22 mai 2017

Léa COLLIN – Mélanie PETITCUENOT

Polytech lyon

Problème deS N-Reines

Algorithmes d’optimisation

# Table des matières

[Table des matières 1](#_Toc483223118)

[Introduction 1](#_Toc483223119)

[I. Modélisation du problème 2](#_Toc483223120)

[a. Modélisation du damier 2](#_Toc483223121)

[b. Modélisation de la fitness 2](#_Toc483223122)

[c. Modélisation des voisins 3](#_Toc483223123)

[II. Guide d’utilisation programme 5](#_Toc483223124)

[III. Algorithme Recuit Simulé 7](#_Toc483223125)

[a. Variation du nombre d’itération 7](#_Toc483223126)

[b. Variation de la température initiale et fonction de décroissance 10](#_Toc483223127)

[IV. Algorithme Tabou 13](#_Toc483223128)

[a. Variation du nombre d’itération 13](#_Toc483223129)

[b. Variation du nombre de voisins 13](#_Toc483223130)

[V. Algorithme génétique 14](#_Toc483223131)

[a. Variation du nombre d’itération 14](#_Toc483223132)

[b. Variation de la taille de la population 14](#_Toc483223133)

# *Introduction*

Le but du problème des N-reines est de placer N reines dans un échiquier de N par N cases sans que les dames ne puissent se menacer mutuellement (en respectant les règles du jeu d’échecs). Alors, deux dames ne devraient jamais être sur la même ligne, colonne ou diagonale.

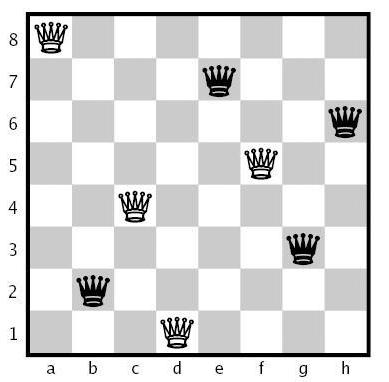


Figure 1 : Exemple de solution pour 8 reines

Il s’agit d’un problème mathématique qui peut être résolu à l’aide de programmes informatiques. C’est pourquoi nous l’avons appliqué à trois algorithmes d’optimisation discrète afin de les tester, et de comparer leurs résultats.

Dans la suite de ce rapport, nous allons vous présentez comment nous avons modélisé ce problème informatiquement, puis les différents résultats que nous avons obtenus.

# Modélisation du problème

## Modélisation du damier

La première étape dans ce projet a été de modéliser correctement le damier et la position des dames afin que les calculs soient rapides et ne requièrent pas une mémoire excessive.

En effet, notre but étant de résoudre le problème avec un N grand (l’idéale étant d’aller jusqu’à 1 000 dames), il était indispensable que la génération des tableaux et leur stockage aient une complexité mémoire minime.

Nous avons donc testé différentes solutions pour modéliser le tableau. Créer un vrai damier avec des cases et une position x et y pour chaque reine s’est vite avéré trop gourmand en mémoire. Nous avons donc opté pour un simple tableau qui listerait les positions de chaque reine. L’index du tableau représente la colonne et le chiffre contenu à cet index, la ligne.

De plus, pour optimiser le tableau dès son initialisation, les reines sont positionnées de manière à ce qu’il n’y qu’une reine sur chaque ligne et sur chaque colonne. Pour ce faire, nous plaçons les reines sur la diagonale principale puis nous mélangeons les colonnes.

public void initRows(){  
 for(int i = 0; i < size ; i++ ){  
 rows.add(i);  
 }  
 *shuffle*(rows);  
}

## Modélisation de la fitness

Par la suite, nous avons fait le choix du calcul de la fitness. La fitness étant la fonction objectif (à maximiser ou minimiser) nous avons décidé que la fitness représenterait le nombre de conflit entre les reines. Elle sera donc à minimiser.

