21/05/2017

Rapport projet

Problème des n-dames

JEUNE Samuel
GORIA Théo

Table des matières

Introduction

Dans le cadre de l'UE optimisation discrète il nous a été demandé de trouver une solution pour placer n dames sur un échiquier de taille n×n sans que les dames ne puissent se menacer mutuellement et ceci pour n pouvant être très grand. Pour répondre à ce problème nous avons choisi d'utiliser trois métaheuristiques vues en cours : le recuit simulé, la méthode taboue et l'algorithme génétique.

Ce dossier contient la description de nos choix de modélisation, les tests effectués et les résultats obtenus avec ces différents algorithmes.

Modélisation du problème

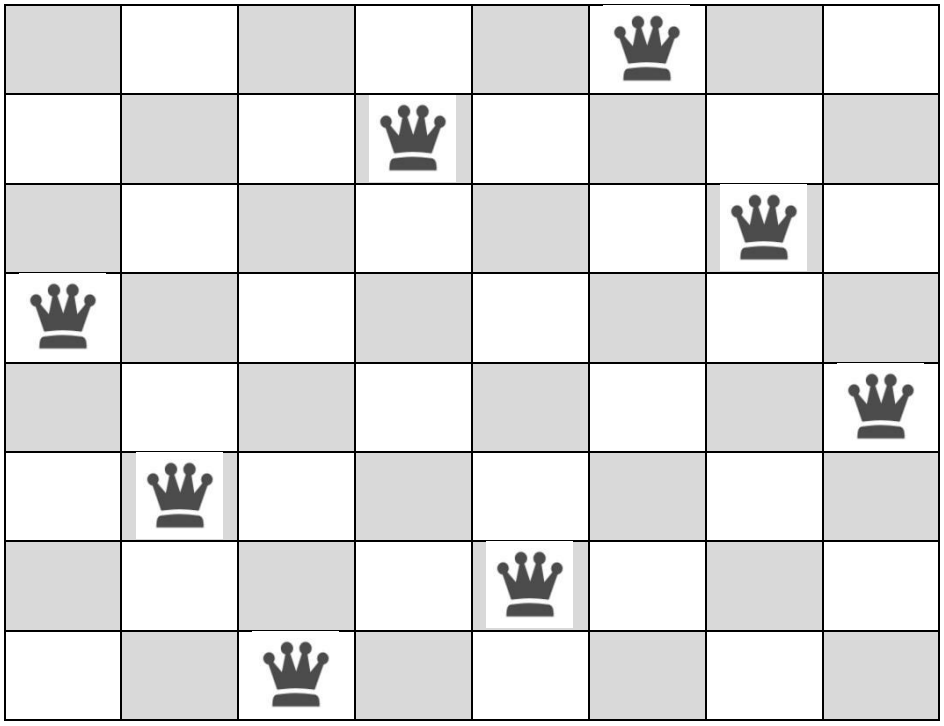
Modélisation de l'échiquier

Nous avons choisi de représenter le plateau du jeu d'échec par un tableau d'entier de taille n (n représentant la taille de l'échiquier). Chaque case de ce tableau représente une colonne de l'échiquier. Les valeurs inscrites dans les cases du tableau représentent les positions verticales des dames dans leurs colonnes respectives en partant du haut et en comptant à partir de 0.

Par exemple ce tableau :

3	5	7	1	6	0	2	4

Représente l'échiquier suivant :



Nous avons fait ce choix car pour que les dames ne puissent pas se menacer il ne peut pas y avoir deux dames sur la même colonne. Les solutions de ce problème comportent donc forcément une et une seule dame par colonne ce qui nous permet de représenter chaque colonne par la position de la dame qu'elle contient. De la même façon il ne doit pas y avoir deux dames sur la même ligne. Avec cette modélisation, si nous initialisons correctement l'échiquier (c'est-à-dire en évitant d'attribuer la même valeur à deux colonnes) et si nous définissons correctement le voisinage (voir chapitre dédié) nous pouvons également éviter tous les positionnements qui comportent deux dames sur la même ligne.

