en int si nécessaire
    population_initiale = population_initiale.astype(int)
    ID_objets = array = np.arange(0, len(population_initiale[0]))
    pop_size = (len(population_initiale), len(ID_objets))

    print(f"En utilisant la correction {i + 1}")
    fks.affichage(nbr_generations, capacite_max, poids, valeur, ID_objets,␣
↪population_initiale, pop_size, "0")
    #print(f"En utilisant la fitness négative {i+1}")
    #fks.affichage(nbr_generations, capacite_max, poids, valeur, ID_objets,␣
↪population_initiale, pop_size, "négatif")
```

```
En utilisant la correction 1

Fitness de la dernière génération:
[77 77 77 77 77 77 77 77 77 77 77 68 77 77  0]

La solution optimale est:
[array([1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
       0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0])]
objets n° [0, 8, 12, 19, 20, 23, 26, 28, 29, 34, 37]
(100, 15)
Avec une valeur de 77 € et un poids de 29.0 kg
Les objets qui maximisent la valeur contenue dans le sac sans le déchirer :
0
8
12
19
20
23
26
28
29
34
37
```



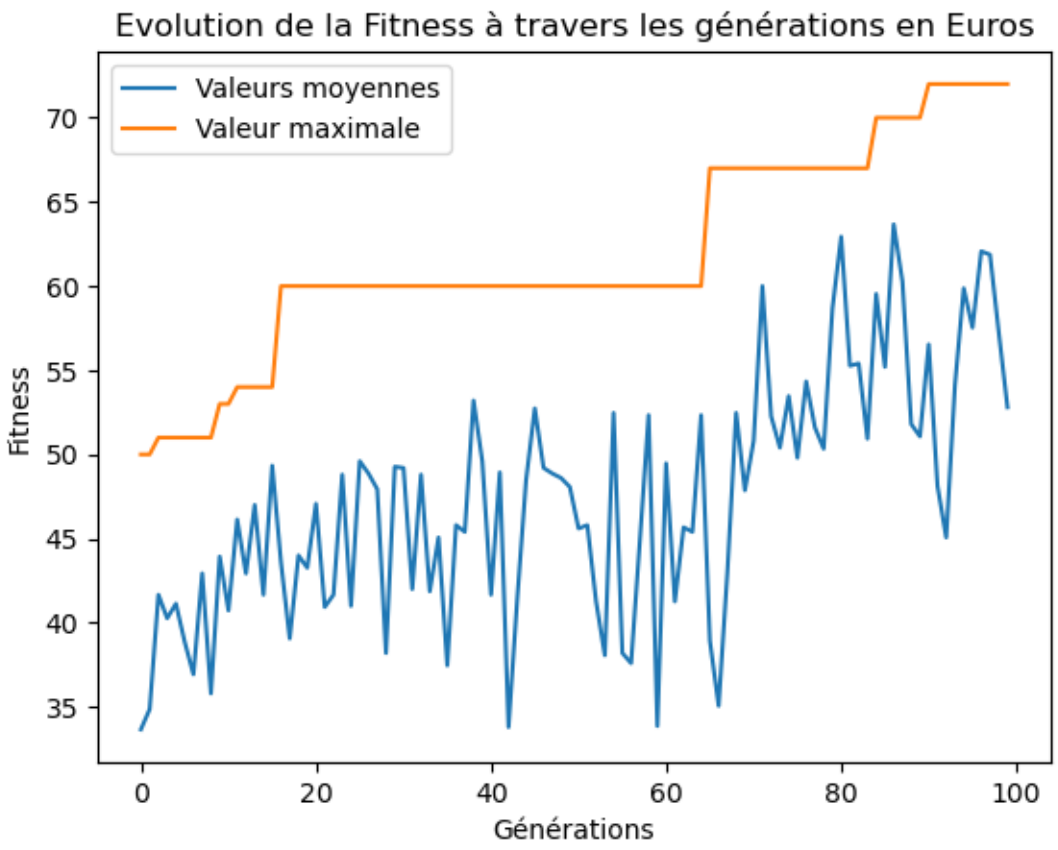

```
En utilisant la correction 2

Fitness de la dernière génération:
[74 74 74 74 74 73 73 65  0 70  0 61  0 66 66]

La solution optimale est:
[array([0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0,
       0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1])]
objets n° [2, 4, 5, 8, 16, 17, 19, 23, 27, 38, 39]
(100, 15)
Avec une valeur de 74 € et un poids de 30.0 kg
Les objets qui maximisent la valeur contenue dans le sac sans le déchirer :
2
4
5
8
16
17
19
23
```

27
38
39



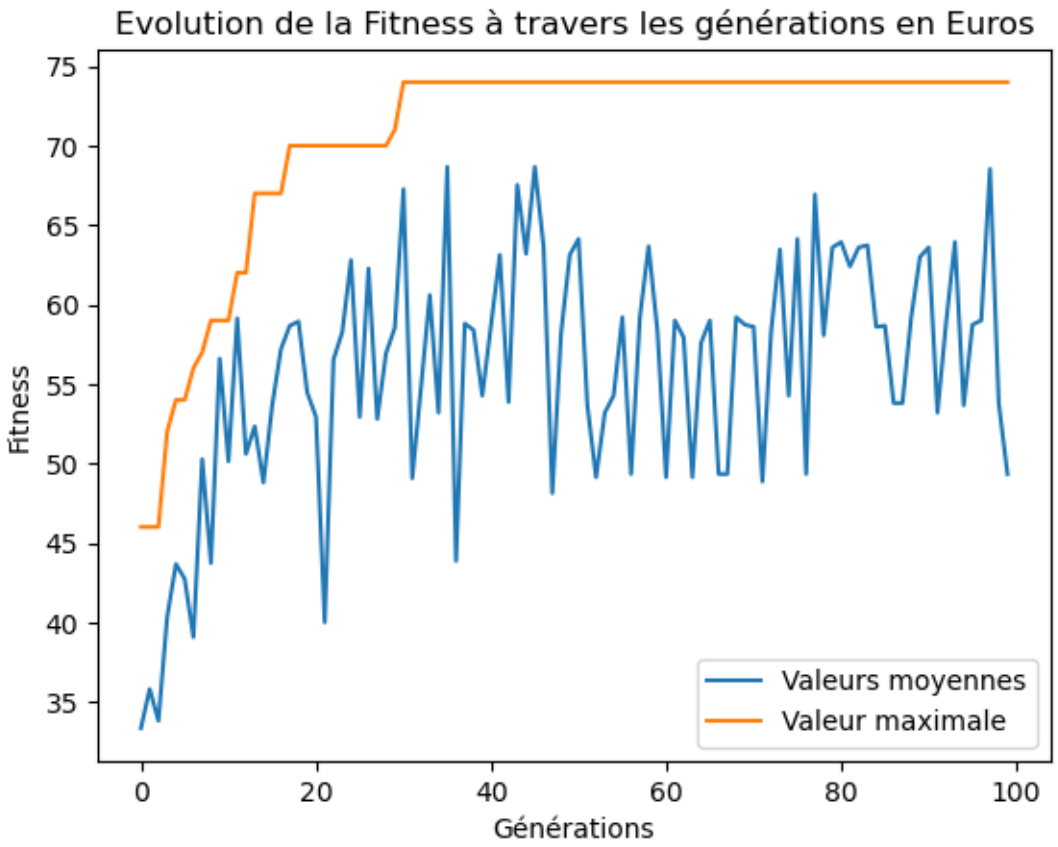

```
En utilisant la correction 3

Fitness de la dernière génération:
[72 72 72 72 72 72 72 72  0  0 72  0  0 72  0]

La solution optimale est:
[array([0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
       1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0])]
objets n° [5, 6, 21, 22, 25, 27, 31, 35, 36, 37]
(100, 15)
Avec une valeur de 72 € et un poids de 29.0 kg
Les objets qui maximisent la valeur contenue dans le sac sans le déchirer :
5
6
21
22
25
```

27
31
35
36
37




```
En utilisant la correction 4

Fitness de la dernière génération:
[74 74 74 74 74 74 74 67  0 74  0 68  0 71 74]

La solution optimale est:
[array([1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1,
       1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0])]
objets n° [0, 3, 11, 15, 16, 18, 20, 21, 22, 23, 34, 35]
(100, 15)
Avec une valeur de 74 € et un poids de 29.0 kg
Les objets qui maximisent la valeur contenue dans le sac sans le déchirer :
0
3
11
```

15
16
18
20
21
22
23
34
35



3.2.1 Réponse :

Avec la correction, il arrive à trouver une meilleure solution. Par exemple sur la 4 itération nous avons un sac de valeur 55€ pour un poids de 29kg, alors que la fitness négative nous donne un sac de valeur 46€ pour un poids de 20kg.

4 Nous souhaitons utiliser cet algorithme à la gestion du portefeuille.

Pour des raisons de minimisation du risque, nous considérons que le client peut acheter plusieurs actions du même titre dans la mesure où il respecte la limite des 20% du budget total. L'achat de plusieurs actions ne concerne qu'un seul titre. A la suite des estimations des gains escomptés de chaque titre, il souhaite augmenter ses gains en respectant les limites de son budget et la contrainte de minimisation de risque. Adapter le programme à la gestion du portefeuille.

Ici, on peut prendre deux fois le meme objet, mais pas plus de 20% de l'ensemble des objets

Les paramètres sont les suivants : - nombre_action_différente : nombre d'actions différentes - nombre_max_par_action : nombre maximum d'actions par action - prix_max_action : prix maximum d'une action - prix_actions : tableau des prix des actions - budget : budget total - rentabilites : tableau des rentabilités des actions (pourcentage / an) - taille_portefeuille : taille du portefeuille = nombre d'actions différentes - limite_budget : limite du budget pour une action = 20% du budget total

[11]:
```
import numpy as np
import random
import matplotlib.pyplot as plt
```

4.1 Initialisation des paramètres

[12]:
```
# Initialisation des paramètres
nombre_action_differente = 10
nombre_max_par_action = 50
prix_max_action = 100
budget = 1000
taille_portefeuille = nombre_action_differente
limite_budget = budget * 0.2
taille_population = 10

probabilite_mutation = 0.1
nombre_iteration = 100


def init_parametres(prix_max_action, nombre_action_differente):
    prix_actions = np.random.randint(1, prix_max_action,␣
 ↪size=nombre_action_differente)
    rentabilites = np.random.randint(-100, 100, size=nombre_action_differente) /
 ↪ 100
    max_action = np.random.randint(1, nombre_max_par_action,␣
 ↪size=nombre_action_differente)
    return prix_actions, rentabilites, max_action

