Université des Sciences et de la Technologie
Houari Boumediene Faculté Informatique

TP Métaheuristiques

Résolution des N-Reines avec les métaheuristiques

Etudiants :
Manel OUCHAR
Zakaria YOUSFI
SII G03

10 mai 2023

Table des matières

Table des figures

Chapitre 1

Introduction Générale

Le problème de n-reines est un problème classique de l'informatique combinatoire qui consiste à placer **n** reines sur un échiquier de taille **n x n**, de sorte qu'aucune reine ne soit menacée par une autre. Bien que ce problème puisse être facilement résolu pour des valeurs de n faibles, il devient rapidement difficile pour des valeurs plus grandes.

Pour résoudre ce problème, de nombreuses approches ont été développées, dont les méthodes métaheuristiques qui se sont avérées être une approche efficace. Ce sont des algorithmes d'optimisation basés sur des principes heuristiques qui peuvent être utilisés pour explorer efficacement l'espace de recherche du problème et trouver une solution satisfaisante.

Parmi les méthodes métaheuristiques les plus couramment utilisées pour résoudre le problème de n-reines, on trouve les algorithmes génétiques (GA) et l'optimisation par essaims particulaires (PSO).

Les algorithmes génétiques utilisent une approche évolutionnaire pour trouver une solution optimale en créant des générations de solutions candidates et en appliquant des opérations de sélection, de croisement et de mutation pour générer de nouvelles solutions. D'un autre côté, l'optimisation par essaim de particules est basée sur un modèle d'essaim où chaque particule représente une solution candidate, et les particules sont mises à jour itérativement en fonction de leur propre meilleure solution et de la meilleure solution globale du groupe.

Dans ce rapport, nous nous concentrerons sur la résolution du problème de n-reines à travers les méthodes métaheuristiques GA et PSO. Nous étudierons l'efficacité de ces méthodes pour résoudre le problème de n-reines et nous comparerons les résultats obtenus pour déterminer quelle méthode est la plus appropriée pour résoudre ce problème.

Chapitre 2

Modélisation du problème

2.1 Espace de solution

L'espace de solution pour ce problème est constitué de toutes les configurations possibles pour placer les n reines sur l'échiquier de taille n x n. Chaque configuration est représentée par un vecteur de taille n, où chaque élément i représente la ligne dans laquelle la reine est placée sur la colonne i. Par exemple, pour le problème des 8-reines, le vecteur de solution [1, 3, 4, 0, 5, 2, 7, 6] signifie que la reine sur la première colonne est placée dans la deuxième ligne, la reine sur la deuxième colonne est placée dans la quatrième ligne, etc...

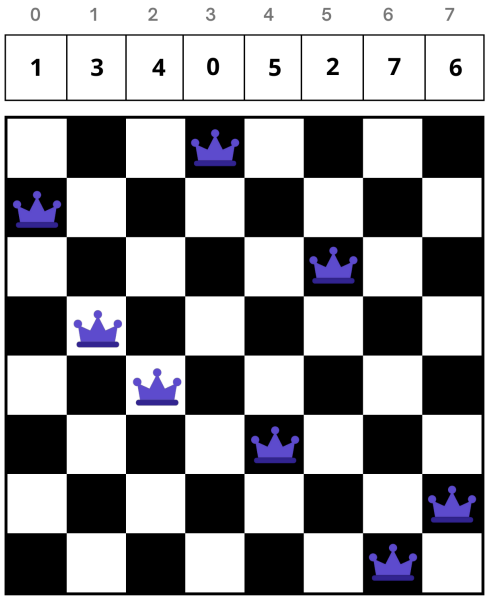

FIGURE 2.1 – Modélisation d'une solution de 8 reines.

Ici pour 8-reines, le nombre de solutions est de 92 solutions uniques possibles. Pour des tailles de problèmes plus grandes, le nombre de solutions augmente rapidement et l'espace de solution explose. Par conséquent, il est nécessaire d'utiliser des approches métaheuristiques pour trouver des solutions efficaces pour des tailles de problèmes plus grandes.

n	Size of solution space (*n*!)	Number of solutions
1	1	1
2	2	0
3	6	0
4	24	2
5	120	10
6	720	4
7	5040	40
8	40320	92
9	362880	352
10	3628800	724
11	39916800	2680
12	479001600	14200
13	6227020800	73712
14	87178291200	365596
15	1307674368000	2279184
16	20922789888000	14772512
17	355687428096000	95815104
18	6402373705728000	666090624
19	121645100408832000	4968057848
20	2432902008176640000	39029188884
21	51090942171709440000	314666222712
22	1124000727777607680000	2691008701644
23	25852016738884976640000	24233937684440
24	620448401733239439360000	227514171973736
25	15511210043330985984000000	2207893435808352
26	403291461126605635584000000	22317699616364044

FIGURE 2.2 – Espace de solution.

2.2 Fonction Fitness

2.2.1 description

La fonction fitness est une fonction qui évalue la qualité d'une solution en solution candidate et détermine à quelle distance elle se trouve de la solution optimale. Pour le problème des n-reines, la fonction de fitness est définie comme le nombre de paires de reines menacées. Plus le nombre de reines menacées est faible, plus la solution est optimale. Si la solution optimale est atteinte, la fonction de fitness retourne 0.

En effet, l'espace de solution pour ce problème est complexe et énorme, mais une fonction de fitness bien définie peut aider à guider la recherche des solutions optimales à travers des approches métaheuristiques.

2.2.2 Pseudo code

Ci-dessous le pseudo-code de cette fonction :

```
Fonction fitness(solution, N):
conflits = 0;
pour i de 0 a N-1 faire:
    pour j de i+1    N-1 faire:
        // Verification de la conflit horizontale
        si (solution[i] == solution[j]) alors:
            conflits = conflits + 1;
        // Verification de la conflit diagonale
        si (abs(i - j) == abs(solution[i] - solution[j])) alors:
            conflits = conflits + 1;
    fait;
fait;
retourner conflits;
```

Listing 2.1 – Pseudo-code de la fonction fitness.

Chapitre 3

Les Métaheuristiques

3.1 Algorithme Génétique

3.1.1 Description

L'algorithme génétique est un algorithme de recherche basé sur les mécanismes de la sélection naturelle et de la génétique[1], basé sur la théorie de Darwin sur l'évolution des espèces. Celle ci repose sur trois principes : le principe de variation, le principe d'adaptation et le principe d'hérédité[6].

- **Le principe de variation** : Chaque individu au sein d'une population est unique. Ces différences, plus ou moins importantes, vont être décisives dans le processus de sélection.
- **Le principe d'adaptation** : Les individus les plus adaptés à leur environnement atteignent plus facilement l'âge adulte. Ceux ayant une meilleure capacité de survie pourront donc se reproduire davantage.
- **Le principe d'hérédité** : Les caractéristiques des individus doivent être héréditaires pour pouvoir être transmises à leur descendance. Ce mécanisme permettra de faire évoluer l'espèce pour partager les caractéristiques avantageuse à sa survie[6].

Ce paradigme, associé avec la terminologie de la génétique, nous permet d'exploiter les algorithmes génétiques : Nous retrouvons les notions de Population, d'Individu, de Chromosome et de Gène[6].

- La population est l'ensemble des solutions envisageables.
- L'individu représente une solution.

— Le Chromosome est une composante de la solution.
— Le Gène est une caractéristique, une particularité.