Tout ceci nous permet donc de réduire le nombre de solutions à explorer. Cette modélisation nous permet de passer d'un ensemble de solutions de taille n^n à un ensemble de taille n! . En effet nous avons n possibilités pour remplir la première case, n-1 pour la deuxième, etc. jusqu'à la dernière case où il ne reste plus qu'une seule possibilité. Par exemple pour un échiquier de taille = 8 nous passons de 8^8 = 16 777 216 solutions possibles à 8! = 40 320 solutions possibles.

Initialisation de l'échiquier

Nous initialisons l'échiquier en attribuant à chaque case du tableau une position aléatoire en faisant attention de ne pas remplir deux fois une case du tableau avec la même valeur. De cette façon la solution initiale ne comporte jamais deux dames sur la même ligne (ni sur la même colonne puisque notre modélisation ne le permet pas).

Définition du voisinage

Nous avons choisi de définir le voisinage de la manière suivante : les solutions voisines d'un positionnement sont celles obtenues après avoir inversé deux colonnes. De cette manière nous ne créons jamais de solutions contenant deux dames sur la même ligne.

Calcul de la fitness

Etant donné que nous évitons tout conflit horizontale et verticale, pour calculer la fitness il ne nous reste plus qu'à vérifier les diagonales. Pour cela nous regardons pour chaque couple de colonne si l'écart (en valeur absolu) de la position de leurs dames est égal à la distance entre ces deux colonnes. Si c'est le cas alors il y a un conflit et nous incrémentons la fitness.

Particularité de ce problème

La particularité de ce problème par rapport à ceux vu en cours est qu'il y a une solution optimale et que nous connaissons sa valeur. Dans chacun des algorithmes nous avons donc pu ajouter une condition d'arrêt supplémentaire quand la fitness de la solution courante est égale à 0.

Modélisation de l'algorithme génétique

Reproduction

Afin de sélectionner les solutions qui doivent se croiser nous avons décidé d'utiliser la méthode de la roulette. Nous mettons toutes les solutions dans une liste puis nous trions la liste par fitness, de la plus basse à la plus haute. Ensuite nous tirons les éléments qui vont se reproduire en donnant plus de chance à ceux qui se trouvent au début de la liste. De cette manière nous donnons plus de chance aux bonnes solutions mais sans empêcher totalement les solutions moins bonnes de se reproduire ce qui nuirait à la diversité.

Croisement

Pour croiser les solutions nous récupérons les solutions récupérées à l'étape de la sélection puis nous tirons un premier nombre aléatoire qui représente le nombre de valeurs (cases du tableau) qui vont être récupérées chez le premier parent. Ensuite nous tirons un deuxième nombre aléatoire qui représente l'indice de la première valeur prise chez le premier parent (les autres étant récupérées chez le deuxième parent). Par exemple, supposons que nous tirons respectivement 3 et 2, si on prend ces deux parents :

3	5	7	1	6	0	2	4

Et

4	7	2	6	5	1	0	3

Les cases allant de 2 à 3+2 (=5) exclus vont être pris chez le premier parent (cases bleues) le reste étant récupérer chez le deuxième parent (cases rouges). Cependant à cause de notre modélisation il faut faire attention de ne pas prendre deux fois la même valeur. Nous commençons donc par remplir les valeurs récupérées chez le premier parent. Ensuite nous plaçons un curseur au début du tableau puis nous prenons les valeurs manquantes une à une chez le deuxième parent si elles ne sont pas déjà utilisées, dans le cas contraire nous déplaçons le curseur sur la case suivante puis nous recommençons jusqu'à avoir rempli toutes les cases de gauche. Nous faisons le même chose pour les cases de droite en partant de la fin. Les cases réellement récupérées chez le deuxième parent sont donc les suivantes :

4	7	2	6	5	1	0	3

La solution résultante de ce croisement va donc être :

4	2	7	1	6	5	0	3

Cette méthode déplace des colonnes du deuxième parent ce qui peut créer des conflits mais c'est la meilleure solution que nous ayons pu trouver pour croiser deux solutions sans créer de conflits horizontaux.

