Université Abdelmalek Essaidi

Ecole National des Science Appliquées

De Al Hoceima

Filière : Génie informatique

Résolution du problème des N dames avec trois algorithmes

Sous python

Réalisé par :

Mohamed MIRI

Année Universitaire : 2020/2021

Bouddha-ctrl/NQueen_5_algorithms (github.com)
LinkedIn : https://www.linkedin.com/in/mohamed-miri-37299a163/
Email : miri.mohamed.1998@hotmail.com

Sommaire

Introduction

Les **mathématiques** sont un ensemble de connaissances abstraites résultant de raisonnements logiques appliqués à des objets divers, et elles possèdent plusieurs branches telles que : l'arithmétique, l'algèbre, l'analyse …

Plusieurs d'entre nous trouve les maths difficiles contrairement aux autres qui les trouve amusantes à travers les **jeux mathématiques.**

Ce qui distingue un jeu mathématique d'un autre jeu ordinaire, c'est l'accent mis sur l'analyse mathématique du jeu, la logique nécessaire à son accomplissement, plus que sur la façon de jouer.

Cette complexité de jeux nous laisse confus de choisit le bon algorithme pour le résoudre la problématique du jeu.

Alors on va voir résoudre le problème de N dames avec l'algorithme Hill Climbing, recherche en faisceau et l'algorithme génétique et les comparer à la fin.

Chapitre 1 : Problématique

Dans ce chapitre nous allons donner un aperçu historique du problème du sac à dos, ensuite, nous allons définir le problème et sa formulation mathématique, puis quelque algorithme utiliser dans la résolution de ce problème, enfin nous allons citer quelques domaines d'application dans la vie réelle.

I. Définition :

Le problème du N-Queen est t'introduit aux publique en 1848 par Max Bezzel, son but est de placer N dames d'un jeu d'echecs sur un échiquier de NxN cases, sans que les dames ne puissent se menacer mutuellement, conféremement aux règle d'echecs. Par conséquence deux dames ne devraient jamais partager la meme rangée, colonne ou diagonale.

II. Résolution :

Il existe quatre types de programmation qui aide à la résolution de ce problème :

1. Programmation Récursive : (Exemple : backtracking)

   En exprimant qu'une solution du problème des n-dames peut être obtenue, par récurrence, à partir d'une solution quelconque du problème des (n-1)-dames par l'adjonction d'une dame. La récurrence commence avec la solution du problème de 0-dame qui repose sur un échiquier vide.

2. Programmation par contrainte : (Exemple SAT resolve)

   On cherche des états ou des objets satisfaisant un certain nombre de *contraintes* ou de critères, Càd représenter le problème sous forme d'une formule de logique propositionnelle, détermine s'il existe une assignation des variables propositionnelles qui rend la formule vraie.

3. Programmation logique : (Exemple : Hill Climbing, Beam Search)

   Une forme de programmation qui définit les applications à l'aide d'une base des faits ; ensemble de faits élémentaires concernant le domaine visé par l'application.
   Une base de règle ; règles de logique associant des conséquences plus ou moins directes à ces faits.

Une base d'inférence ; exploite ces faits et ces règles en réaction à une question ou requête.

4. Algorithme génétique.

III. Analyse de la méthode naïve (brute force) :

La méthode naïve trouve toutes les solutions possibles, les tester puis trouve la bonne, comme on sait les solutions d'un problème sont des vecteurs de longueurs n, où n est la taille du problème, et les composantes des vecteurs valant de 1 à n.
On peut conclure que le nombre de solution (Espace d'état) est : n x puissance(n)
Et si on suppose que C est la complexité du calcule de la valeur d'une seule possibilité, la complexité de cette méthode sera : $C * n^n = \mathrm{O}(n^n)$

Pour N= 8, Le nombre de combinaisons est : $8^8 = 16\ 777\ 216$, pour 92 solutions.
Si on suppose que la complexité C s'exécute dans 10^{-6} seconde, le temps d'exécution du code complet pour N = 8 objets sera : (le pire cas)

$$(16\ 777\ 216 - 91) * 10^{-6} sec = 16.7\ sec$$

Pour N=12 :

$$8.9161004 * 10^{12} * 10^{-6}\ sec = 3.4\ mois$$

Chapitre 2 : Résolution

I. L'environnement de travail :

Python :

Python est un langage de programmation interprété, multi paradigme et multiplateformes. Il favorise la programmation impérative structurée, fonctionnelle et orientée objet.