Avec ces notions, nous obtenons trois opérateurs d'évolution :

1. **La sélection** : consiste à choisir les individus les mieux adaptés afin d'avoir une population de solution la plus proche de converger vers l'optimum global. Cet opérateur est l'application du principe d'adaptation de la théorie de Darwin. Il existe plusieurs techniques de sélection : sélection par tournoi, roulette, elitiste...etc[6]. Voici un exemple avec des individus en représentation binaire une fois la sélection effectuée :

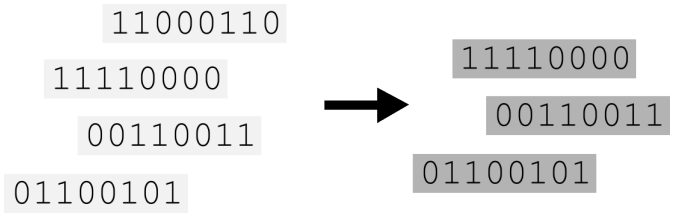

FIGURE 3.1 – Exemple de sélection dans l'algorithme génétique.

2. **Le croisement** : est le résultat obtenue lorsque deux chromosomes partage leurs particularités. Celui-ci permet le brassage génétique de la population et l'application du principe d'hérédité de la théorie de Darwin. Il existe deux méthodes de croissement : simple ou double enjambement[6]. Voici un exemple avec un simple croisement :

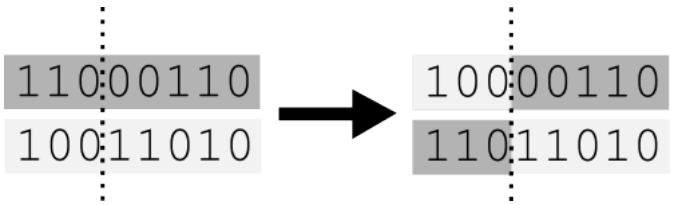

FIGURE 3.2 – Exemple de croisement dans l'algoriehme génétique.

3. **La mutation** : consiste à altérer un gène dans un chromosome selon un facteur de mutation. Ce facteur est la probabilité qu'une mutation soit effectuée sur un individu. Cet opérateur est l'application du principe de variation de la théorie de Darwin et permet, par la même occassion, d'éviter une convergence prématurée de l'algorithme vers un extremum local[6].
   Voici un exemple de mutation sur un individu ayant un seul chromosome :

3.1.2 Fonctionnement

Les principes de bases étant expliqués, voici le fonctionnement de l'algorithme géné-

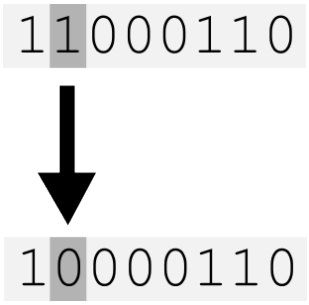

FIGURE 3.3 – Exemple de mutation dans l'algorithme génétique.

tique :

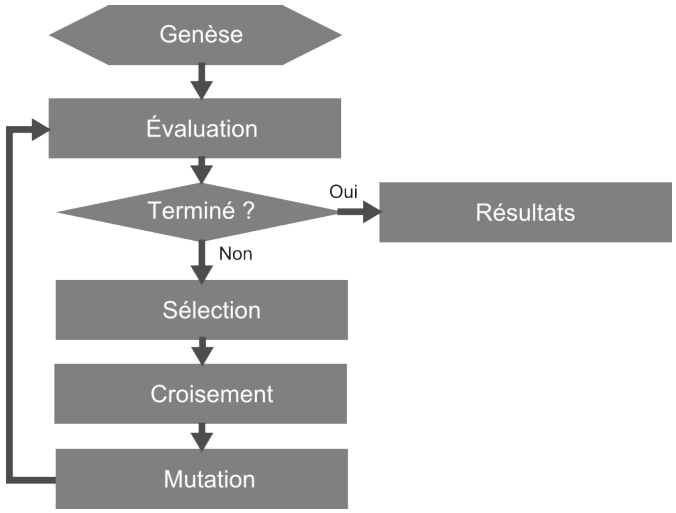

FIGURE 3.4 – Algorithme génétique.

L'algorithme génétique commence par une population initiale de solutions candidates aléatoires. Chaque solution est évaluée en utilisant la fonction fitness définie auparavant.

Une fois que la qualité de chaque solution dans la population est évaluée, l'algorithme commence à générer de nouvelles solutions candidates en utilisant des opérations de sélection, de croisement et de mutation. Les nouvelles solutions candidates sont ensuite évaluées en utilisant la fonction d'évaluation.[1][4]

Ce processus est répété jusqu'à ce qu'un critère d'arrêt soit atteint. Ce critère d'arrêt peut être le nombre maximal d'itérations, la convergence de la solution, ou d'autres critères spécifiques au problème.

3.1.3 Pseudo-code

```
1    Algorithme GA(taille_population, nb_iterations, taux_croisement,
   taux_mutation, taux_selection)
```

```
2
3       // Initialisation de la population
4       pour i de 1 a taille_population faire
5           individu[i] = initialisation_aleatoire();
6
7       // Boucle principale de l'algorithme
8       pour iteration de 1 a nb_iterations faire
9
10          // Evaluation de la population
11          pour i de 1 a taille_population faire
12              fitness[i] = fonction_evaluation(individu[i]);
13
14          // S lection des parents
15          parents = selection_parent(individu, fitness); // X2
16
17          // Croisement des parents
18          enfants = croisement(parents, taille_population);
19
20          // Mutation des enfants
21          enfants = mutation(enfants, taux_mutation);
22
23          // Evaluation de la nouvelle population
24          pour i de 1 a taille_population faire
25              fitness_enfants[i] = fonction_evaluation(enfants[i]);
26
27          // Remplacement de la population
28          individu = Remplacer(parents, enfants);
29      fait;
30  Fin.
```

Listing 3.1 – Pseudo-code de l'algorithme génétique.

3.1.4 Implémentation

3.1.4.1 selection

Nous avons décidé d'utiliser la **sélection élitiste** dans notre algorithme génétique. cette méthode nous permet de conserver les meilleurs individus de chaque génération, ce qui nous assure de ne pas perdre les solutions les plus performantes. De plus, elle permet d'éviter les problèmes de convergence prématurée en préservant une diversité génétique suffisante. Enfin, la sélection élitiste est simple à implémenter et ne nécessite pas de paramétrage complexe, ce qui facilite grandement le développement de notre algorithme.

Voici le pseudo-code :

```
fonction elitistSelection(population, elitismCount, n):
    trier(population) // trier la population par ordre decroissant de fitness
    eliteChromosomes <- prendre les elitismCount meilleurs chromosomes de la population triee
    retourner eliteChromosomes
```

Listing 3.2 – Pseudo-code de la selection

3.1.4.2 crossover

Nous avons choisi d'utiliser le **croisement à un point** dans notre algorithme génétique pour résoudre le problème de n-reines. Cette méthode nous permet de créer des descendants en combinant les caractéristiques des deux parents et en échangeant une partie de leurs génotypes à un point de croisement. Cette approche est particulièrement adaptée à notre problème d'optimisation, car elle permet de maintenir une diversité génétique suffisante et d'éviter une convergence prématurée vers des solutions sous-optimales. De plus, le croisement à un point est une méthode bien établie et largement utilisée dans la littérature pour résoudre des problèmes d'optimisation similaires.

Voici le pseudo-code :

```
fonction croisementUnPoint(parents, taillePopulation, n)
    // Initialisation d'un generateur de nombres aleatoires
    // Boucle sur chaque paire de parents
    pour i de 0    taillePopulation/2 par pas de 2:
        // Selectionne un point de croisement au hasard
        pointCroisement <- random.nextInt(n)
        // Copie les genes des parents jusqu'au point de croisement
        pour j de 0    pointCroisement:
            enfants[i][j] <- parents[i][j]
            enfants[i+1][j] <- parents[i+1][j]
        // Copie les g nes des parents a partir du point de croisement
        pour j de pointCroisement a n:
            enfants[i][j] <- parents[i+1][j]
            enfants[i+1][j] <- parents[i][j]
    // Retourne la nouvelle population d'enfants
    retourner enfants
fin fonction
```

Listing 3.3 – Pseudo-code du croisement

3.1.4.3 mutation

Pour la mutation on a opté pour **la mutation par insertion** dans notre algorithme. Cette méthode consiste à ajouter aléatoirement une nouvelle reine à une position aléatoire du plateau de jeu tout en retirant une reine existante à une position aléatoire. Cette approche est particulièrement adaptée à notre problème d'optimisation, car elle permet d'explorer de nouvelles régions de l'espace de recherche et de sortir des optima locaux.