Mutation

Cette étape est la plus simple une mutation consiste à effectuer une action aléatoire (inversion de deux colonnes choisit aléatoirement) sur une solution. Les mutations se produisent selon une probabilité que nous passons en argument à notre algorithme.

Ajustement des paramètres

Recuit simulé

Pour ajuster les paramètres du recuit simulé nous avons de fixer la taille de la grille à 50 pour que le temps d'exécution de l'algorithme ne soit pas trop long. Ensuite nous avons fait quelques tests de façon à pouvoir fixer de façon empirique une première fois les paramètres. Nous avons fixé nombre d'itération à 5000, la température initiale à 10000, la température final (condition d'arrêt, qui correspond à n2 dans le cours) à 0.2 et mu à 0.7 (mu correspond à la valeur avec laquelle nous multiplions la température à chaque itération). Ensuite nous avons fait varié chacun de ces paramètres un à un pour optimiser ces valeurs.

Pour commencer nous avons fait varié mu parmi les valeurs suivantes : 0.2, 0.5, 0.7, 0.8 et 0.9. Pour chacune de ces valeurs nous avons fait 20 tests. Voici les résultats obtenus :

Mu	% de réussite	Temps moyen
0,2	20%	39 ms
0,5	50%	28 ms
0,7	50%	32 ms
0,8	95%	30 ms
0,9	90%	1 s, 44 ms

Nous pouvons voir que globalement le temps moyen ne varie pas trop, sauf pour mu=0,9 puisqu'avec une valeur si élevé la température vas diminuer trop doucement et l'algorithme vas donc mettre plus de temps à s'exécuter. Les meilleurs résultats sont obtenus avec mu = 0 ,8.

Ensuite nous avons fait varier la valeur du nombre d'itérations parmi les valeurs suivantes : 1000, 5000, 10000 ,20000, 50000. Pour chacune de ces valeurs nous avons fait 20 tests. Voici les résultats obtenus :

nbIterations	% de réussite	Temps moyen
1000	5%	17 ms
5000	60%	31 ms
10000	85%	27 ms
20000	95%	58 ms
50000	100%	3 s, 5 ms
100000	100%	7 s, 42 ms

Ici, comme nous pouvions nous y attendre, les meilleurs résultats ont été obtenus avec les valeurs les plus haute. Cependant le meilleur ration temps moyen / % de réussite a été obtenu avec 10000.

Ensuite nous avons fait varier la température initiale parmi les valeurs suivantes : 100, 500, 5000, 1000, 10000, 50000. Pour chacune de ces valeurs nous avons fait 20 tests. Voici les résultats obtenus :

tempInit	% de réussite	Temps moyen
100	75%	31 ms
500	45%	47 ms
1000	50%	38 ms
5000	65%	33 ms
10000	95%	22 ms
50000	45%	45 ms

Nous pouvons remarquer que le temps moyen d'exécution est toujours du même ordre de grandeur, par contre les meilleurs résultats ont été obtenus avec une température initiale égale à 10000. Cela peut s'expliquer car avec une température trop basse il est possible de tomber dans un minimum local sans pouvoir en sortir, à l'inverse une température initiale trop élevé peu empêcher l'algorithme de se poser vers un minimum.

Ensuite nous avons fait varier la température finale parmi les valeurs suivantes : 0.01, 0.05, 0.1, 0.2, 0.5. Pour chacune de ces valeurs nous avons fait 20 tests. Voici les résultats obtenus :

tempFinale	% de réussite	Temps moyen
0,01	100%	55 ms
0,05	100%	48 ms
0,08	100%	48 ms
0,1	100%	46 ms
0,2	35%	36 ms
0,5	0%	23 ms

Nous pouvons voir qu'entre 0.01 et 0.1 les résultats sont toujours bon (100% de réussite). Avec des valeurs supérieurs le pourcentage de réussite s'effondre. Parmi les valeurs comprises entre 0.01 et 0.1, globalement le temp moyen d'exécution reste de même ordre de grandeur. Sur ce paramètre nous avions donc fait une erreur il faut prendre une valeur <= 0.1. Etant donné que plus cette valeur est basse, plus l'algorithme vas faire d'itérations, il faut éviter de prendre des valeurs trop basses. De plus 0.1 semble être la borne supérieure, nous pensons donc que la valeur optimale doit être autour de 0.08.