Il est doté d'un typage dynamique fort, d'une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exceptions ; il est ainsi similaire à Perl, Ruby, Scheme, Smalltalk …

Bibliothèque :

On va utiliser les bibliothèques suivantes : random, typing, time, functools

```
import random, time
from typing import List , Tuple
from functools import partial
```

Pour la bibliothèque random, on va utiliser les fonctions :

- Sample : choisit K éléments unique d'une liste.
- Choices : choisit K éléments selon leurs poids.
- Seed : initialise le générateur des nombres aléatoire, génère les mêmes nombres aléatoires pour le même entier passer au paramètre.
- Randint : génère un entier entre A et B du paramètre inclusive.
- Random : génère un nombre flottant ente 0 et 1 exclusive.

II. La Modélisation du problème :

Pour représenter un état du problème, on va utiliser un codage, une liste de nombre :

$$State = [x_1, x_2, x_3, \dots, x_i, \dots, x_n] \; avec \; i, x \in [1, n]$$

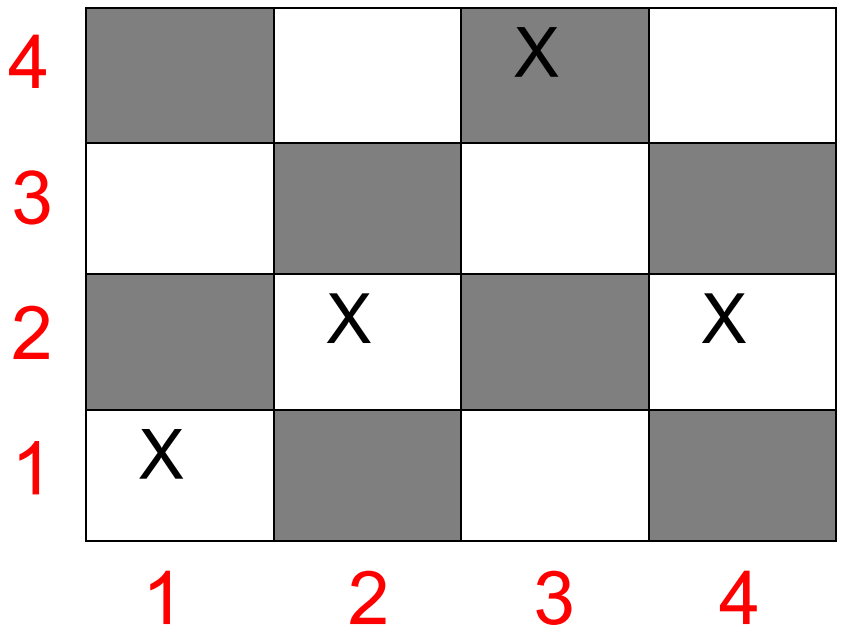
Où l'indice i de la liste représente la colonne et la valeur $x_i$de l'indice i représente la ligne.

Python :

```
State = List[int]
```

Exemple :

$$State = [1,2,4,2]$$



III. Les fonctions en commun :

1. La création aléatoire d'un état :

Comme on a dit au début, notre but est de placer une dame par ligne, colonne et diagonale.

Le codage ci-dessus assure une dame par colonne, pour avoir une dame par ligne, on doit s'assurer qu'on a une occurrence de chaque nombre dans la liste.

Alors à la place de :

$$State = [1,2,4,2]$$

On replace le nombre 2 qu'il a deux occurrences, avec le nombre restant, 3

$$State = [1,2,4,3]$$

Python :

```
def generate_State(lenght: int) -> State  :
    return random.sample(list(range(1,lenght+1)),k=lenght)
```

La fonction prend la longueur de la liste (la taille du problème) en paramètre, et retourne une liste de nombre compris entre 1 et N mélangé.

list(range(1,lenght+1)) : est une liste de nombre de 1 à N (1,2,3,4…,N)

La méthode random.sample : choisit K élément aléatoirement de la liste passer en paramètre sans répétition.

2. La création d'un ensemble d'état :

Pour l'algorithme génétique (population) et Beam Search, on va avoir besoin d'une liste des listes pour stocker les états actuels. Qu'on va l'appeler State_Space même s'il représente qu'une partie de l'espace d'état.

Python :

```
State_Space = List[State]
def generate_State_Space(size: int):
    return [generate_State(size) for _ in range(size)]
```

La fonction prend la taille de la population en paramètre, elle exécute la méthode generate_State() taille fois, et retourne une Liste des états générer aléatoirement.

3. La fonction d'évaluation :

La fonction d'évaluation (fitness) calcule le nombre d'attaque entre les dames, directement et indirectement.
Puisqu'on est sure qu'il n'a pas d'attaque sur les lignes et les colonnes, il nous reste de vérifie les diagonales et les antidiagonales.