Voici le pseudo-code :

```
1  Fonction mutation(enfants,mutationRate,populationSize, n) -
2      Pour i de 0 a populationSize/2 - 1 faire
3          Si un nombre aleatoire entre 0 et 1 est inferieur
   mutationRate alors
4              mutationPoint = un entier aleatoire entre 0 et n-1
5              enfants[i][mutationPoint] = un entier aleatoire entre 0
   et n-1
6          Fin Si
7      Fin Pour
8      Retourner enfants
9  Fin Fonction
```

Listing 3.4 – Pseudo-code de la mutation

3.1.4.4 remplacement

Nous avons choisi d'utiliser le **remplacement elitiste**. Cette méthode consiste à remplacer les individus les moins performants de la population par de nouveaux individus générés par le croisement et la mutation. Cependant, nous sauvegardons toujours les meilleurs individus de la population précédente, même s'ils ne sont pas aussi performants que les nouveaux individus, afin de préserver la diversité génétique et d'éviter une convergence prématurée vers des solutions sous-optimales.

Voici le pseudo-code :

```
1  fonction remplacement_elitiste(population, enfants,
   taille_population, n)
2      // On cr e un tableau combine des parents et des enfants
3      combined = nouveau tableau d'entiers de taille (
   taille_population + enfants.taille) x n
4      copie(population, comb, taille_population) // On copie les
   parents dans le tableau combine
5      copie(enfants, comb, enfants.taille, taille_population) // On
   copie les enfants dans le tableau combine
6
```

```
7         // On trie le tableau combine en fonction du fitness de chaque
      individu, en ordre ascendant
8         tri_par_fitness(comb, n)
9
10        // On copie les individus elites dans la nouvelle population
11        nouvelle_population = nouveau tableau d'entiers de taille
      taille_population x n
12        copie_elites(comb, nouvelle_population, taille_population)
13
14        retourne nouvelle_population
15    fin fonction
```

Listing 3.5 – Pseudo-code du remplacement

3.2 Algorithme PSO

3.2.1 Description

L'algorithme PSO (Particle Swarm Optimization) est une métaheuristique d'optimisation qui s'inspire de la biologie pour résoudre des problèmes d'optimisation, développée par *Eberhart* et *Kennedy* en 1995[7]. Elle a été inspirée du comportement social des animaux vivant dans des essaims, notamment des bancs de poissons et des vols groupés d'oiseaux.

Cette méthode repose sur un ensemble d'individus, à l'origine disposés de façon aléatoire, appelées particules et qui se déplacent dans l'espace de recherche. Chacune des particules est considérée comme une solution du problème et possède une position Xid et une vitesse Vid. En outre, chaque particule a une mémoire de sa meilleure position visitée Pid et aussi de celle de son voisinage Pgd.

L'évolution des équations de l'algorithme dans le cas continu est déterminée comme suit :

$$\begin{cases} V_{id}^{t+1} = \omega\, V_{id}^{t} + C_1 R_1 (P_{id} - X_{id}) + C_2 R_2 (P_{gd} - X_{id}) & (1) \\ X_{id}^{t+1} = X_{id}^{t} + V_{id}^{t+1} & (2) \end{cases}$$

On notera que ω désigne le coefficient d'inertie, les coefficients C1 et C2 sont des constantes déterminées de façon empirique en fonction de la relation C1 + C2 4 et enfin, R1 et R2 sont des nombres positifs aléatoires qui suivent une distribution uniforme sur [0,1] [7].

La stratégie de déplacement d'une particule, comme elle est représentée sur la figure ci-dessous, est influencée par les trois composants suivants :

1. **Un élément d'inertie** (ω Vidt) : la particule tend à suivre sa direction actuelle de mouvement ;
2. **Une composante cognitive** (C1R1 (Pid - Xid)) : la particule tend à se déplacer vers la meilleure position par laquelle elle est déjà passée ;
3. **Une composante sociale** (C2R2 (Pgd - Xid)) : la particule a tendance à compter sur l'expérience de son voisinage et donc elle se dirige vers la meilleure position déjà atteinte par ses voisins.

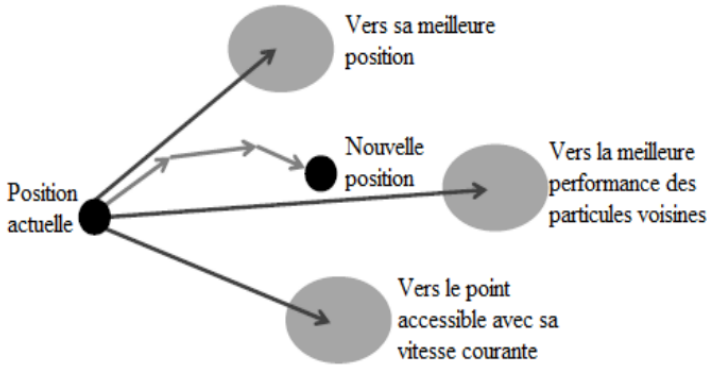

FIGURE 3.5 – Mouvement d'une particule.

3.2.2 Fonctionnement

L'idée de base derrière PSO est de créer une population de particules, chacune représentant une solution possible pour le problème à résoudre. Chaque particule possède une position et une vitesse, qui sont mises à jour à chaque itération de l'algorithme.

Au début de l'algorithme, les positions de chaque particule sont initialisées de manière aléatoire dans l'espace de recherche. La fonction fitness à optimiser est évaluée pour chaque particule, ce qui donne une mesure de la qualité de la solution qu'elle représente.

À chaque itération, les particules sont en mouvement dans l'espace de recherche en fonction de leur vitesse et de leur position actuelle. La vitesse est mise à jour en fonction de la meilleure solution trouvée par chaque particule jusqu'à présent, ainsi que de la meilleure solution trouvée par l'ensemble des particules dans le groupe.

L'objectif de ce processus est de guider les particules vers les meilleures solutions possibles en explorant l'espace de recherche de manière coopérative. L'algorithme continue d'itérer jusqu'à ce qu'un critère d'arrêt soit atteint, tel qu'un nombre maximal d'itérations ou une amélioration suffisamment importante de la qualité de la solution.

3.2.3 Pseudo-code

```
1   Algorithme PSO(nb_particules, nb_iterations, particules)
2
3       // Initialisation aleatoire des positions et vitesses des
    particules
4       pour i := 1 a nb_particules faire
5           pour j := 1 a n faire
6               particules[i].position[j] := Random();
7               particules[i].vitesse[j] := Random();
8
9       // Boucle principale de l'algorithme
10      pour t := 1 a nb_iterations faire
11          // Mise a jour des positions et vitesses de chaque particule
12          pour i := 1 a nb_particules faire
13              pour j := 1 a n faire
14                  particules[i].vitesse[j] := w*particules[i].vitesse[
    j] + c1*Random*(particules[i].meilleure_position[j] - particules[i].
    position[j]) + c2*Random*(meilleure_solution[j] -particules[i].
    position[j]);
15              pour j := 1 a n faire
16                  particules[i].position[j] := particules[i].position[
    j] + particules[i].vitesse[j];
17
18          // Evaluation de la qualite de chaque solution
19          pour i := 1 a nb_particules faire
20              evaluation := fonction_objectif(particules[i].position);
21              si evaluation < particules[i].meilleure_evaluation alors
22                  particules[i].meilleure_position := particules[i].
    position;
23                  particules[i].meilleure_evaluation := evaluation;
24              fsi;
25              si evaluation < meilleure_evaluation alors
26                  meilleure_solution := particules[i].position;
27                  meilleure_evaluation := evaluation;
28  Fin.
```

Listing 3.6 – Pseudo-code de l'algorithme PSO.