# prix_actions, rentabilites, max_action = init_parametres(prix_max_action,␣
 ↪nombre_action_differente)
```

```
# print(f'prix_actions: {prix_actions}')
# print(f'rentabilites: {rentabilites}')
# print(f'max_action: {max_action}')
```

4.2 Initialisation de la population

[13]:
```
def individu_initiale(taille_portefeuille, max_action):
    population_initiale = np.zeros(taille_portefeuille)
    for i in range(taille_portefeuille):
        population_initiale[i] = np.random.randint(0, max_action[i])
    return population_initiale


# indicidu = individu_initiale(taille_portefeuille, max_action)
# print(f'population_initiale: {indicidu}')


def population_initiale(taille_population, taille_portefeuille, max_action):
    population_initiale = []
    for i in range(taille_population):
        population_initiale.append(individu_initiale(taille_portefeuille,␣
 ↪max_action))
    return population_initiale

# population_initiale = population_initiale(taille_population,␣
 ↪taille_portefeuille, max_action)
# print(f'population_initiale: {population_initiale}')
```

4.3 Création de la fonction de fitness

[14]:
```
# Fonction de fitness

def fitness(individu, prix_actions, rentabilites, budget, limite_budget):
    fitness = 0
    for i in range(len(individu)):
        fitness += individu[i] * prix_actions[i] * rentabilites[i]
    if np.sum(individu * prix_actions) > budget:
        fitness = 0
        for k in range(len(individu)):
            if individu[k] * prix_actions[k] > limite_budget:
                fitness = 0
    return fitness

# print(f'fitness: {fitness(indicidu, prix_actions, rentabilites, budget,␣
 ↪limite_budget)}')
```

4.4 Correction de la population si individu non faisable

[15]:
```
# Correction de la population initiale

def correction_individu(individu, budget, limite_budget):
    # Vérifier si la fitness de l'individu est nulle. Si c'est le cas, il␣
 ↪nécessite une correction.
    if fitness(individu, prix_actions, rentabilites, budget, limite_budget) ==␣
 ↪0:
        # Initialisation d'un individu corrigé rempli de zéros.
        individu_corrige = np.zeros(len(individu))

        # Génération d'une liste d'indices correspondant aux actions et mélange␣
 ↪aléatoire de cette liste.
        indices = list(range(len(individu)))
        random.shuffle(indices)

        # Initialisation du budget utilisé pour la correction.
        compte_budget = 0

        # Parcours des indices mélangés pour corriger l'individu.
        for j in indices:
            # Vérification si l'ajout de l'action courante ne dépasse pas le␣
 ↪budget et la limite par action.
            if individu[indices[j]] * prix_actions[indices[j]] <= limite_budget␣
 ↪and \
                compte_budget + individu[indices[j]] * prix_actions[indices[j]]␣
 ↪<= budget:
                # Affectation de l'action à l'individu corrigé.
                individu_corrige[indices[j]] = individu[indices[j]]
                # Mise à jour du budget utilisé.
                compte_budget += individu[indices[j]] * prix_actions[indices[j]]

            # Vérification si le budget est dépassé.
            if compte_budget > budget:
                for k in range(individu[indices[j]]):
                    # Vérification et ajustement pour chaque action pour rester␣
 ↪dans les limites du budget.
                    if k * prix_actions[indices[j]] <= limite_budget and \
                       compte_budget + k * prix_actions[indices[j]] <= budget:
                        individu_corrige[k] = k * prix_actions[indices[j]]
                    else:
                        compte_budget += k * prix_actions[indices[j]]
        # Retour de l'individu corrigé.
        return individu_corrige
    else:
        # Si la fitness n'est pas nulle, retourner l'individu original.
```

```
        return individu



def correction_population(population, budget, limite_budget):
    # Initialisation d'une nouvelle liste pour la population corrigée.
    population_corrige = []

    # Parcours de chaque individu de la population pour correction.
    for i in range(len(population)):
        population_corrige.append(correction_individu(population[i], budget,␣
 ↪limite_budget))

    # Retour de la population corrigée.
    return population_corrige


# population_corrige = correction_population(population_initiale, budget,␣
 ↪limite_budget)
# print(f'population_corrige: {population_corrige}')
```

4.5 Fonction de sélection

[16]:
```
# Fonction de sélection

def selection(population, prix_actions, rentabilites, budget, limite_budget):
    # Initialisation d'une liste pour stocker la fitness de chaque individu de␣
 ↪la population.
    fitness_population = []

    # Calcul de la fitness pour chaque individu de la population.
    for i in range(len(population)):
        # Appel de la fonction fitness pour l'individu i avec les paramètres␣
 ↪donnés.
        fitness_population.append(fitness(population[i], prix_actions,␣
 ↪rentabilites, budget, limite_budget))

    # Conversion de la liste des fitness en un tableau numpy pour un traitement␣
 ↪plus efficace.
    fitness_population = np.array(fitness_population)

    # Obtention des indices des éléments triés selon leur fitness, en ordre␣
 ↪croissant.
    indices = np.argsort(fitness_population)

    # Inversion de l'ordre des indices pour avoir un tri en ordre décroissant␣
 ↪(meilleure fitness en premier).
```

```
    indices = indices[::-1]

    # Conversion de la liste de population en un tableau numpy pour une␣
 ↪indexation facile.
    population = np.array(population)

    # Réarrangement de la population selon les indices triés pour que les␣
 ↪meilleurs individus soient en premier.
    population = population[indices]

    # Retour de la population triée en fonction de leur fitness, du meilleur au␣
 ↪moins bon.
    return population


# print(f'selection: {selection(population_corrige, prix_actions, rentabilites,␣
 ↪budget, limite_budget)}')
```

4.6 Fonction de croisement

[17]:
```
# Fonction de croisement

def croisement(population, prix_actions, rentabilites, budget, limite_budget):
    population = selection(population, prix_actions, rentabilites, budget,␣
 ↪limite_budget)
    population_enfants = []
    chromosome1_parent1 = population[0][0:int(len(population[0]) / 2)]
    chromosome2_parent1 = population[0][int(len(population[0]) / 2):]
    chromosome1_parent2 = population[1][int(len(population[0]) / 2):]
    chromosome2_parent2 = population[1][0:int(len(population[0]) / 2)]
    chromosome_enfant1 = np.concatenate((chromosome1_parent1,␣
 ↪chromosome2_parent2))
    chromosome_enfant2 = np.concatenate((chromosome2_parent1,␣
 ↪chromosome1_parent2))
    chromosome_enfant3 = np.concatenate((chromosome1_parent1,␣
 ↪chromosome1_parent2))
    chromosome_enfant4 = np.concatenate((chromosome2_parent2,␣
 ↪chromosome2_parent2))
    population_enfants.append(chromosome_enfant1)
    population_enfants.append(chromosome_enfant2)
    population_enfants.append(chromosome_enfant3)
    population_enfants.append(chromosome_enfant4)
    return population_enfants

# print(f'croisement: {croisement(population_corrige, prix_actions,␣
 ↪rentabilites, budget, limite_budget)}')
```

4.7 Fonction de mutation

[18]:
```
# Fonction de mutation

def mutation(individu, prix_actions, rentabilites, budget, limite_budget):
    taille_individu = len(individu)
    fitness_individu_initiale = fitness(individu, prix_actions, rentabilites,␣
 ↪budget, limite_budget)
    individu_mute = individu
    indice1 = np.random.randint(0, taille_individu)
    indice2 = np.random.randint(0, taille_individu)
    individu_mute[indice1], individu_mute[indice2] = individu_mute[indice2],␣
 ↪individu_mute[indice1]
    individu_mute_corrige = correction_individu(individu_mute, budget,␣
 ↪limite_budget)
    # fitness_individu_mute = fitness(individu_mute_corrige, prix_actions,␣
 ↪rentabilites, budget, limite_budget)
    # if fitness_individu_mute > fitness_individu_initiale:
    #     return individu_mute_corrige
    # else:
    #     return individu
    return individu_mute_corrige

# print(f'mutation: {mutation(population_corrige[0])}')
```

4.8 Fonction de mutation de la population

[19]:
```
# Fonction de mutation de la population

def mutation_population(population, probabilite_mutation, budget,␣
 ↪limite_budget, rentabilites, prix_actions):
    for i in range(len(population)):
        if np.random.random() < probabilite_mutation:
            population[i] = mutation(population[i], prix_actions, rentabilites,␣
 ↪budget, limite_budget)
    return population