Dans notre cas, il s’agit de deux reines (ou plus) étant sur la même diagonale.

public int fitness(){  
  
 int fitness = 0;  
  
 for(int i = 0; i < size ; i++){  
  
 int columnRef = rows.get(i);  
  
 for(int j = i + 1 ; j < size ; j++){  
 int columnTest = rows.get(j);  
  
 if(Math.*abs*(i-j) == Math.*abs*(columnRef - columnTest)){  
  
 fitness++;  
 }  
 }  
 }  
 return fitness;  
}

## Modélisation des voisins

La dernière chose à mettre en place dans notre modélisation était la génération de voisins. Nous avons décidé qu’un damier serait considéré comme voisin s’il possédait au plus deux dames positionnées différemment. Alors, pour générer l’ensemble des voisins, nous devions échanger chacune des colonnes deux par deux.

public ArrayList<Board> selectNeighbours(){  
  
 ArrayList<Board> neighbours = new ArrayList<Board>();  
  
 ArrayList<Integer> rowsNeighbour;  
  
 for(int i = 0 ; i < size ; i++ ){  
  
 for (int j = i + 1; j < size ; j++){  
  
 rowsNeighbour = new ArrayList<>(rows);  
 rowsNeighbour.set(i, rows.get(j));  
 rowsNeighbour.set(j, rows.get(i));  
 neighbours.add(new Board(rowsNeighbour));  
  
 }  
  
 }  
  
 return neighbours;  
}

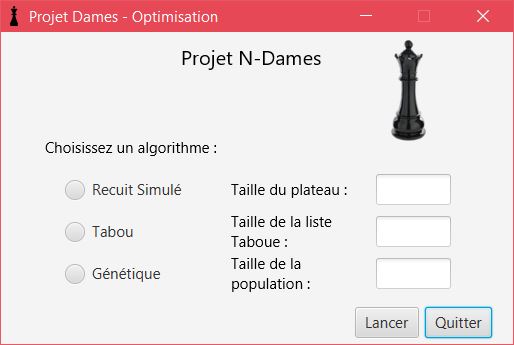
Nous avons également créé une méthode permettant de générer un voisin aléatoire. Pour cela, deux colonnes sont échangés de manière aléatoire.

public Board neighbourRandom(){  
  
 ArrayList<Integer> rowsNeighbour = new ArrayList<>(rows);  
  
 Random r = new Random();  
  
 int row1 = r.nextInt(size);  
  
 int row2 = r.nextInt(size);  
  
 while (row2 == row1){  
  
 row2 = r.nextInt(size);  
  
 }  
  
 int temp = rows.get(row1); //number of the column  
 rowsNeighbour.set(row1, rows.get(row2));  
 rowsNeighbour.set(row2, temp);  
  
 return new Board(rowsNeighbour);  
}

Maintenant, nous allons vous presenter les différents tests effectués sur les différents algorithmes. Nous avons effectué plusieurs tests pour chaque possibilité, ainsi les données présentées sont des moyennes pour être au plus près de la vérité.

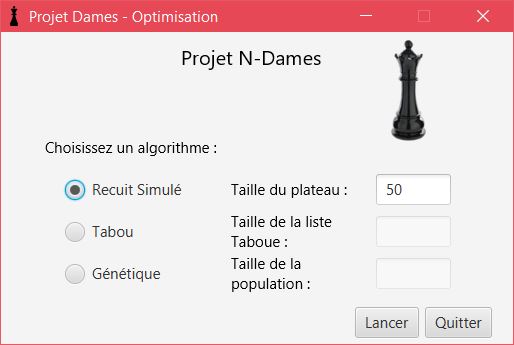
# Guide d’utilisation programme

Bienvenue à tous sur notre programme d’optimisation qui a pour but de placer N reines sur une chess-game de taille N \* N en minimisant les conflits. Lorsque vous lancerez l’application, la fenêtre suivante s’ouvrira :

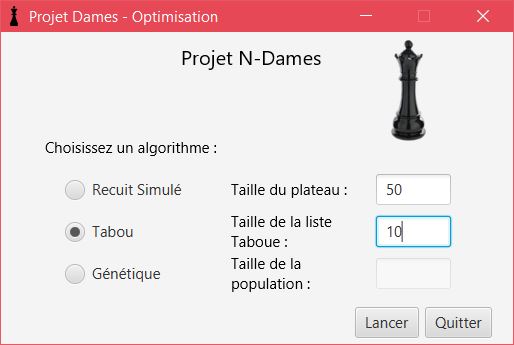


A ce stade, il vous reste à choisir avec quel algorithme vous souhaitez placer vos reines. Les 3 algorithmes que nous avons développés sont les suivants : Recuit Simulé, Tabou et Génétique.