Chapitre 4

Expérimentations

4.1 Environnement expérimental

4.1.1 Matériel

L'implémentation de la solution a été réalisée sur un ordinateur ayant les configurations matérielles suivantes :

— Un microprocesseur Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 1.99 GHz.
— Une mémoire de 16,0 Go de RAM.
— Système d'exploitation Windows 11 Famille 64bits.

4.1.2 Logiciel

Pour l'environnement de développement, nous avons opté pour : IntelliJ IDEA 3.3.2022.

4.2 Expérimentation sur les paramètres

Les résultats des différents paramètres ont été obtenus en variant la taille du problème (de 8 à 20) et en prenant la fitness moyenne de 10 exécutions pour chaque taille.

4.2.1 Paramètres de l'algorithme génétique

4.2.1.1 Taille de la population

Pour expérimenter sur la taille de la population, nous avons fixer les autres paramètres comme suit :

— Le nombre maximum de générations = 1000
— Le taux de mutation = 0.5
— Le taux de sélection = 0.5

L'application de notre solution d'algorithme génétique nous a permis de dresser le tableau et le graphe suivants :

GA - Taille de la population	
Taille de la population	**Moyenne fitness score**
100	1,325
200	0,75
300	0,7
400	0,625
500	0,6
600	0,475
700	0,35
800	0,39
900	0,325
1000	0,35
1100	0,325
1200	0,3
1300	0,275
1400	0,25
1500	0,2

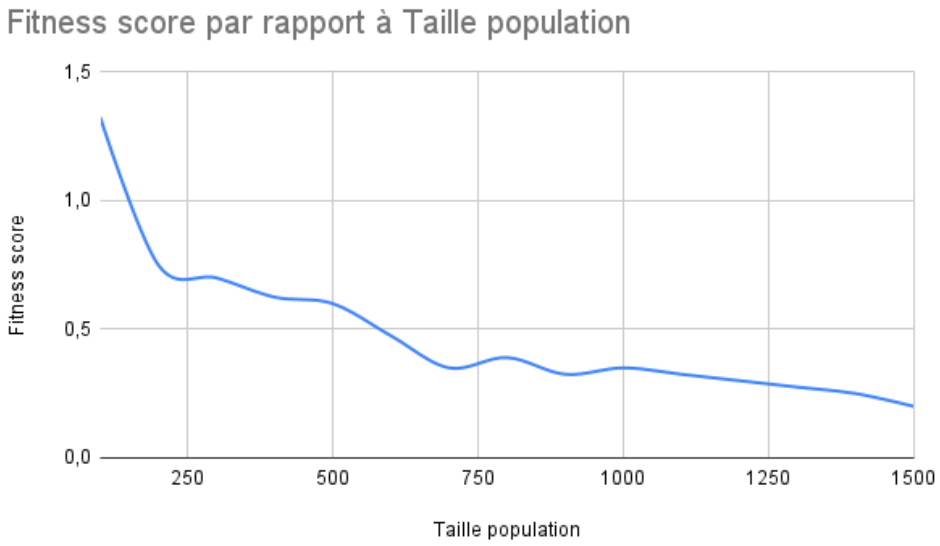

FIGURE 4.1 – Graphe moyenne Fitness score par rapport à Taille population.

- Analyse :

D'après ces résultats, l'augmentation de la taille de la population améliore le fitness score moyen, mais après un certain point, celle ci n'a plus beaucoup d'effet.

Une raison possible de l'amélioration du fitness score, est que l'augmentation de la taille de population permet une exploration plus large de l'espace de solutions, conduisant à une plus grande chance de trouver une bonne solution. Cependant, après un certain point, l'augmentation de la taille de la population peut entraîner une diversité excessive, qui nous éloigne de la solution optimale.

4.2.1.2 Taux de sélection

Pour expérimenter sur le taux de sélection, nous avons fixer les autres paramètres comme suit :

— Le nombre maximum de générations = 1000
— Le taux de mutation = 0.5
— La taille de la population = 100

L'application de notre solution d'algorithme génétique nous a permis de dresser le tableau et le graphe suivants :

GA - Taux de sélection	
Taux de sélection	**Moyenne fitness score**
0,1	1,7
0,2	1,25
0,3	1,2
0,4	1,1
0,5	1,125
0,6	1
0,7	0,925
0,8	0,95
0,9	0,75
1	0,55

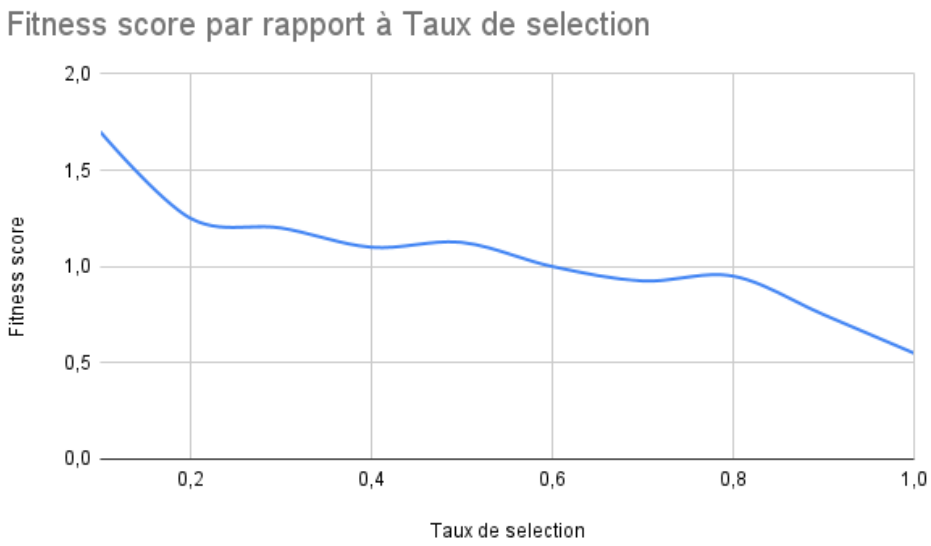

FIGURE 4.2 – Graphe moyenne Fitness score par rapport au Taux de sélection.

- Analyse :

D'après les résultats, nous pouvons observer que le fitness score moyen diminue généralement lorsque le taux de sélection augmente.

Une raison à cela est qu'un taux de sélection plus élevé entraîne plus d'individus. Bien que cela puisse conduire à une convergence plus rapide vers une bonne solution, cela

réduit également la diversité génétique dans la population et peut entraîner l'algorithme à converger prématurément vers une solution non-optimale.