# print(f'mutation_population: {mutation_population(population_corrige,␣
 ↪probabilite_mutation, budget, limite_budget)}')
```

[20]:
```
# Fonction pour revenir à un population de taille initiale

def population_taille_initiale(population, prix_actions, rentabilites, budget,␣
 ↪limite_budget):
    population = selection(population, prix_actions, rentabilites, budget,␣
 ↪limite_budget)
    population = population[0:taille_population]
```

```
    return population
```

4.9 Algorithme génétique

[21]:
```
# Fonction principale
prix_actions, rentabilites, max_action = init_parametres(prix_max_action,␣
 ↪nombre_action_differente)
print(f'prix_actions: {prix_actions}')
print(f'rentabilites: {rentabilites}')
print(f'max_action: {max_action}')
population_initia = population_initiale(taille_population, taille_portefeuille,␣
 ↪max_action)
population_corrige = correction_population(population_initia, budget,␣
 ↪limite_budget)


def algorithme_genetique(population, prix_actions, rentabilites, budget,␣
 ↪limite_budget, probabilite_mutation, nombre_iteration):
    # Initialisation de listes pour suivre l'évolution de la fitness au fil des␣
 ↪générations.
    historique_fitness = []
    max_fitness = []

    # Boucle principale de l'algorithme, itérant sur le nombre de générations␣
 ↪défini.
    for i in range(nombre_iteration):
        # Génération d'enfants par croisement de la population actuelle.
        enfants = croisement(population, prix_actions, rentabilites, budget,␣
 ↪limite_budget)

        # Correction des enfants pour s'assurer qu'ils respectent les␣
 ↪contraintes de budget et limite par action.
        enfants_corrige = correction_population(enfants, budget, limite_budget)

        # Fusion de la population actuelle et des enfants corrigés.
        population = np.concatenate((population, enfants_corrige))

        # Application des mutations sur la population avec une probabilité␣
 ↪donnée.
        population = mutation_population(population, probabilite_mutation,␣
 ↪budget, limite_budget, rentabilites, prix_actions)

        # Réduction de la taille de la population à sa taille initiale, et tri␣
 ↪en fonction de la fitness.
        population = population_taille_initiale(population, prix_actions,␣
 ↪rentabilites, budget, limite_budget)
```

```
        # Évaluation de la fitness du meilleur individu dans la population␣
↪actuelle.
        fitness_meilleur_individu = fitness(population[0], prix_actions,␣
↪rentabilites, budget, limite_budget)

        # Ajout de la fitness du meilleur individu à l'historique.
        historique_fitness.append(fitness_meilleur_individu)

        # Mise à jour de la liste des meilleures fitness si nécessaire.
        max_historique_fitness = max(historique_fitness)
        if max_historique_fitness > fitness_meilleur_individu:
            max_fitness.append(max_historique_fitness)
        else:
            max_fitness.append(fitness_meilleur_individu)

    # Retour de la population finale, l'historique de la fitness et le maximum␣
↪de fitness à chaque génération.
    return population, historique_fitness, max_fitness
```

```
prix_actions: [30 48 78 43 56 39 91 46 52 19]
rentabilites: [-0.3  -0.78 -0.24 -0.01 -0.13 -0.11 -0.71  0.18 -0.64 -0.82]
max_action: [48  9 12 27 17 20  5 10 47 47]
```

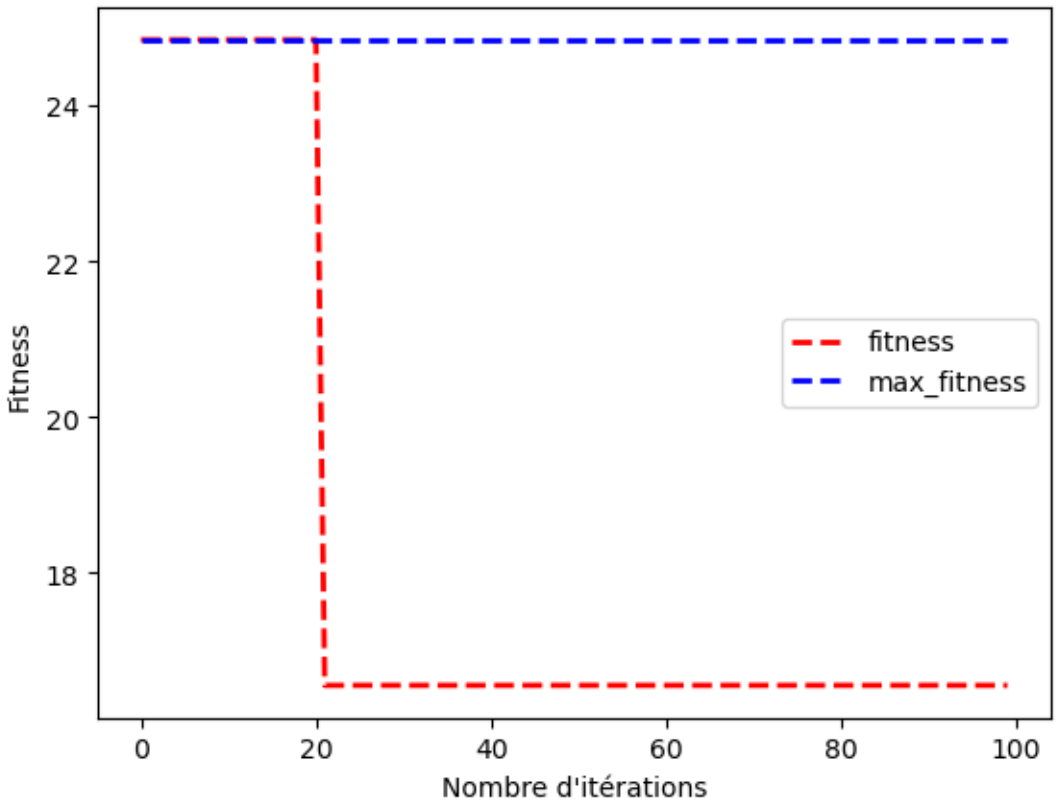
[22]:
```
# Affichage des résultats

def affichage(population, historique_fitness, max_fitness):
    plt.plot(historique_fitness, 'r--', linewidth=2, label='fitness')
    plt.plot(max_fitness, 'b--', linewidth=2, label='max_fitness')
    plt.xlabel('Nombre d\'itérations')
    plt.ylabel('Fitness')
    plt.legend()
    plt.show()
    # print(f'population: {population}')
    # print(f'historique_fitness: {historique_fitness}')
    # print(f'max_fitness: {max_fitness}')
```

4.10 Résultats

[23]:
```
# Résultats

print(f'population_actions: {population_corrige}')
print(f'prix_actions: {prix_actions}')
print(f'rentabilites: {rentabilites}')
population, historique_fitness, max_fitness =␣
↪algorithme_genetique(population_corrige, prix_actions, rentabilites,
```

```
                                                                    budget,␣
  ↪limite_budget, probabilite_mutation,
                                                                    ␣
  ↪nombre_iteration)
affichage(population, historique_fitness, max_fitness)
```

```
population_actions: [array([0., 0., 0., 0., 0., 0., 0., 3., 0., 0.]), array([0.,
0., 0., 1., 0., 0., 0., 0., 0., 2.]), array([0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.]), array([3., 0., 0., 0., 0., 0., 0., 1., 0., 9.]), array([0., 3., 0., 0.,
0., 5., 2., 0., 0., 0.]), array([0., 0., 0., 0., 0., 0., 0., 2., 0., 0.]),
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 3.]), array([0., 1., 0., 0., 0., 0.,
0., 2., 0., 0.]), array([0., 2., 0., 0., 0., 0., 1., 2., 2., 4.]), array([0.,
0., 0., 0., 3., 0., 2., 0., 0., 0.])]
prix_actions: [30 48 78 43 56 39 91 46 52 19]
rentabilites: [-0.3  -0.78 -0.24 -0.01 -0.13 -0.11 -0.71  0.18 -0.64 -0.82]
```



5 Choisir 4 instances des données du problème et stocker les dans des fichiers csv.

Améliorer les performances de l'algorithme génétique sur ces quatre instances. Décrivez dans le rapport les améliorations réalisées et les motivations.

[24]:
```
import os
```

[25]:
```
# Données du problème générées aléatoirement
nombre_action_differente = 10
nombre_max_par_action = 50
prix_max_action = 100
budget = 1000
taille_portefeuille = nombre_action_differente
limite_budget = budget * 0.2
taille_population = 10