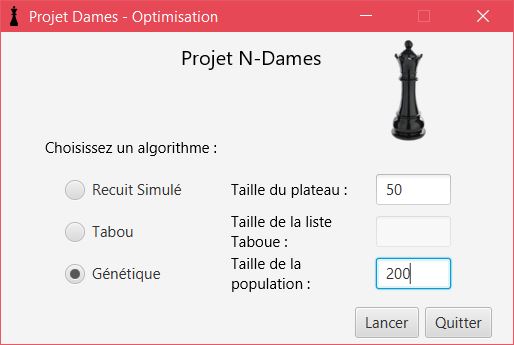
Lorsque vous choisissez « Recuit Simulé », il vous faudra remplir la taille du plateau de cette façon :



Pour l’algorithme « Tabou » il faudra également saisir la taille de la liste Tabou de cette manière :



Enfin, pour l’algorithme génétique, il faudra saisir la taille de la population à la place de la taille de la liste Tabou :



Cliquez ensuite sur « Lancer » et l’algorithme choisit sera mis en route.

A la fin de l’exécution, les résultats sont présentés sous cette forme :

La **fitness initiale** au début de l’algorithme est de 64. Il y a 64 conflits entre les reines.

La **fitness finale** est optimale, elle vaut 0, les reines n’ont aucun conflit entre elles.

L’algorithme a réalisé 12851 **itérations** du Recuit Simulé (sur 100 000 autorisées).

L’algorithme a mis 0.44 **secondes** pour trouver une solution.



Vous pouvez ensuite relancer une exécution en revenant au Menu ou Quitter l’application.

Bon courage !

# Algorithme Recuit Simulé

## Variation du nombre d’itération

Dans l’algorithme du recuit simulé, le nombre d’itérations indique le nombre de fois que la recherche de voisins va être effectué. Cela permet d’éviter que l’algorithme ne s’arrête jamais, dans le cas où il ne serait pas en mesure de trouver une fitness nul.

Nous avons choisi d’étudier le comportement de notre algorithme pour 1000, 10 000, 100 000 et 1 000 000 d’itérations. Pour cela nous n’avons pas touché les autres paramètres afin de conserver l’exclusivité du nombre d’itérations.

Voici donc, ci-dessous, l’évolution de la fitness initiale, de la fitness finale, du nombre d’itérations utilisées et du temps obtenus par notre algorithme pour ces 4 nombres d’itérations, et pour N allant jusqu’à 1000.

**La fitness initiale**

Les fitness initiales en fonction de la taille de N et de nos nombres d’itérations sont les suivantes, en moyenne.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N \ NB ITE | 1000 | 10000 | 100000 | 1000000 |
| 10 | 8 | 5 | 6 | 7 |
| 50 | 31 | 43 | 44 | 31 |
| 100 | 58 | 73 | 69 | 75 |
| 500 | 280 | 337 | 309 | 313 |
| 1000 | 659 | 649 | 635 | 677 |

Comme on pourrait le déduire logiquement, les fitness initiales augmentent proportionnellement à la taille du tableau (N). C’est-à-dire, pour un tableau grand et donc un nombre de reines grand, la fitness initiale est plus élevée que pour un petit tableau. On remarque également que les fitness initiales ne dépendent pas du nombre d’itérations (notre paramètre), ce qui est également logique puisque lorsque la fitness initiale est calculée, l’algorithme est à l’itération 0.

**La fitness finale**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N \ NB ITE | 1000 | 10000 | 100000 | 1000000 |
| 10 | 0 | 0 | **0** | 0 |
| 50 | 1,43 | 0 | **0** | 0 |
| 100 | 8,5 | 0 | **0** | 0 |
| 500 | 102,25 | 18,5 | **0** | 0 |
| 1000 | 306,5 | 74 | **1,5** | 0 |

Avec 1000 itérations, on remarque que lorsque la taille du tableau augmente, la fitness finale augmente également. En effet, cela signifie que 1000 itérations ne sont pas suffisantes pour atteindre l’objectif (fitness = 0). Avec 10 000 itérations, la fitness finale n’est toujours pas bonne pour des grands tableaux, mais elle reste meilleure que pour 1000 itérations. Enfin, pour 100 000 et 1 000 000 d’itérations, l’objectif est toujours atteint (ou presque). On peut donc considérer utiliser au moins 100 000 itérations pour faire fonctionner notre algorithme.

**Le temps**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N \ NB ITE | 1000 | 10000 | 100000 | 1000000 |
| 10 | 0,0005 | 0,00189 | **0,000903** | 0,0003 |
| 50 | 0,007428571429 | 0,016 | **0,0187** | 0,023 |
| 100 | 0,033 | 0,1689 | **0,1715** | 0,27 |
| 500 | 1,095 | 10,5 | **46,084** | 30,958 |
| 1000 | 4,9775 | 40,569 | **450,398** | 667,99 |

On remarque que les courbes de ce graphique sont inversées par rapport au graphique précédemment. Cela parait logique, puisqu’en effet, plus il y a d’itérations, plus l’algorithme met de temps (en moyenne). Suite aux conclusions du premier graphique, on peut choisir 100 000 itérations car l’algorithme est plus rapide que pour 1 000 000 d’itérations.