4.2.1.3 Taux de mutation

Pour expérimenter sur le taux de mutation, nous avons fixer les autres paramètres comme suit :

— Le nombre maximum de générations = 1000
— Le taux de sélection = 0.5
— La taille de la population = 100

L'application de notre solution d'algorithme génétique nous a permis de dresser le tableau et le graphe suivants :

GA - Taux de mutation	
Taux de mutation	**Moyenne fitness score**
0,1	0,575
0,2	0,5
0,3	0,35
0,4	0,1
0,5	0,425
0,6	0,875
0,7	0,8
0,8	1
0,9	1,25
1	1,3

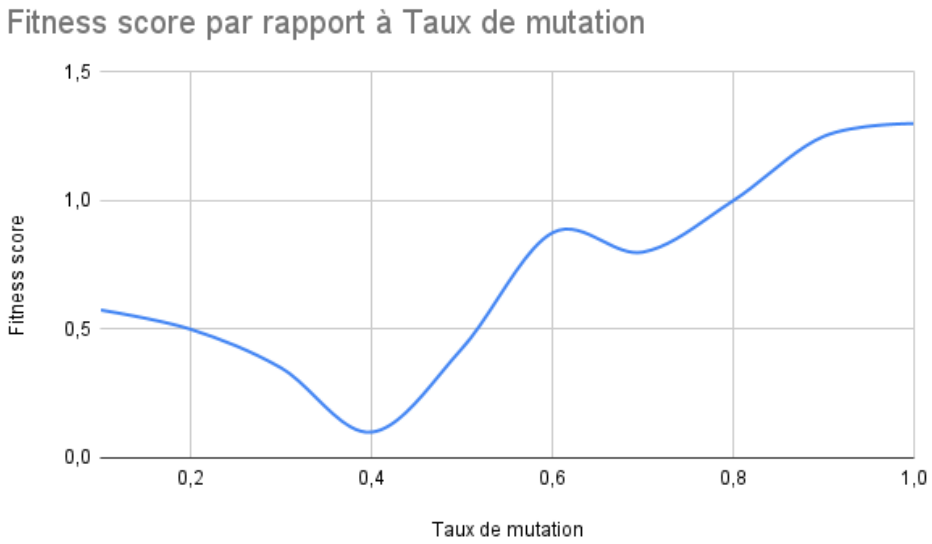

FIGURE 4.3 – Graphe moyenne Fitness score par rapport au Taux de mutation.

- Analyse :

D'après les résultats, on peut observer que le fitness score moyen diminue d'abord à mesure que le taux de mutation augmente, puis augmente après un certain point (0.4 est le taux le plus optimal).

Une raison de la diminution initiale peut être que la mutation permet d'explorer des solutions plus diverses dans l'espace de solutions et peut aider à éviter les optimums locaux. Cependant, à mesure que le taux de mutation augmente, il y a une plus grande probabilité que les mutations perturbent des bonnes solutions déjà trouvées, ce qui peut conduire à une augmentation du score.

4.2.1.4 Nombre de générations

Pour expérimenter sur le nombre de générations, nous avons fixer les autres paramètres comme suit :

— Le taux de sélection = 0.5
— Le taux de mutation = 0.5
— La taille de la population = 100

L'application de notre solution d'algorithme génétique nous a permis de dresser le tableau et le graphe suivants :

GA - Nombre de générations	
Nombre de générations	**Moyenne fitness score**
500	1,55
1000	1,25
1500	1
2000	0,95
2500	1,1
3000	1,075
3500	1
4000	0,95
4500	0,875
5000	0,75
5500	0,6899

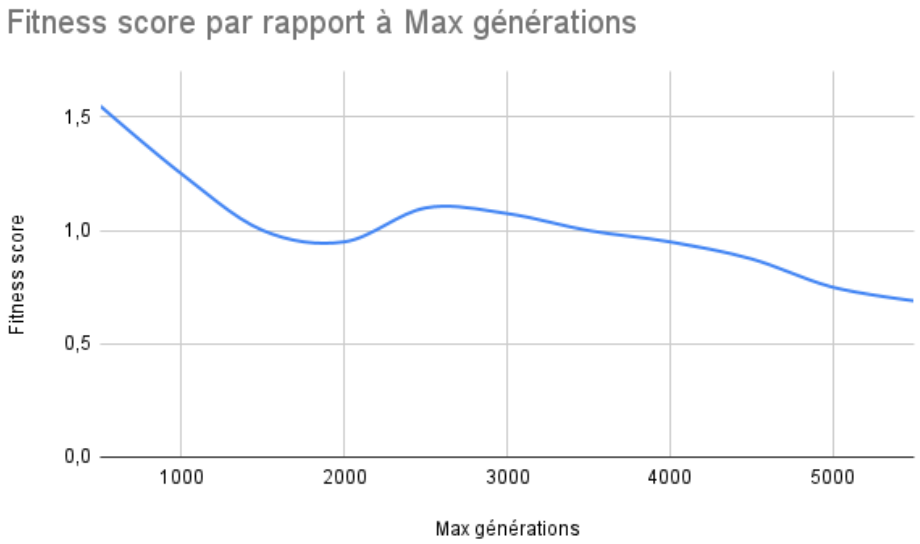

FIGURE 4.4 – Graphe moyenne Fitness score par rapport au Nombre de générations.

- Analyse :

D'après les résultats, on peut observer que le score de fitness moyen diminue généralement à mesure que le nombre de générations augmente.

Cela peut est dû au fait que l'algorithme génétique explore une plus grande variété de solutions au début, mais converge vers des solutions de plus en plus proches de l'optimum à mesure que le nombre de générations augmente. Cependant, cela peut également entraîner une convergence prématurée vers un minimum local, au lieu d'atteindre l'optimum global.

4.2.2 Paramètres de l'algorithme PSO

Pour notre implémentation de l'algorithme PSO, il est important de noter que nous avons fait le choix de mener ses tests en faisant un croisement pour le mouvement des particules, car cette adaption été meilleure que celle avec la vélocité.

Cette adaptation nécessite deux paramètres : le paramètre cognitif c1 et social c2, qui définissent l'intervalle de croisement de l'algorithme tel que : .

Pour chaque position, un nombre aléatoire est généré et comparé au coefficient de croisement c1. Si le nombre aléatoire est inférieur à c1, cela signifie qu'une mutation doit être appliquée. Dans ce cas, un autre nombre aléatoire est généré et comparé au coefficient de croisement c2. Si ce nombre aléatoire est inférieur à c2, la nouvelle position de la particule sera mise à jour avec la meilleure position globale. Sinon, la nouvelle position sera égale à la position actuelle de la particule.

Si le nombre aléatoire est supérieur à c1, cela signifie qu'aucune mutation n'a besoin d'être appliquée et la nouvelle position de la particule sera mise à jour avec sa meilleure position personnelle.

4.2.2.1 Taille de l'essaim

Pour expérimenter sur la taille de l'essaim, nous avons fixer les autres paramètres comme suit :

— Le nombre maximum d'itérations = 1000
— coefficient de croisement c1 = 0.5
— coefficient de croisement c2 = 0.5

L'application de notre solution d'algorithme PSO nous a permis de dresser le tableau et le graphe suivants :

PSO - Taille de l'essaim	
Taille de l'essaim	**Moyenne fitness score**
100	2,89
250	2,8
500	2,825
750	2,375
1000	1,775
1250	1,6
1500	1,75
1750	1,35
2000	1,39
2250	1,28
2500	1,3

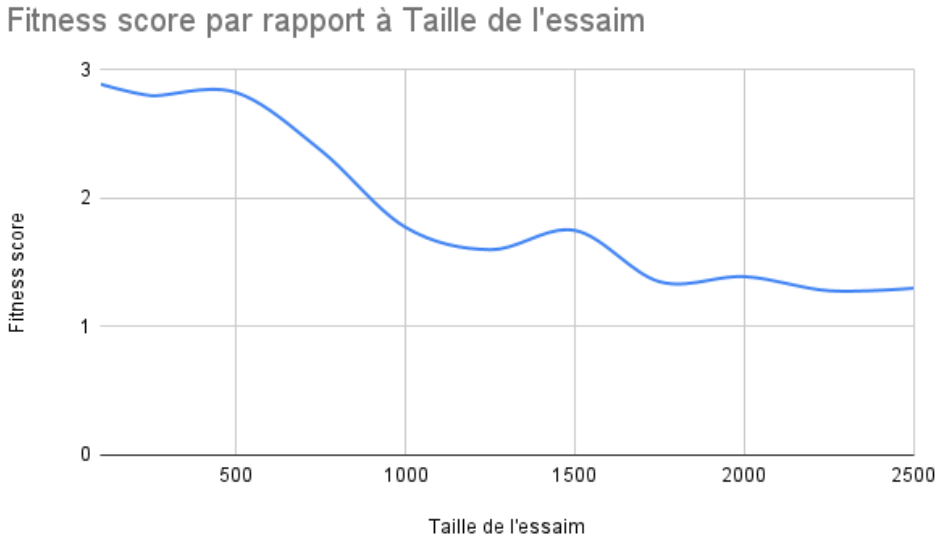

FIGURE 4.5 – Graphe moyenne Fitness score par rapport à Taille de l'essaim.