probabilite_mutation = 0.1
nombre_iteration = 1000
```

5.1 Création des fichiers csv pour comparer les résultats des algorithmes sur les mêmes données

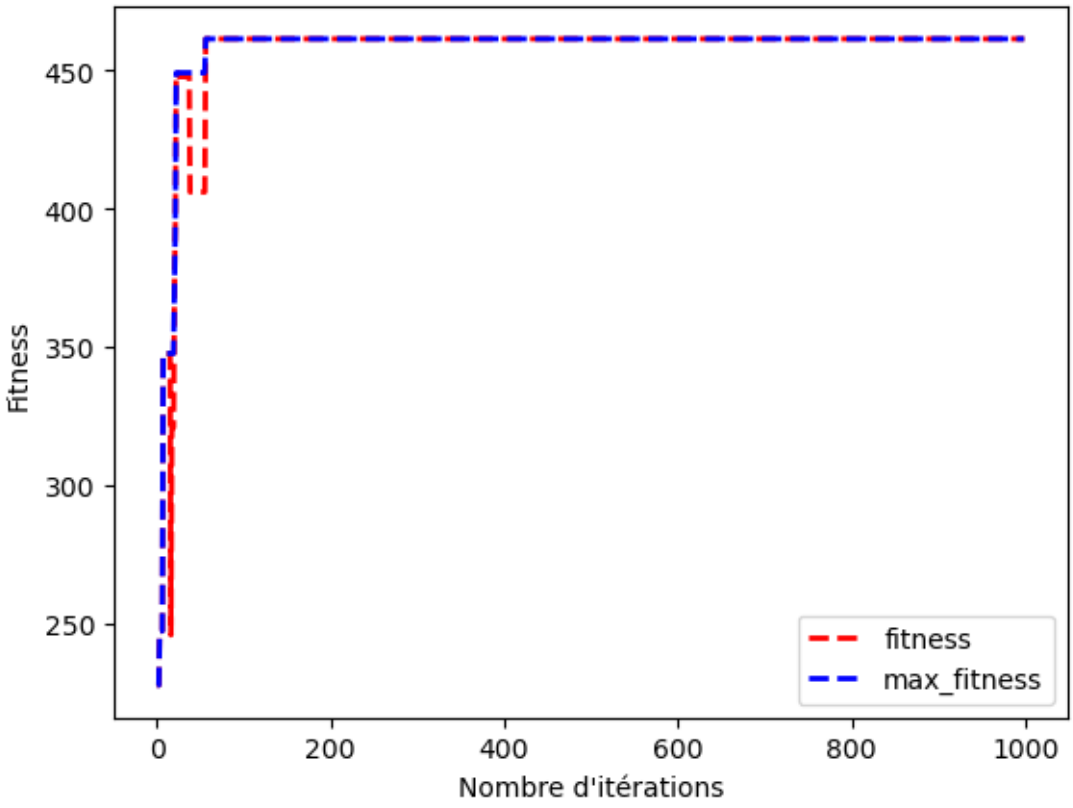
[26]:
```
# Création de 4 instances du problème

# Créez le dossier 'actions' s'il n'existe pas
if not os.path.exists('./actions'):
    os.makedirs('./actions')

for k in range(4):
    prix_actions, rentabilites, max_action = init_parametres(prix_max_action,␣
 ↪nombre_action_differente)
    print(f'prix_actions: {prix_actions}')
    print(f'rentabilites: {rentabilites}')
    print(f'max_action: {max_action}')
    population_initia = population_initiale(taille_population,␣
 ↪taille_portefeuille, max_action)
    population_corrige = correction_population(population_initia, budget,␣
 ↪limite_budget)
    # Sauvegarde de la population initiale
    nom_fichier = f'./actions/population_action{k + 1}.csv'
    np.savetxt(nom_fichier, population_corrige, delimiter=",", fmt='%d')
    nom_prix = f'./actions/prix_action{k + 1}.csv'
    np.savetxt(nom_prix, prix_actions, delimiter=",", fmt='%d')
    nom_rentabilites = f'./actions/rentabilites_action{k + 1}.csv'
    np.savetxt(nom_rentabilites, rentabilites, delimiter=",", fmt='%.2f')
```

```
prix_actions: [13 83  2 52  9 78 61 16 76 23]
rentabilites: [ 0.15  0.91 -0.33  0.01 -0.55  0.44 -0.9   0.5  -0.81  0.51]
max_action: [36 29 33 25 42 14 22 27 32  3]
prix_actions: [35 87 82 11 69  3 14 23 15 24]
rentabilites: [ 0.77  0.23  0.22 -1.    0.51 -0.24 -0.98  0.93 -0.86 -0.86]
max_action: [ 7 39 46 28 41 31 34 24  6  7]
prix_actions: [21 45 89 88 92 94 43 10 95 59]
```

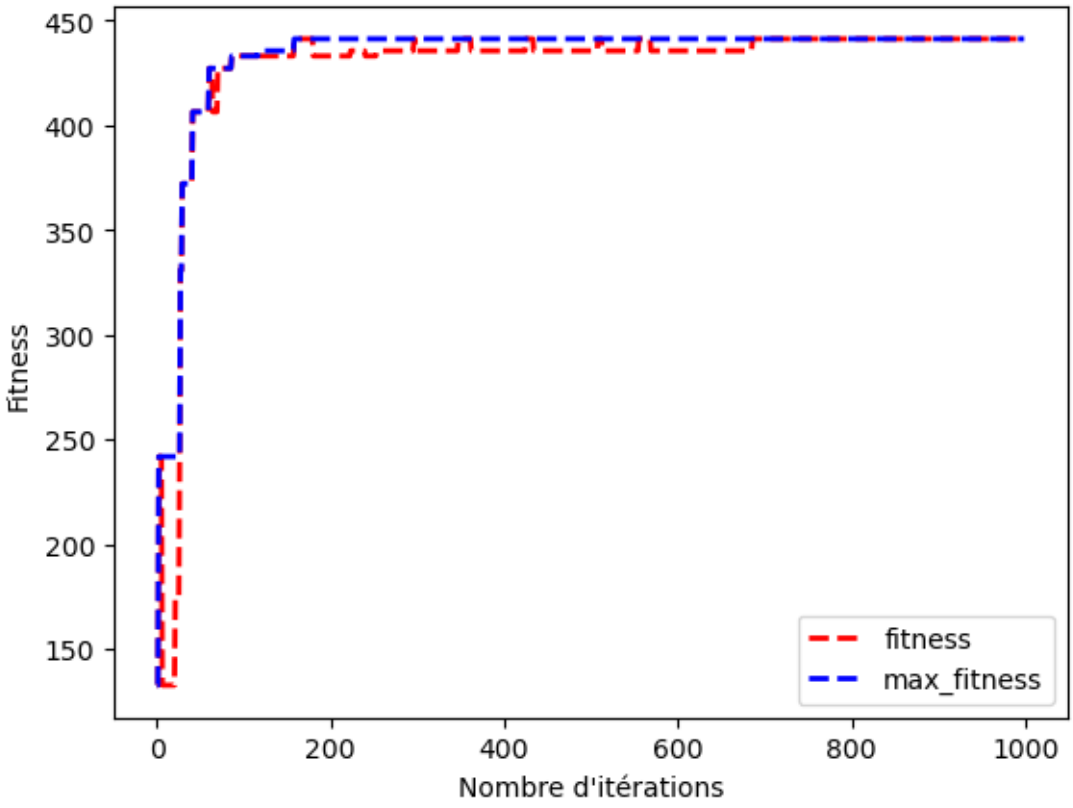
```
rentabilites: [-0.46  0.02  0.92 -0.33  0.39  0.74  0.08 -0.11  0.33  0.08]
max_action: [41 18 10 31 21 17 13 12 25 47]
prix_actions: [92 63 42  3 39 89  2 47 12 70]
rentabilites: [-0.8   0.12  0.73 -0.64  0.93  0.78  0.85 -0.52  0.99  0.3 ]
max_action: [46  3  1 23 11 26 36 12 47 19]
```

[27]:
```python
# Chargement des données

for k in range(4):
    # Charger la matrice depuis le fichier CSV
    nom_fichier = f'./actions/population_action{k + 1}.csv'
    population = np.loadtxt(nom_fichier, delimiter=",")
    nom_prix = f'./actions/prix_action{k + 1}.csv'
    prix_actions = np.loadtxt(nom_prix, delimiter=",")
    nom_rentabilites = f'./actions/rentabilites_action{k + 1}.csv'
    rentabilites = np.loadtxt(nom_rentabilites, delimiter=",")

    # Convertir en int si nécessaire
    # population = population.astype(int)
    # prix_actions = prix_actions.astype(int)
    # rentabilites = rentabilites.astype(int)



    # Algorithmes génétiques
    population, historique_fitness, max_fitness =␣
 ↪algorithme_genetique(population, prix_actions, rentabilites, budget,
                                                                        ␣
 ↪limite_budget, probabilite_mutation,
                                                                        ␣
 ↪nombre_iteration)