Pour finir, il reste une faille dans notre raisonnement, il est évident que le nombre d'itérations optimal vas dépendre de la taille de l'échiquier, nous avons donc choisit de retester les différentes valeurs de ce nombre d'itération avec une grille de taille 100 afin de nous faire une idée plus précise. Voici les résultats obtenus :

nbIterations	% de réussite	Temps moyen
1000	0%	16 ms
5000	0%	1 s et 59 ms
10000	15%	2 s et 59 ms
20000	20%	5 s et 2 ms
50000	30%	12 s et 2 ms
50000	40%	27 s et 12 ms

Ces résultats montrent que plus le nombre d'itérations est élevé plus le pourcentage de réussite augmente (pas terrible ici, certainement car les paramètres n'ont pas encore été optimisés) et plus le temps d'exécution augmente aussi. La différence avec les résultats obtenus précédemment nous prouve que le nombre d'itérations optimal dépend de la taille de la grille, il faudra donc fixer le nombre d'itération en fonction de la taille de l'échiquier.

Tous ces résultats ont été obtenus avec l'exécution de la classe main.testRecuitParameters.

Aux vues de ces résultats, nous avons choisi de fixer les paramètres comme ceci :

- mu = 0.8
- Nombre d'itérations = $300 \times \sqrt{(4 \times n)}$
- Température initiale = 10 000
- Température finale = 0.08

La formule pour le nombre d'itérations a été choisi pour que sa valeur ne s'envole pas trop vite quand la taille de l'échiquier augmente. Nos tests nous ont montré que cette valeur est suffisante, y compris pour des tailles d'échiquier importantes. Au début nous avions opté pour une formule linéaire (nombre d'itérations = 200 * taille de l'échiquier) ce qui fonctionnait très bien mais faisait s'envoler le temps d'exécution quand la taille augmentait. Cette formule fonctionne beaucoup mieux et permet d'obtenir des solutions en temps satisfaisant.

Méthode tabou

Pour la méthode Tabou, les seuls paramètres variables sont la taille de la liste des actions taboue et le nombre d'itérations maximum. Nous avons donc effectué une série de test pour vérifier l'influence de ces paramètres en fonction de la taille de la grille.

Commençons par fixer le nombre d'itérations à 500 et faisons varier taille de la liste taboue. Voilà les résultats de nos tests :

Pour n = 20 :

Taille de la liste tabou	% de réussite	Temps moyen
1	100	2ms
5	100	1ms
10	100	1ms

Pour n = 50 :

Taille de la liste tabou	% de réussite	Temps moyen
1	100	28ms
5	100	26ms
10	100	24ms
50	100	24ms

Pour n = 100 :

Taille de la liste tabou	% de réussite	Temps moyen
1	100	20ms
5	100	45ms
10	100	48ms
50	100	41ms
100	100	28ms

Pour n = 200 :

Taille de la liste tabou	% de réussite	Temps moyen
1	100	2s 29ms
5	100	3s 40ms
10	100	2s 51ms
50	100	3s 36ms
100	100	3s 51ms

Nous pouvons remarquer que cet algorithme est efficace pour ce problème et que les temps moyens sont du même ordre à par quelque exception avec une légère tendance à décroitre quand la taille de la liste augmente.

Compte tenu de la faible influence de la taille de la liste tabou sur les résultats et que nous nous doutons que sa taille optimale dépend de la taille de l'échiquier nous avons décidé de fixer sa taille à la taille de l'échiquier / 10.

Maintenant fixons la taille de la liste taboue à 200 et faisons varier le nombre d'itération maximal. Voici les résultats obtenus.