- Analyse :

Les résultats indique qu'augmenter la taille de l'essaim améliore le score de fitness moyen, mais après un certain point, l'augmentation de la taille de l'essaim n'a pas beaucoup d'effet.

Une raison de l'amélioration du score de fitness moyen avec l'augmentation de la taille de l'essaim pourrait être qu'un essaim plus grand permet une exploration plus grande de l'espace de solutions, conduisant à une plus grande chance de tomber sur une bonne solution. Cependant, après un certain point, l'augmentation de la taille de l'essaim peut entraîner trop de diversité, rendant plus difficile la convergence des particules vers une bonne solution.

4.2.2.2 Nombre d'itérations

Pour expérimenter sur le nombre d'itérations, nous avons fixer les autres paramètres comme suit :

— Taille de l'essaim = 500
— coefficient de croisement c1 = 0.5
— coefficient de croisement c2 = 0.5

L'application de notre solution d'algorithme PSO nous a permis de dresser le tableau et le graphe suivants :

PSO - Nombre d'itérations	
Nombre d'itérations	**Moyenne fitness score**
500	3,575
1000	3,25
1500	3,1
2000	3,175
2500	2,5
3000	2,1
3500	2,75
4000	2,5
4500	2,325
5000	2,525
5500	1,899999

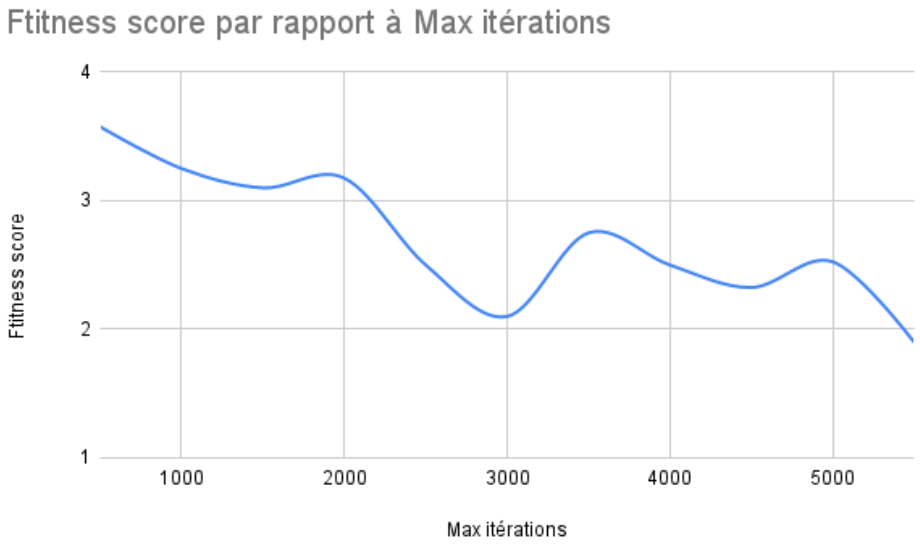

FIGURE 4.6 – Graphe moyenne Fitness score par rapport au Nombre d'itérations.

- Analyse :

D'après les résultats, on peut observer que le score de fitness moyen diminue généralement à mesure que le nombre d'itérations augmente.

L'une des raisons possibles est que les particules de l'algorithme PSO explore une plus grande variété de solutions au début, mais convergent vers des solutions de plus en plus proches de l'optimum à mesure que le nombre d'itérations augmente. Cependant, cela peut également entraîner une convergence prématurée vers un minimum local, au lieu d'atteindre l'optimum global.

4.2.2.3 coefficient de croisement c1

Pour expérimenter sur le coefficient de croisement c1, nous avons fixer les autres paramètres comme suit :

— Taille de l'essaim = 500
— Nombre d'itérations = 1000
— coefficient de croisement c2 = 0.5

L'application de notre solution d'algorithme PSO nous a permis de dresser le tableau et le graphe suivants :

PSO - coefficient de croisement c1	
coefficient de croisement c1	**Moyenne fitness score**
0,1	2,875
0,2	2,875
0,3	2,825
0,4	3,05
0,5	3,175
0,6	3,125
0,7	3,25
0,8	3,375
0,9	3,55
1	3,875

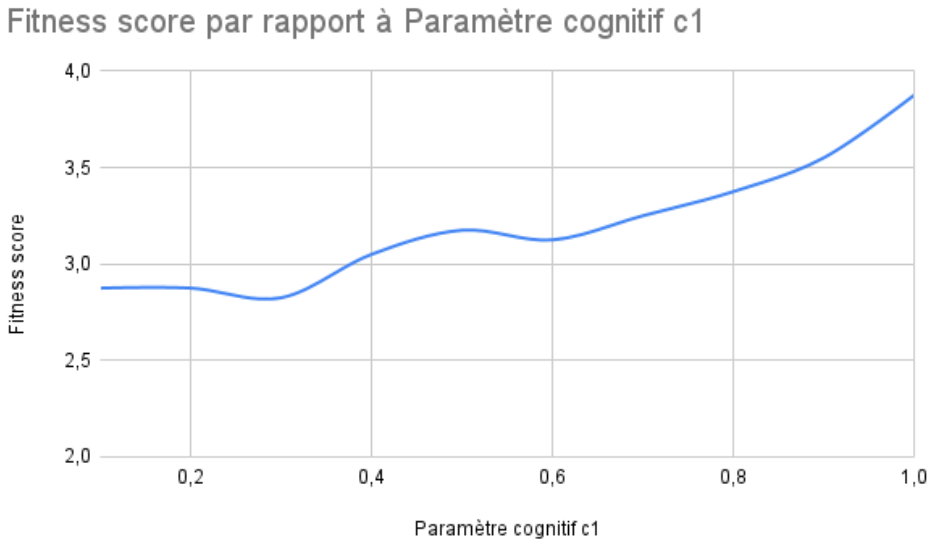

FIGURE 4.7 – Graphe moyenne Fitness score par rapport au coefficient de croisement c1.

- Analyse :

D'après les résultats, on peut voir que le score de fitness moyen augmente lentement lorsque le paramètre cognitif c1 est augmenté de 0,1 à 1,0. Cela indique que les particules dans l'algorithme PSO utilisent plus leur meilleure position personnelle pour guider leur recherche, plutot que se diriger vers la meilleure solution de tout l'essaim, ce qui les éloignent de la solution optimale.