    print(f"Pour le fichier {nom_fichier}")
    # Affichage des résultats
    affichage(population, historique_fitness, max_fitness)
```

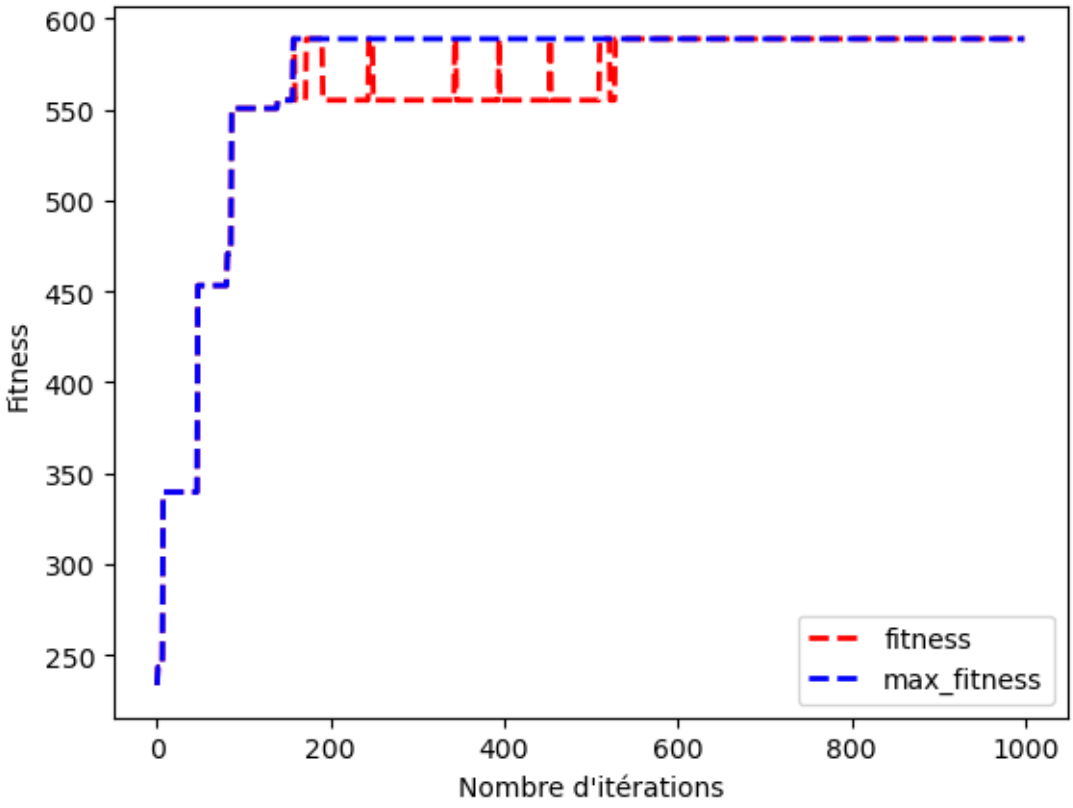
```
Pour le fichier ./actions/population_action1.csv
```



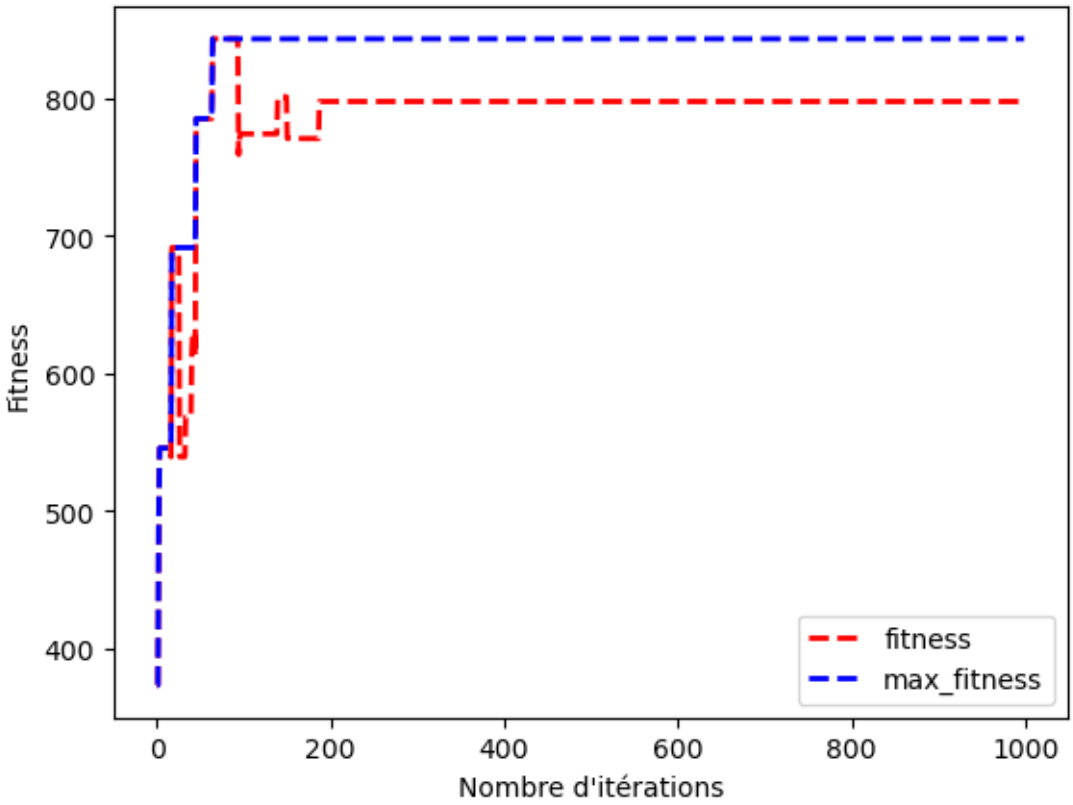
Pour le fichier ./actions/population_action2.csv



Pour le fichier ./actions/population_action3.csv



Pour le fichier ./actions/population_action4.csv



Bilan : - on peut avoir une chute de la fitness, car on mute des individus qui étaient meilleurs que les parents, ceci permet d'éviter de rester bloqué dans un minimum local

Axes d'amélioration : - on peut faire varier la probabilité de mutation en fonction de la fitness - on peut faire varier la taille de la population en fonction de la fitness - on peut faire varier le nombre d'itérations en fonction de la fitness - on peut faire varier la probabilité de croisement en fonction de la fitness - on peut utiliser d'autres méthodes de croisement : n point, masque, ... - on peut utiliser d'autres méthodes de mutation : inversion, décalage, ... - on peut utiliser d'autres méthodes de sélection : le meilleur, tournoi, roulette

[28]:
```
# Amélioration de la sélection par tournoi

def selection_tournoi(population, prix_actions, rentabilites, budget,␣
 ↪limite_budget):
    taille_pop = len(population)
    # On prend 10% d'individus au hasard
    indices = np.random.randint(0, taille_pop, size=int(taille_pop / 0.1))
    tournoi = []
    for k in range(len(indices)):
        tournoi.append(population[indices[k]])
    # Calcul de la fitness de chaque individu
    fitness_tournoi = []
```

```
    for j in range(len(tournoi)):
        fitness_tournoi.append(fitness(tournoi[j], prix_actions, rentabilites,␣
 ↪budget, limite_budget))
    # On prend les 2 meilleurs
    fitness_tournoi = np.array(fitness_tournoi)
    indices = np.argsort(fitness_tournoi)
    return tournoi[indices[-1]], tournoi[indices[-2]]
```

[29]:
```
# Amélioration de la méthode de croisement : croisement à 2 points

def croisement_2_points (population, prix_actions, rentabilites, budget,␣
 ↪limite_budget):
    population = selection_tournoi(population, prix_actions, rentabilites,␣
 ↪budget, limite_budget)
    population_enfants = []
    chromosome1_parent1 = population[0][0:int(len(population[0]) / 3)]
    chromosome2_parent1 = population[0][int(len(population[0]) / 3):
 ↪int(2*len(population[0]) / 3)]
    chromosome3_parent1 = population[0][int(2*len(population[0]) / 3):]
    chromosome1_parent2 = population[1][0:int(len(population[0]) / 3)]
    chromosome2_parent2 = population[1][int(len(population[0]) / 3):
 ↪int(2*len(population[0]) / 3)]
    chromosome3_parent2 = population[1][int(2*len(population[0]) / 3):]
    chromosome_enfant1 = np.concatenate((chromosome1_parent1,␣
 ↪chromosome2_parent2, chromosome3_parent1))
    chromosome_enfant2 = np.concatenate((chromosome2_parent1,␣
 ↪chromosome1_parent2, chromosome3_parent2))
    chromosome_enfant3 = np.concatenate((chromosome1_parent1,␣
 ↪chromosome1_parent2, chromosome3_parent1))
    chromosome_enfant4 = np.concatenate((chromosome2_parent2,␣
 ↪chromosome2_parent2, chromosome3_parent2))
    population_enfants.append(chromosome_enfant1)
    population_enfants.append(chromosome_enfant2)
    population_enfants.append(chromosome_enfant3)
    population_enfants.append(chromosome_enfant4)
    return population_enfants
```

[30]:
```
# Amélioration de la méthode de croisement : croisement à masque

def croisement_masque (population, prix_actions, rentabilites, budget,␣
 ↪limite_budget):
    population = selection_tournoi(population, prix_actions, rentabilites,␣
 ↪budget, limite_budget)
    population_enfants = []
    # Générer un masque
    masque = np.random.randint(0, 2, size=len(population[0]))
```

```
    enfant = population[0] * masque + population[1] * (1 - masque)
    population_enfants.append(enfant)
    return population_enfants
```

[31]:
```
def mutation_deplacement(indi, bud, lim_bud):
    # Faire avance de 2 casses deux objets
    taille_individu = len(indi)
    individu_mute = indi
    indice = np.random.randint(0, taille_individu)
    if indice < taille_individu - 2:
        individu_mute[indice], individu_mute[indice + 2] = individu_mute[indice␣
 ↪+ 2], individu_mute[indice]
    else:
        individu_mute[indice], individu_mute[indice - 2] = individu_mute[indice␣
 ↪- 2], individu_mute[indice]
    individu_mute_corrige = correction_individu(individu_mute, bud, lim_bud)
    return individu_mute_corrige


def mutation_population_deplacement(population, probabilite_mutation, budget,␣
 ↪limite_budget, rentabilites, prix_actions):
    for i in range(len(population)):
        if np.random.random() < probabilite_mutation:
            population[i] = mutation_deplacement(population[i], budget,␣
 ↪limite_budget)
    return population
```

J'ai fait le choix de me concentrer sur la variation de la probabilité de mutation, les méthodes de croisement et de sélection.

[32]:
```
def genetique_tournoi_2_point(population, prix_actions, rentabilites, budget,␣
 ↪limite_budget, probabilite_mutation,
                              nombre_iteration):
    historique_fitness = []
    max_fitness = []
    for i in range(nombre_iteration):
        enfants = croisement_2_points(population, prix_actions, rentabilites,␣
 ↪budget, limite_budget)
        enfants_corrige = correction_population(enfants, budget, limite_budget)
        population = np.concatenate((population, enfants_corrige))
        population = mutation_population_deplacement(population,␣
 ↪probabilite_mutation, budget, limite_budget, rentabilites, prix_actions)
        population = population_taille_initiale(population, prix_actions,␣
 ↪rentabilites, budget,
                                                limite_budget)  # On la trie et␣
 ↪on la remet à la taille initiale
```