**Le nombre d’itérations utilisées**

Le tableau ci-dessous présente le nombre d’itérations utilisées par l’algorithme, en pourcentage du nombre d’itérations global (pour 1, l’algorithme utilise la totalité des itérations), et cela pour des tailles de tableaux (N) 10, 50, 100, 500, 1000.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N \ nb ite | 1000 | 10000 | 100000 | 1000000 |
| 10 | 0,424 | 0,0034 | 0,00094 | 0,000178 |
| 50 | 1 | 0,1697 | 0,03972 | 0,000742 |
| 100 | 1 | 0,6147 | 0,04303 | 0,004925 |
| 500 | 1 | 1 | 0,24911 | 0,032396 |
| 1000 | 1 | 1 | 0,988 | 0,090536 |

Ce que l’on peut conclure de ce graphique est qu’il est inutile d’utiliser 1 000 000 d’itérations puisque seules les 10 % premières (100 000 itérations) sont parcourues avant de trouver une solution. A l’inverse, 1000 itérations sont atteintes rapidement par les grands tableaux ce qui montre que ce n’est pas assez. C’est la même conclusion que pour le graphique sur la fitness finale.

On peut donc conclure de manière générale qu’un nombre de 100 000 itérations est bon pour le fonctionnement de notre algorithme. Il permet d’atteindre une bonne solution avec plus de 90% de chance, quel que soit la taille du tableau et tout en conservant un timer « raisonnable".

## Variation de la température initiale et fonction de décroissance

Pour un algorithme du recuit performant, il est conseillé d’utiliser une température initiale forte (permettant d’augmenter la probabilité d’accepter les mauvaises solutions), et une fonction de décroissance de la forme *temp = nu \* temp* (avec nu proche de 1).

En pratique, au vu de la façon dont est implémenté notre algorithme de recuit, et au fur et à mesure de nos différents tests, nous nous sommes rendu compte que le changement de température initiale ne modifiait que très peu nos données finales (que ce soit la fitness obtenue, le temps d’exécution ou le nombre d’itérations utilisées). A quoi cela est-il dû ?

Pour nos tests, nous avons utilisés deux valeurs de nu principalement : un nu = 0.99 pour tester le fonctionnement du recuit avec un nu proche de 1 et un nu = 0.4. Pour 0.4, cela signifie que la température décroit très rapidement, c’est la raison pour laquelle notre température initiale n’impacte pas les résultats. Même si l’on choisit une température élevée, elle va décroitre si rapidement que l’on ne verra pas la différence d’une température initiale très faible.

Ensuite, pour 0.99, nous avons relevé une faible augmentation du temps d’exécution lorsque la température initiale augmente. Cependant, celle-ci étant minime, elle ne nécessite pas de choisir une valeur de préférence pour la température initiale.

Voici les temps obtenus pour des tableaux de taille 10, 500 et 1000.

|  |  |  |  |
| --- | --- | --- | --- |
| temp\taille | taille 10 | taille 100 | taille 1000 |
| 0,1 | 0,0057 | 0,147 | 422,5 |
| 10 | 0,00578 | 0,15 | 420 |
| 50 | 0,00589 | 0,151 | 420,5 |
| 200 | 0,00589 | 0,184 | 420,6 |
| 500 | 0,00596 | 0,2005 | 422,43 |
| 600 | 0,0061 | 0,21 | 422,3 |
| 750 | 0,0072 | 0,2 | 430,9 |
| 1000 | 0,0072 | 0,201 | 480 |

D’après ces 3 graphiques, on observe bien que le temps d’exécution augmente de quelques centaines de secondes pour un petit tableau à une quarantaine de secondes pour un tableau grand. Il est cependant préférable de conserver une température élevée au début, même si l’on n’en voit pas l’utilité à l’œil nu. Dans une autre configuration, une température faible pourrait bloquer l’algorithme trop rapidement en l’empêchant de parcourir à sa guise les solutions potentielles.

,

# Algorithme Tabou

## Variation du nombre d’itération

## Variation du nombre de voisins

# Algorithme génétique

## Variation du nombre d’itération

## Variation de la taille de la population