4.2.2.4 coefficient de croisement c2

Pour expérimenter sur le coefficient de croisement c2, nous avons fixer les autres paramètres comme suit :

— Taille de l'essaim = 500
— Nombre d'itérations = 1000
— coefficient de croisement c1 = 0.5

L'application de notre solution d'algorithme PSO nous a permis de dresser le tableau et le graphe suivants :

PSO - coefficient de croisement c2	
coefficient de croisement c2	**Moyenne fitness score**
0,1	3,4
0,2	3,325
0,3	3,375
0,4	3
0,5	3
0,6	2,95
0,7	3,025
0,8	2,975
0,9	2,8
1	2,8

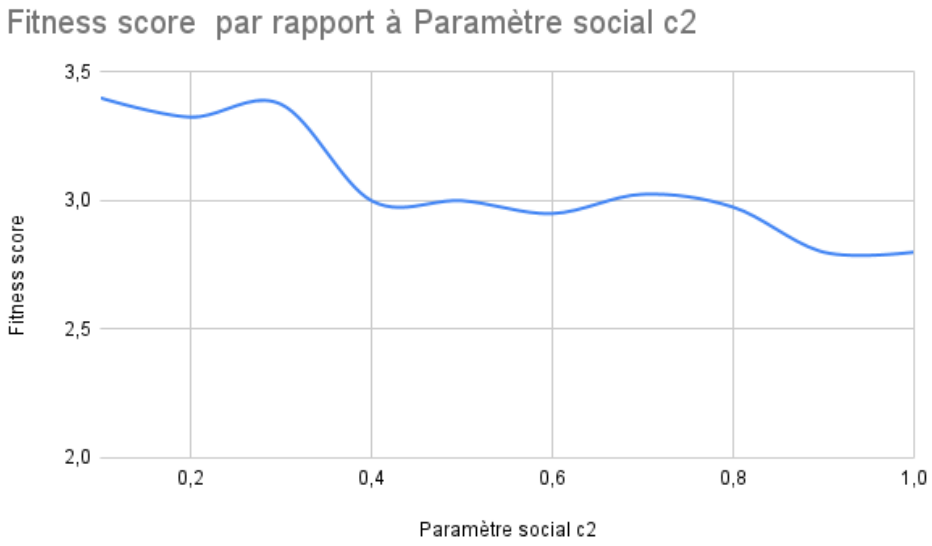

FIGURE 4.8 – Graphe moyenne Fitness score par rapport au coefficient de croisement c2.

- Analyse :

La tendance des résultats indique qu'augmenter c2 initialement améliore le score de fitness moyen, mais après un certain point, une augmentation supplémentaire de c2 n'a pas beaucoup d'effet.

Une des raisons possibles pour l'amélioration initiale du score de fitness moyen avec l'augmentation de c2 est que cela permet aux particules d'utiliser la meilleure solution trouvée dans l'essaim plutot que se focaliser sur la meilleure solution personnelle. Cela peut être bénéfique car ça permet à la particule de suivre une direction plus prometteuse pour atteindre l'optimum global, plutôt que de se limiter à sa propre connaissance.

Cependant, cela peut également conduire à une convergence prématurée si une seule particule (ou un petit groupe de particules) domine l'essaim et attire toutes les autres particules vers une zone locale suboptimale.

4.3 Expérimentation sur la taille du problème

L'exécution de nos algorithmes GA et PSO sur différentes tailles N du problème des n reines (10 exécutions pour chaque taille), nous a permis d'avoir les résultats suivants :

4.3.1 Algorithme génétique

GA		
N	**Moyenne fitness score**	**Temps d'exécution (secondes)**
25	1,2	7,651000023
50	1,8	18,64540024
75	3	47,1052002
100	4	86,63040009
125	6,8	128,6626007
150	10,3	173,0999996
175	17,1	234,8736002
200	27,01	283,3893997

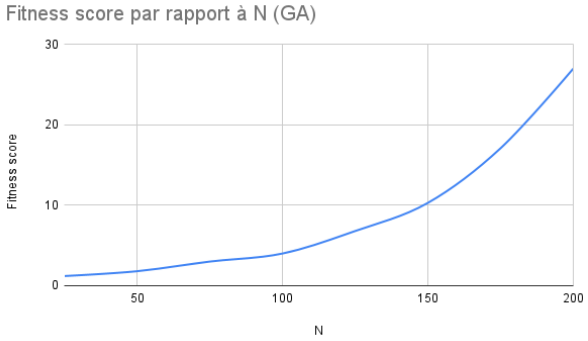

FIGURE 4.9 – Graphe Fitness score par rapport à N pour GA

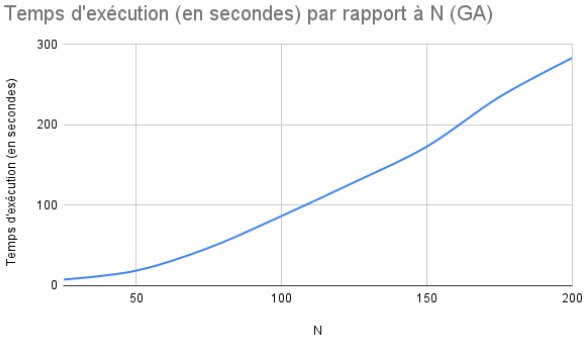

FIGURE 4.10 – Graphe Temps d'exécution (en secondes) par rapport à N pour GA

- Analyse :

La tendance des résultats indique que plus la taille du problème augmente, plus le score de fitness physique moyen et le temps d'exécution augmentent également.

Le raison de cette tendance est que plus la taille du problème augmente, plus l'espace de recherche augmente de manière exponentielle, ce qui rend plus difficile de trouver une bonne solution. Cela signifie que plus d'itérations et d'évaluations de la fonction fitness sont nécessaires pour trouver une bonne solution, ce qui augmente le temps d'exécution.

De plus, avec une taille de problème plus importante, il devient plus difficile d'éviter de rester bloqué dans des optimum locaux, ce qui peut augmenter le fitness score.

4.3.2 Algorithme PSO

PSO		
N	**Moyenne fitness score**	**Temps d'exécution (secondes)**
25	6,6	4,088000107
50	17	8,640400124
75	27,8	15,87360039
100	39,8	25,30399933
125	51,4	38,48939972
150	67,2	55,16999969
175	75,9	76,89999222
200	92,1	95,09999998

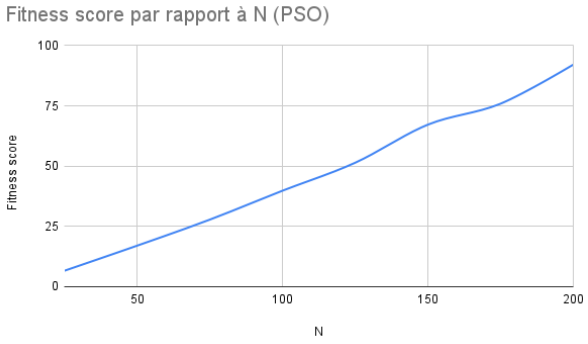

FIGURE 4.11 – Graphe Fitness score par rapport à N pour PSO

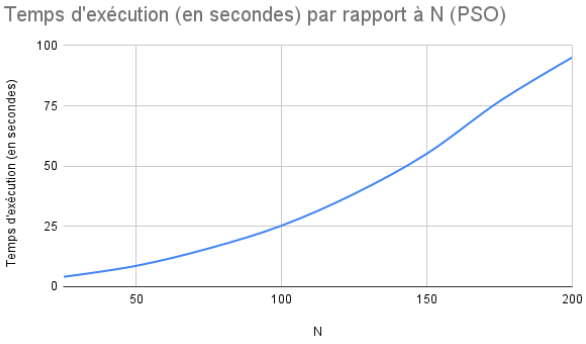

FIGURE 4.12 – Graphe Temps d'exécution (en secondes) par rapport à N pour PSO

- Analyse :

La tendance dans les résultats indique qu'à mesure que la taille du problème augmente, le score de fitness moyen augmente, tandis que le temps d'exécution augmente également.

Le raison de cette tendance est que la taille de l'espace de recherche augmente avec la taille du problème, ce qui rend plus difficile de trouver une bonne solution. Cela signifie que plus d'itérations et d'évaluations de la fonction de fitness sont nécessaires pour trouver une bonne solution, ce qui augmente le temps d'exécution.

De plus, avec une taille de problème plus grande, il devient plus difficile d'éviter de rester coincé dans des optimum locaux, ce qui peut augmenter le fitness score.