```
        fitness_meilleur_individu = fitness(population[0], prix_actions,␣
↪rentabilites, budget, limite_budget)
        historique_fitness.append(fitness_meilleur_individu)
        max_historique_fitness = max(historique_fitness)
        if max_historique_fitness > fitness_meilleur_individu:
            max_fitness.append(max_historique_fitness)
        else:
            max_fitness.append(fitness_meilleur_individu)
    return population, historique_fitness, max_fitness
```

[33]:
```
def genetique_tournoi_masque(population, prix_actions, rentabilites, budget,␣
↪limite_budget, probabilite_mutation,
                             nombre_iteration):
    historique_fitness = []
    max_fitness = []
    for i in range(nombre_iteration):
        enfants = croisement_masque(population, prix_actions, rentabilites,␣
↪budget, limite_budget)
        enfants_corrige = correction_population(enfants, budget, limite_budget)
        population = np.concatenate((population, enfants_corrige))
        population = mutation_population_deplacement(population,␣
↪probabilite_mutation, budget, limite_budget, rentabilites, prix_actions)
        population = population_taille_initiale(population, prix_actions,␣
↪rentabilites, budget,
                                                limite_budget)  # On la trie et␣
↪on la remet à la taille initiale
        fitness_meilleur_individu = fitness(population[0], prix_actions,␣
↪rentabilites, budget, limite_budget)
        historique_fitness.append(fitness_meilleur_individu)
        max_historique_fitness = max(historique_fitness)
        if max_historique_fitness > fitness_meilleur_individu:
            max_fitness.append(max_historique_fitness)
        else:
            max_fitness.append(fitness_meilleur_individu)
    return population, historique_fitness, max_fitness
```

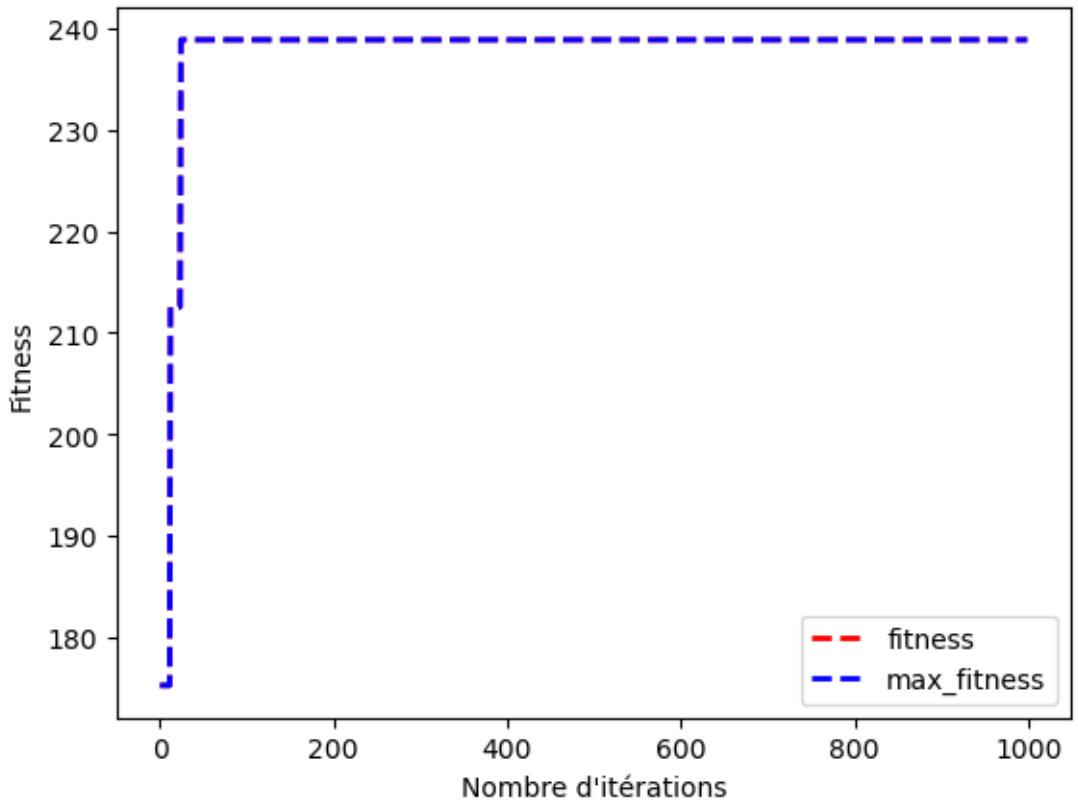
Résultats de l'amélioration de la méthode de croisement 2 points + selection par tournoi :

5.2 Amélioration de la méthode de croisement 2 points + selection par tournoi

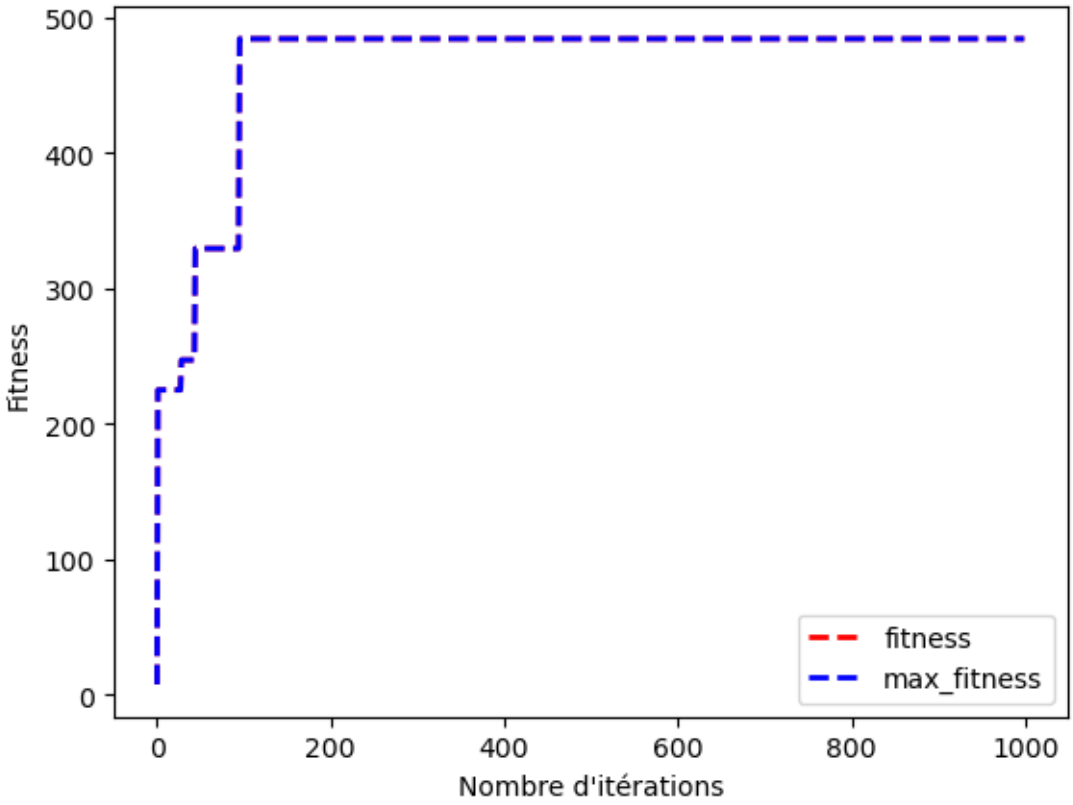
[34]:
```
probabilite_mutation = 0.05 # 20%

for k in range(4):
    # Charger la matrice depuis le fichier CSV
    nom_fichier = f'./actions/population_action{k + 1}.csv'
    population = np.loadtxt(nom_fichier, delimiter=",")
    nom_prix = f'./actions/prix_action{k + 1}.csv'
```

```
    prix_actions = np.loadtxt(nom_prix, delimiter=",")
    nom_rentabilites = f'./actions/rentabilites_action{k + 1}.csv'
    rentabilites = np.loadtxt(nom_rentabilites, delimiter=",")



    # Algorithmes génétiques
    population, historique_fitness, max_fitness =␣
↪genetique_tournoi_2_point(population, prix_actions, rentabilites, budget,
                                                                        ␣
↪limite_budget, probabilite_mutation,
                                                                        ␣
↪nombre_iteration)

    print(f"Pour le fichier {nom_fichier} et l'amélioration de la méthode de␣
↪croisement 2 points + selection par tournoi :")