Pour n = 20 :

Nombre d'itérations maximum	% de réussite	Temps d'exécution
10	80%	1ms
100	100%	1ms
500	100%	1ms
1000	100%	1ms

Pour n = 50 :

Nombre d'itérations maximum	% de réussite	Temps d'exécution
10	55%	5ms
100	95%	22ms
500	100%	19ms
1000	100%	17ms

Pour n = 100 :

Nombre d'itérations maximum	% de réussite	Temps d'exécution
10	0%	16ms
100	55%	18ms
500	100%	21ms
1000	100%	19ms

Pour n = 200 :

Nombre d'itérations maximum	% de réussite	Temps d'exécution
10	0%	35ms
100	0%	58ms
500	85%	2s 4ms
1000	100%	2s 44ms

Nous pouvons constater le nombre d'itérations optimale dépend, comme nous pouvions nous y attendre, de la taille de l'échiquier. Comme nous avons ajouté un cas d'arrêt quand la fitness est égale à 0. Au-delà d'une certaine valeur qui est suffisante pour trouver une solution, ce paramètre n'a plus d'influence sur les performances de l'algorithme.

Tous ces résultats ont été obtenus avec la classe main.TestTabouParameters

Aux vues de ces résultats, nous avons choisi de fixer les paramètres comme ceci :

- Taille de la liste taboue = n/10
- Nombre d'itérations = n*5

Algorithme génétique

Les paramètres de l'algorithme génétiques sont : la taille de la population, le nombre de générations, et le taux de chance de mutation.

De la même manière que pour le recuit simulé nous avons commencé par faire quelques tests pour fixer les paramètres une première fois puis nous avons fait varier chacun de ces paramètres pour essayer de les optimiser.

Tous les résultats qui vont suivre ont été obtenus avec la classe main.TestGeneticParameters. L'algorithme a été testé 20 fois avec chacun des paramètres pour faire ces statistiques. Pour tenir compte de l'influence de la taille de l'échiquier nous avons fait les tests avec deux tailles : 50 et 100.

Nous avons commencé par faire varier la taille de la population. Voici les résultats obtenus :

Pour n=50 :

Taille de la population	% de réussite	Fitness moyenne	Temps moyen
10	0%	15	19ms
50	0%	13	15ms
100	0%	13	37ms
500	0%	12	3s 37ms
1000	0%	11	9s 19ms

Pour n=100 :

Taille de la population	% de réussite	Fitness moyenne	Temps moyen
10	0%	14	12ms
50	0%	11	30ms
100	0%	33	2s 28ms
500	0%	32	10s 19ms
1000	0%	32	23s 18ms

Ces résultats montrent que la taille de la population n'influence pas beaucoup les résultats obtenus. Par contre le temps moyen d'exécution augmente proportionnellement à la taille de la population. Nous obtenons des résultats satisfaisant (en comparaisons des autres) avec une population de 50 solutions, nous pensons donc qu'il n'est pas nécessaire de générer plus de solutions.

Nous avons ensuite fait varier le nombre de générations :

Pour n = 50 :

Nombre de génération	% de réussite	Fitness moyenne	Temps moyen
100	0%	14	16ms
500	0%	13	18ms
1000	0%	13	1s 29ms
5000	0%	12	6s 17ms
10000	0%	12	14s 33ms
50000	0%	11	1m 15s 51ms

Pour n = 100 :

Nombre de génération	% de réussite	Fitness moyenne	Temps moyen
100	0%	35	49ms
500	0%	33	2s 6ms
1000	0%	24	4s 49ms
5000	0%	31	21s 1ms
10000	0%	30	43s 31ms
50000	0%	29	3m 37s 1ms

Nous pouvons remarquer que le nombre de génération n'as que très peu d'influence sue la fitness moyenne obtenue à la fin, en revanche le temps d'exécution augmente très vite quand nous augmentons le nombre de générations. Les résultats obtenus après 100 générations sont donc tout à fait convenable.

Nous avons fini par faire varier la chance de mutation :

Pour n =50 :

Chance de mutation	% de réussite	Fitness moyenne	Temps moyen
0,01	0%	13	1s 18ms
0,05	0%	13	1s 11ms
0,08	0%	13	1s 2ms
0,1	0%	12	1s 5ms
0,2	0%	13	1s 8ms
0,5	0%	13	1s 10ms

Pour n = 100 :

Chance de mutation	% de réussite	Fitness moyenne	Temps moyen
0,01	0%	31	4s 27ms
0,05	0%	32	4s 45ms
0,08	0%	33	4s 3ms
0,1	0%	33	4s 47ms
0,2	0%	32	4s 10ms
0,5	0%	33	4s 11ms

Nous pouvons constater, encore une fois, que la chance de mutation a très peu d'influence que ce soit sur les résultats obtenus comme sur le temps d'exécution.