4.4 Comparaison des Métaheuristiques

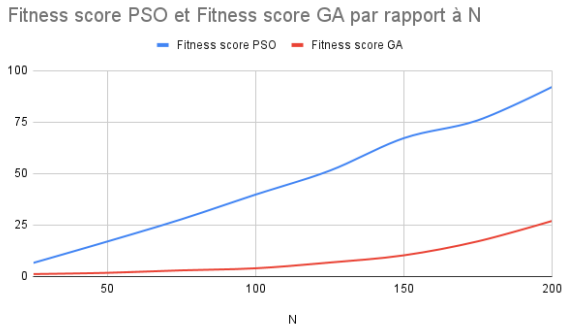

FIGURE 4.13 – Graphe de comparaison du fitness score de PSO et GA par rapport à N

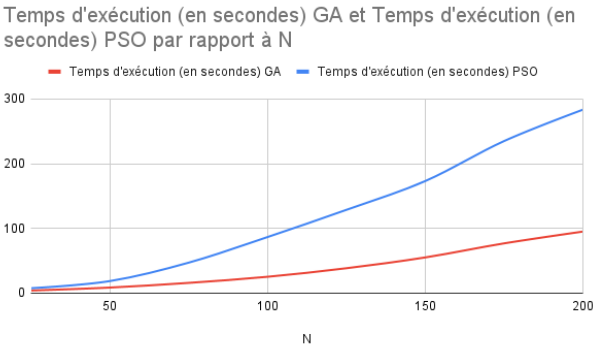

FIGURE 4.14 – Graphe de comparaison du temps d'exécution (en secondes) de PSO et GA par rapport à N

En comparant les deux algorithmes, il semble que l'algorithme génétique ait largement surpassé l'algorithme PSO en termes de score de fitness moyen, avec des scores moins élevés observés pour toutes les tailles de problème testées, surtout les plus grandes.

Une explication possible de la performance supérieure de l'algorithme GA en termes de score de fitness moyen pourrait être attribuée à son approche basée sur des opérateurs génétiques et de la recherche d'un optimum local. Le GA trouve une solution optimale en un temps raisonnable, alors que le PSO nécessite plus de temps pour converger vers une solution optimale.

En revanche, le PSO utilise une approche basée sur la recherche d'un optimum global en modifiant les positions de chaque particule dans l'espace de recherche en fonction de la position du meilleur individu et de la position du meilleur individu de l'ensemble.

Bien que le PSO puisse fonctionner de manière très efficace pour certains problèmes, il peut avoir du mal à trouver une solution optimale pour des problèmes complexes comme le problème des N-Reines.

4.5 Comparaison des approches

Le graphe ci-dessous représente le temps d'exécution (en secondes) des algorithmes DFS, BFS, A*, GA et PSO en fonction de différentes tailles de problème des n reines.

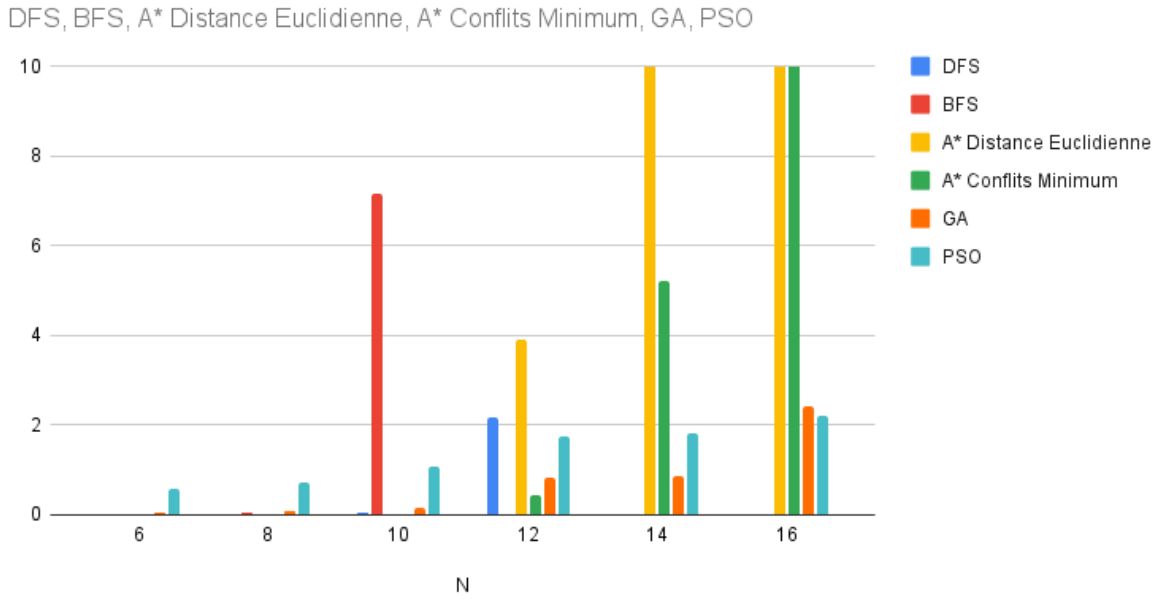

FIGURE 4.15 – Graphe temps d'exécution (en secondes) des algorithmes DFS, BFS, A*, GA et PSO en fonction de N.

- Analyse :

Les résultats montrent que les algorithmes de recherche locale (DFS et BFS) sont très rapides pour résoudre le problème des n reines pour des valeurs de n faibles, mais deviennent très coûteux en temps d'exécution pour des valeurs plus grandes. Les algorithmes de recherche informée (A* Distance Euclidienne et A* Conflits Minimum) ont de meilleures performances que DFS et BFS pour des valeurs plus élevées de n.

Cependant, ils sont tout de même assez lents pour des valeurs de n supérieures à 14. Le GA et le PSO ont des performances relativement similaires pour la résolution du problème des n reines, mais le GA est plus rapide pour des valeurs plus élevées de n et donne des résultats plus optimaux. Cela peut être dû au fait que le GA utilise une approche de recherche globale plutôt qu'une approche de recherche locale comme le PSO.

Chapitre 5

Conclusion générale

À partir résultats obtenus après l'application de divers algorithmes au problème des n reines, on peut conclure que le choix de l'algorithme dépend des exigences et des contraintes du problème.

Les algorithmes de recherche basés sur l'état tels que DFS, BFS et A* sont des algorithmes complets qui peuvent garantir la recherche d'une solution, mais ils peuvent avoir des difficultés avec des tailles de problème plus grandes en raison de leur utilisation de mémoire et l'espace de recherche important du problème.

D'autre part, les algorithmes métaheuristiques tels que GA et PSO peuvent ne pas garantir la recherche de l'optimum global, mais ils peuvent être plus efficaces pour résoudre des problèmes d'optimisation complexes tels que le problème des n reines avec des tailles de problème plus grandes en raison de leur capacité à explorer un grand espace de recherche en moins de temps.

En particulier, l'algorithme GA a surpassé l'algorithme PSO en termes de score de fitness moyen pour le problème des n reines, avec des scores moins élevés observés pour toutes les tailles de problème testées.

En fin de compte, le choix de l'algorithme doit être fait en fonction des exigences spécifiques du problème, telles que la nécessité d'une solution complète, la taille de l'espace de recherche et les ressources informatiques disponibles.

Références

[1] https ://www8.umoncton.ca/umcm-cormier_gabriel/SystemesIntelligents/AG.pdf

[2] https ://igm.univ-mlv.fr/ dr/XPOSE2013/tleroux_genetic_algorithm/fonctionnement.html

[3] https ://www.journaldunet.fr/web-tech/guide-de-l-intelligence-artificielle/1501301-algorithme-genetique-definition-et-apport-en-machine-learning/

[4] https ://fr.wikipedia.org/wiki/Algorithme_g%C3%A9n%C3%A9tique

[5] https ://leria-info.univ-angers.fr/ jeanmichel.richer/polytech_tp_n_reines.php

[6] Kennedy, J. and Eberhart, R. (1995). Particle Swarm Optimization, Proceedings of IEEE International Conference on Neural Networks, pp. 1942-1948, 27th November – 1 December, 1995