    # Affichage des résultats
    affichage(population, historique_fitness, max_fitness)
```

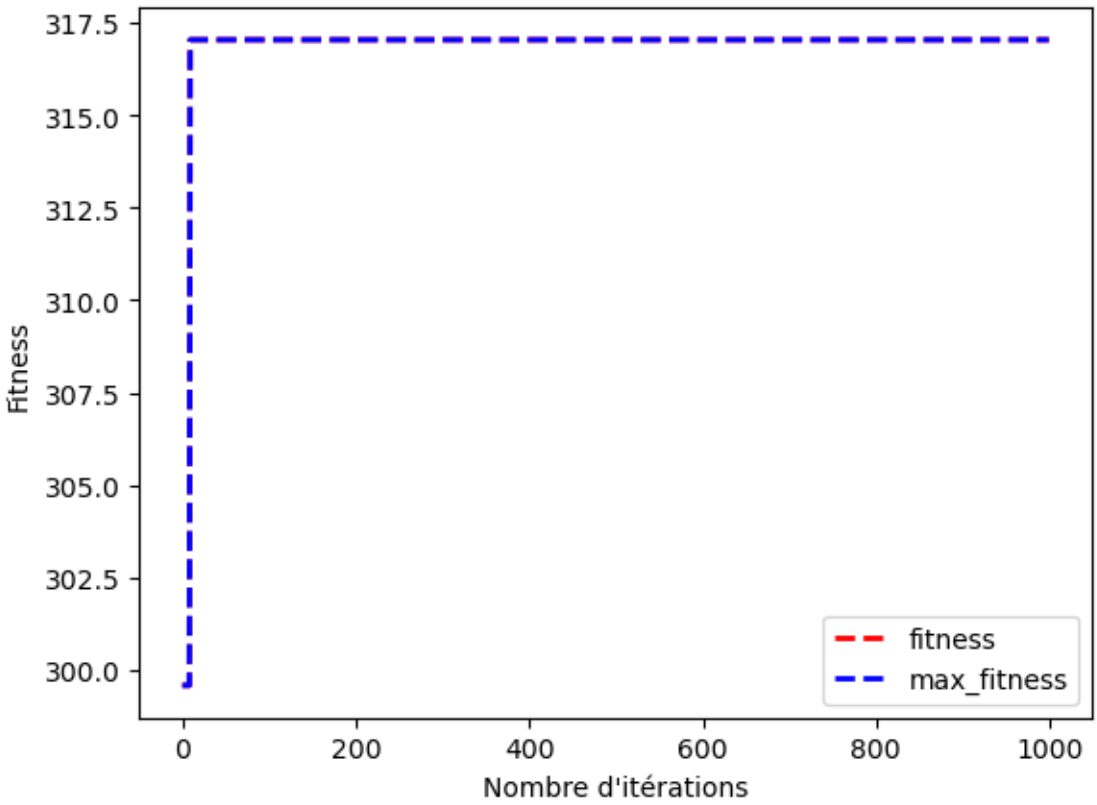
Pour le fichier ./actions/population_action1.csv et l'amélioration de la méthode de croisement 2 points + selection par tournoi :



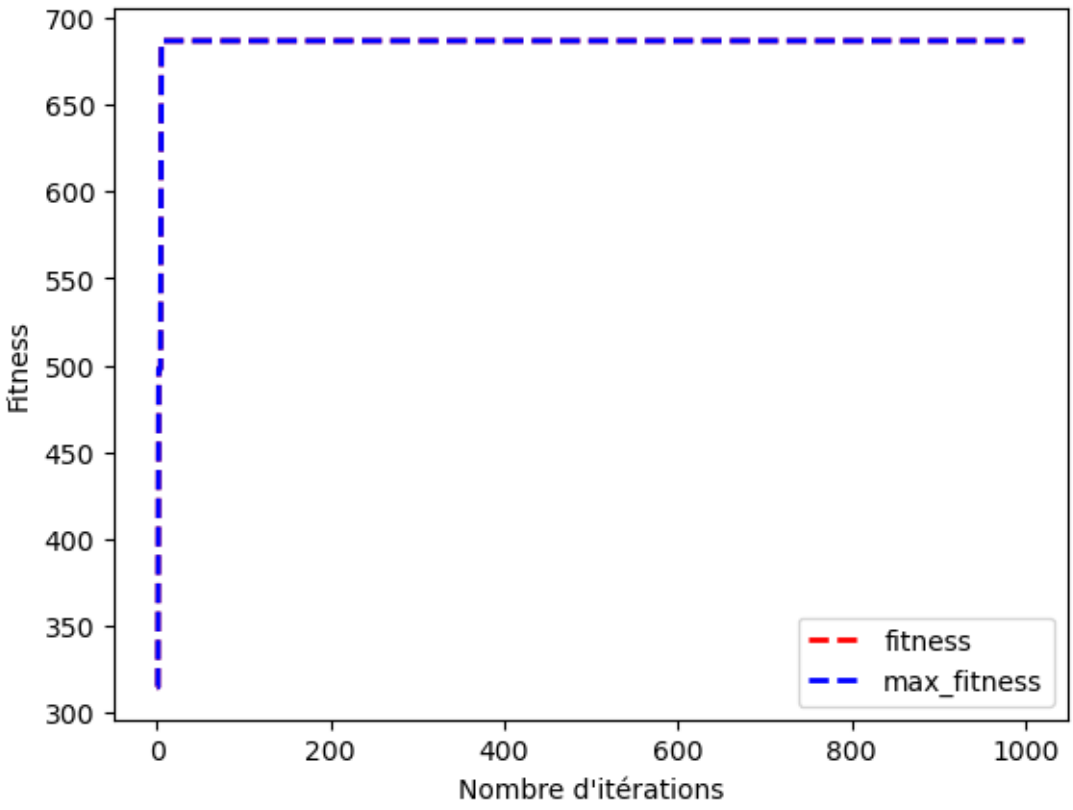
Pour le fichier ./actions/population_action2.csv et l'amélioration de la méthode de croisement 2 points + selection par tournoi :



Pour le fichier ./actions/population_action3.csv et l'amélioration de la méthode de croisement 2 points + selection par tournoi :



Pour le fichier ./actions/population_action4.csv et l'amélioration de la méthode de croisement 2 points + selection par tournoi :



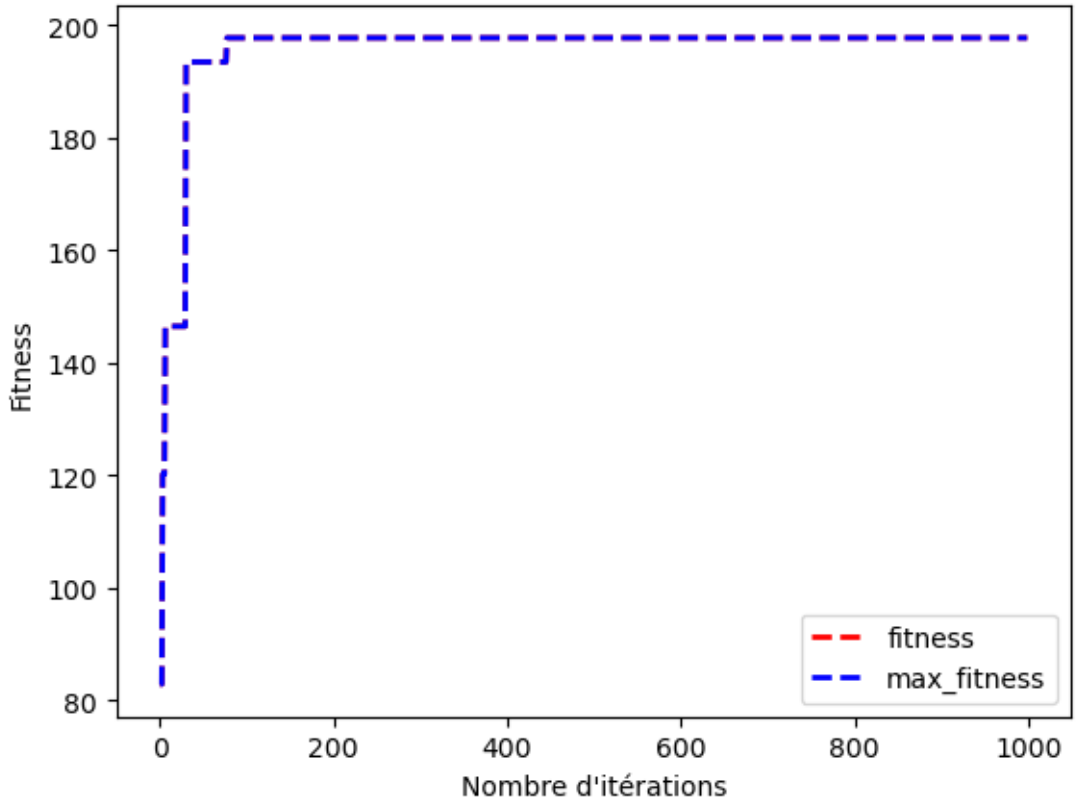
Résultats de l'amélioration de la méthode de croisement masque + selection par tournoi :

5.3 Amélioration de la méthode de croisement masque + selection par tournoi

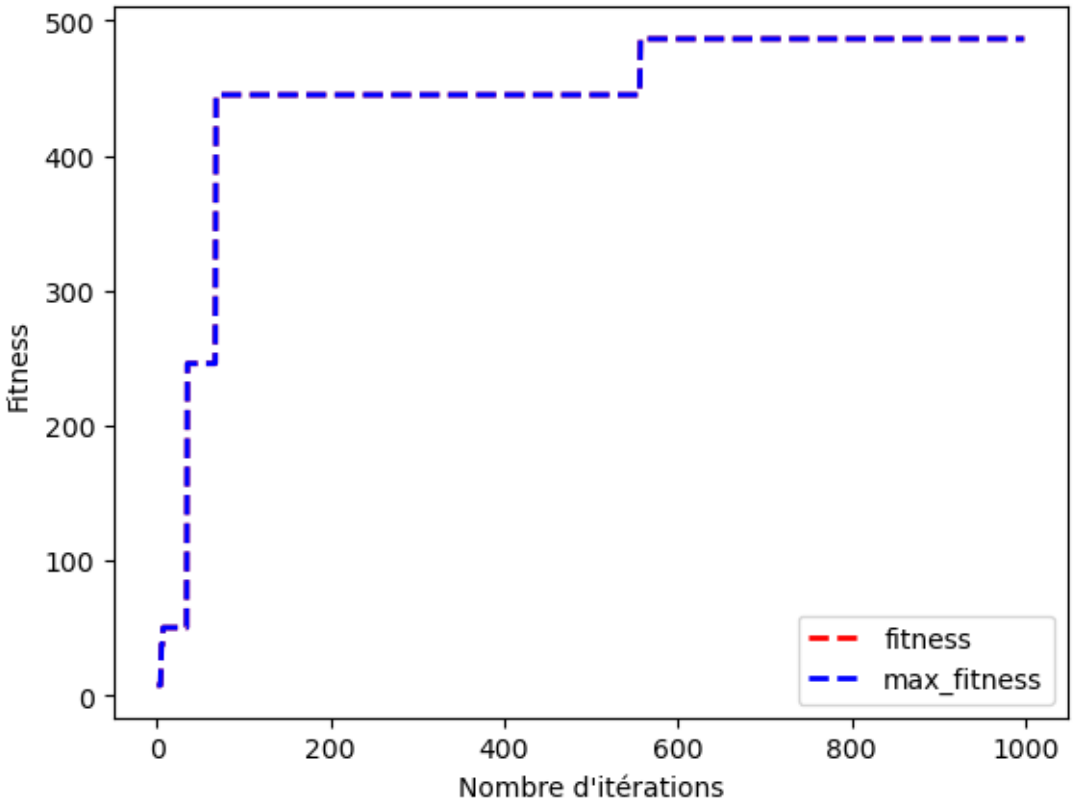
[35]:
```
for k in range(4):
    # Charger la matrice depuis le fichier CSV
    nom_fichier = f'./actions/population_action{k + 1}.csv'
    population = np.loadtxt(nom_fichier, delimiter=",")
    nom_prix = f'./actions/prix_action{k + 1}.csv'
    prix_actions = np.loadtxt(nom_prix, delimiter=",")
    nom_rentabilites = f'./actions/rentabilites_action{k + 1}.csv'
    rentabilites = np.loadtxt(nom_rentabilites, delimiter=",")