Aux vues de ces tests, nous avons décidé de fixer les paramètres de l'algorithme génétique comme ceci :

- Taille de la population = 50
- Nombre de générations = 100
- Chance de mutation = 0.05

Résultats obtenus

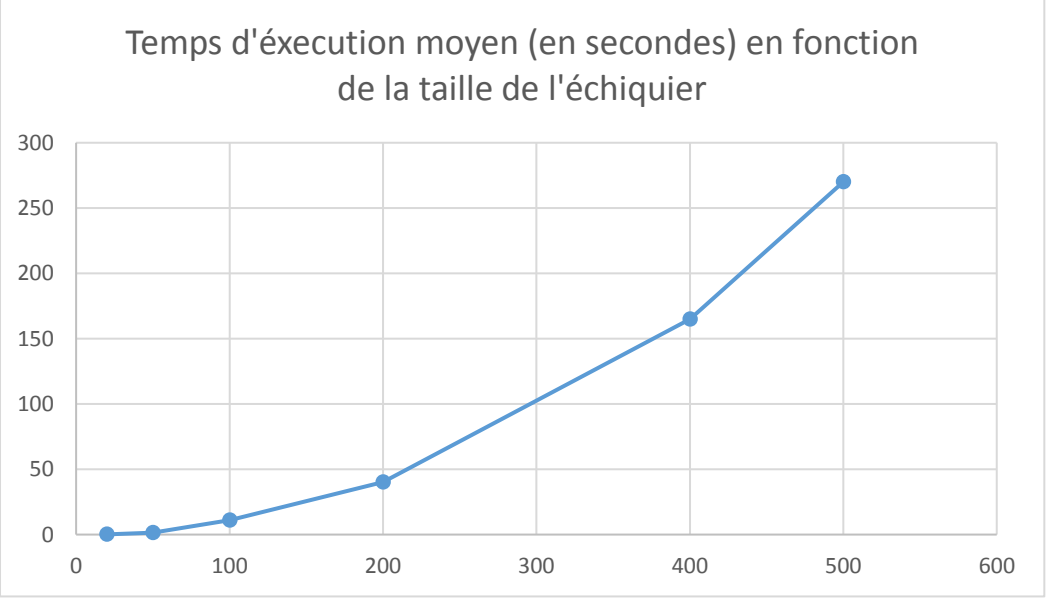
Recuit simulé

Voici les résultats obtenus avec l'algorithme du recuit simulé et les paramètres obtenus ci-dessus avec différentes tailles d'échiquier (l'algorithme est lancé 20 fois avec chaque taille pour faire des statistiques) :

Taille de l'échiquier	% de réussite	Temps moyen
20	100%	11 ms
50	100%	1 s et 52 ms
100	100%	11 s et 12 ms
200	100%	40 s et 12 ms
400	100%	1m 42s 56ms
500	100%	4m 35s 3ms

(Valeurs obtenus avec la classe main.TestRecuit)

Ces résultats nous permettent de tracer la courbe du temps d'exécution en fonction de la taille de la grille :



Nous pouvons constater que cette courbe croit de manière exponentielle. L'obtention de solution pour des tailles d'échiquier plus élevées risque donc d'être plus long.

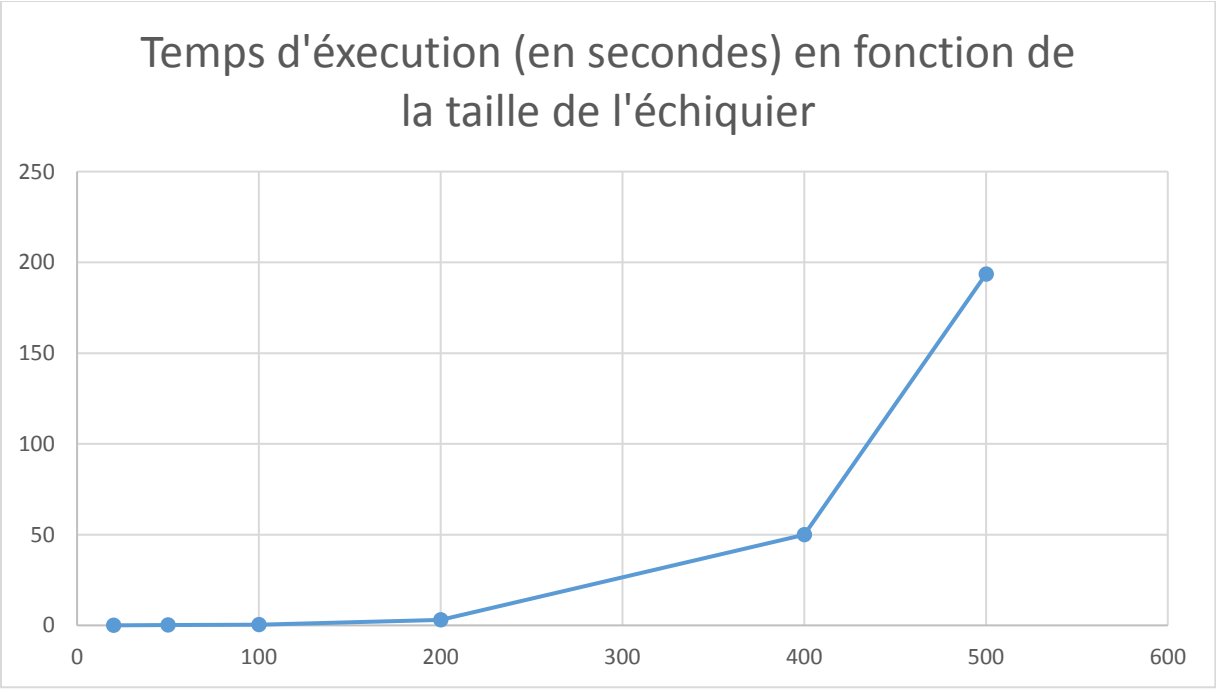
Méthode tabou

Voilà les temps moyens pour arriver à une fitness de 0 avec notre algorithme tabou avec les paramètres fixés précédemment et des séries de 20 tests pour chaque taille de grille :

Taille de la grille	% de réussite	Temps moyen
20	100%	2ms
50	100%	25ms
100	100%	40ms
200	100%	3s 13ms
400	100%	50s
500	100%	3m 13s 52 ms

(Résultats obtenus avec main.TestTabou).

Temps moyen (en ms) e fonction de la taille de la grille :



Tout comme le recuit simulé cet algorithme fonctionne très bien. Nous arrivons à obtenir des solutions pour une taille de 500 en moins de cinq minutes ce qui nous semble acceptable.