    # Algorithmes génétiques
    population, historique_fitness, max_fitness =␣
↪genetique_tournoi_masque(population, prix_actions, rentabilites, budget,
                                                                        ␣
↪limite_budget, probabilite_mutation,
```

```
↪nombre_iteration)

  print(f"Pour le fichier {nom_fichier} et l'amélioration de la méthode de␣
↪croisement masque + selection par tournoi :")

  # Affichage des résultats
  affichage(population, historique_fitness, max_fitness)
```

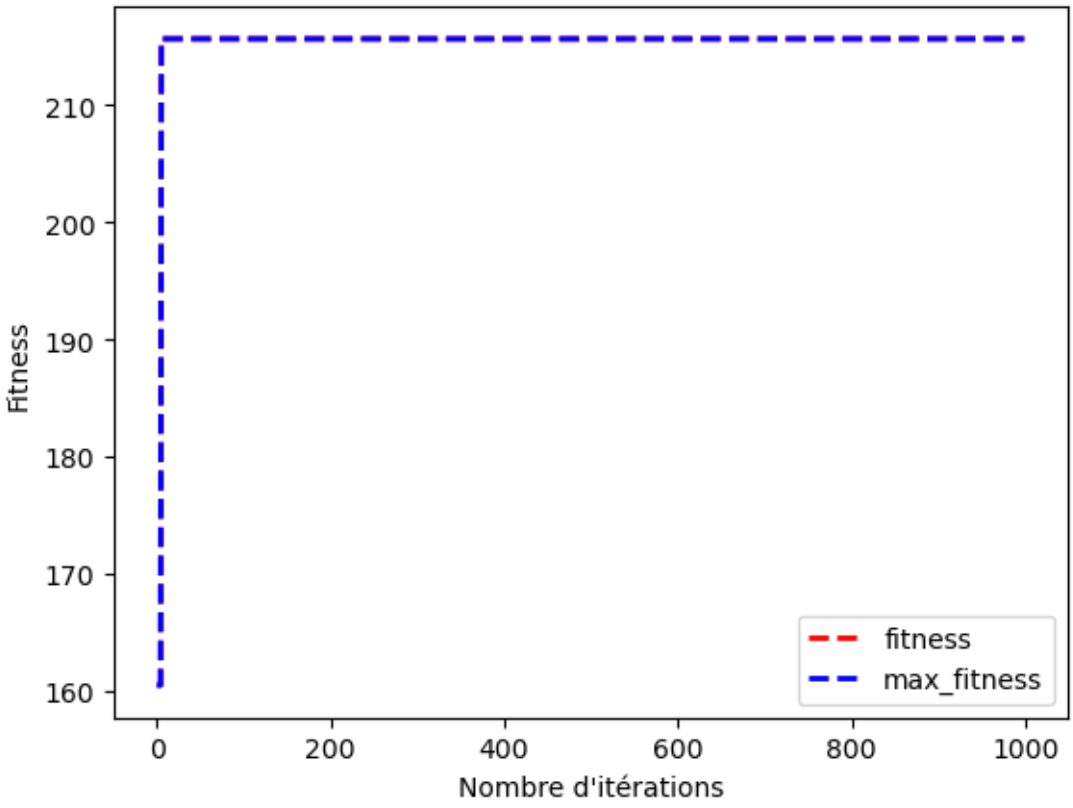
Pour le fichier ./actions/population_action1.csv et l'amélioration de la méthode de croisement masque + selection par tournoi :



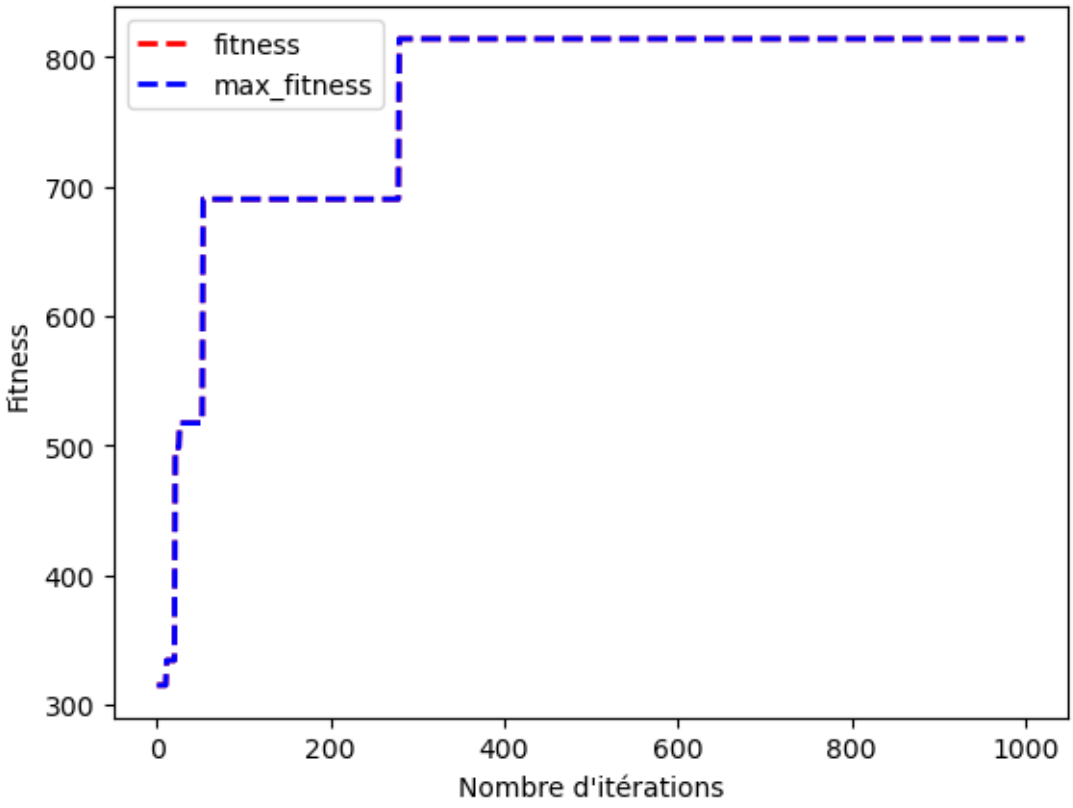
Pour le fichier ./actions/population_action2.csv et l'amélioration de la méthode de croisement masque + selection par tournoi :



Pour le fichier ./actions/population_action3.csv et l'amélioration de la méthode de croisement masque + selection par tournoi :



Pour le fichier ./actions/population_action4.csv et l'amélioration de la méthode de croisement masque + selection par tournoi :



5.4 Pour conclure :

- Nous pouvons voir que la méthode de croisement à 2 points est plus efficace que la méthode de croisement à masque avec les données que nous avons générées.
- Mais nous pouvons remarquer que les améliorations que nous avons apportées ne sont pas forcément meilleures que les méthodes de base.