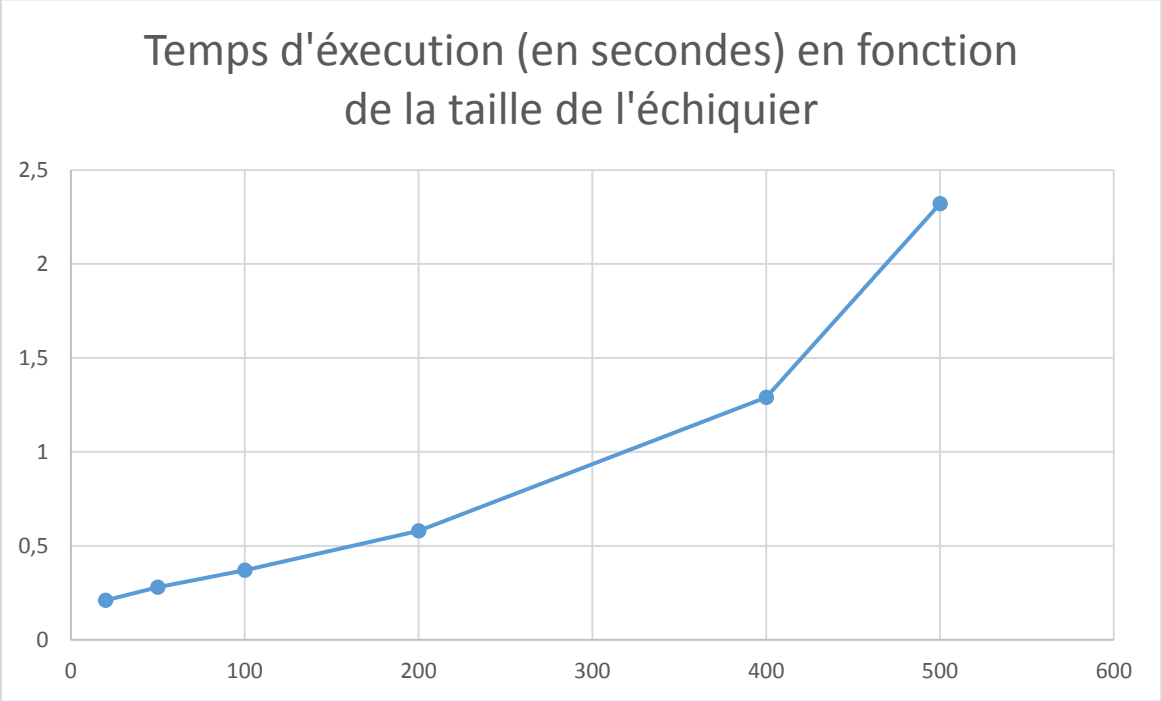
Algorithme génétique

Voici les résultats obtenus avec l'algorithme génétique et les paramètres obtenus ci-dessus avec différentes tailles d'échiquier (l'algorithme est lancé 20 fois avec chaque taille pour faire des statistiques) :

Taille de la grille	Fitness moyenne	Temps d'exécution
20	1	21 ms
50	14	28 ms
100	37	37 ms
200	88	58 ms
400	203	1s 29 ms
500	265	2s 32ms

(Valeurs obtenus avec la classe main.TestGenetic)

Ces résultats nous permettent de tracer la courbe suivante :



Nous pouvons remarquer que cet algorithme est très rapide mais vraiment pas efficace, il ne trouve presque jamais de solution optimale à ce problème.

Conclusion

Les algorithmes de recuit simulé et de la méthode tabou ce sont montrés très efficace pour résoudre le problème des n-dames. Ces algorithmes arrivent à résoudre ce problème, même pour des tailles d'échiquier relativement importants (nous avons testé des tailles allant jusqu'à 500) en temps convenable.

L'algorithme tabou se montre même un peu plus performant que le recuit simulé mais sa courbe à l'aire de croitre de manière plus exponentielle que celle du recuit simulé. Nous pouvons donc nous attendre à ce que le recuit simulé devienne plus performant sur des échiquiers de taille encore supérieur à 500.

En revanche l'algorithme génétique s'est montré très décevant sur ce problème. Il est très rapide mais n'aboutit à aucune solution. Le problème vient du fait que les générations ne s'améliorent pas. C'est d'ailleurs ce qui fait que nous n'avons pas remarqué une grande influence du nombre de générations sur les résultats, ce qui nous a fait choisir un nombre de générations faible et qui doit être la cause des bonnes performances temporelles de cet algorithme. Nous pensons que c'est l'étape du croisement qui fonctionne mal. En effet, avec cette méthode, les solutions fabriquées ne sont pas meilleures que les solutions parentes. C'est pourtant la meilleure méthode que nous avons trouvée compte tenu de notre modélisation. Nous avons bien réfléchi à trouver une autre modélisation plus adaptée à l'algorithme génétique (comme s'autoriser les conflits horizontaux) mais aucune ne s'est montrée plus performante. Nous pouvons aussi remarquer que la mutation est très peu efficace puisqu'elle n'a aucun impact sur les résultats obtenus mais une fois de plus nous n'avons pas trouvé de meilleure solution. Peut-être est-ce simplement l'algorithme génétique qui est très peu adapté à ce problème.