Partie 1 :

Deux dames se situe sur la même diagonale si et seulement si :

$$\forall i, j \in [1, N], \qquad (x_i - i) == (x_j - j)$$

Deux dames se situe sur la même antidiagonale si et seulement si :

$$\forall i, j \in [1, N], \qquad (x_i + i) == (x_j + j)$$

Partie 2 :

Pour calculer le nombre d'attaque, il suffit de calculer :

$\sum \frac{occ(i)*(occ(i)-1)}{2}$, occ : Nombre d'occurrence.

Exemple :

Pour $State = [1,2,4,3]$:

$$Diag(state) = [0,0,1,-1]$$

$$AntiDiag(state) = [2,3,6,7]$$

Nombre d'attaque f1 =1

Nombre d'attaque f2 = 0

Nombre d'attaque f1+f2 = 1

Python :

```python
def fitness(state : State ):

    lenght = len(state)

    #partie 1
    diagonal1 = [state[i] - (i+1) for i in range(lenght)]
    diagonal2 = [state[i] + (i+1) for i in range(lenght)]

    #partie 2
    f1 =sum( [diagonal1.count(i)*(diagonal1.count(i)-1)//2 for i in set(diagonal1)] )
    f2 =sum( [diagonal2.count(i)*(diagonal2.count(i)-1)//2 for i in set(diagonal2)] )

    return f1 + f2
```

On passe l'état en paramètre, la première partie consiste a appliqué la fonction qui trouve les positionnements.

La deuxième partie calcule le nombre d'attaque, on utilise le set pour ne pas calculer l'occurrence d'un nombre (la fonction qui calcule le nombre d'attaque) plus qu'une fois (s'il a plusieurs occurrences).

Ps :

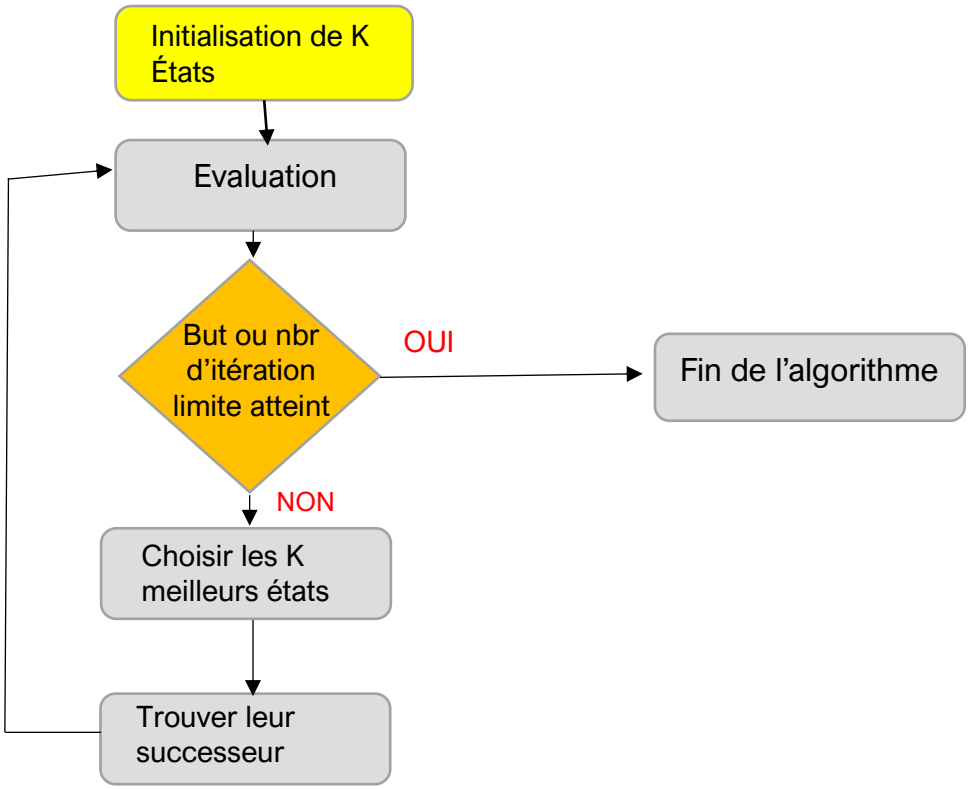
- J'ai utilisé (i+1) dans partie 1 car i commence de 0, contrairement aux valeurs de state qui commence de 1.
- Set est un type de collection qui ne tolère pas les répétitions, quand on transforme une liste à un set, les répétitions serons supprimés.
- Le double slache retourne que la partie entière de la division.

Pour le reste, chaque algorithme à ses propres fonctions.

IV. L'algorithme Beam Search :

Cet algorithme est une amélioration de l'algorithme parcoure en largeur.

A chaque niveau, elle génère tous les successeurs du nœud courant, en les classant selon leur coût heuristique. Cependant, elle ne mémorise qu'un nombre prédéterminé de ces états à chaque niveau.



1. Les successeurs :

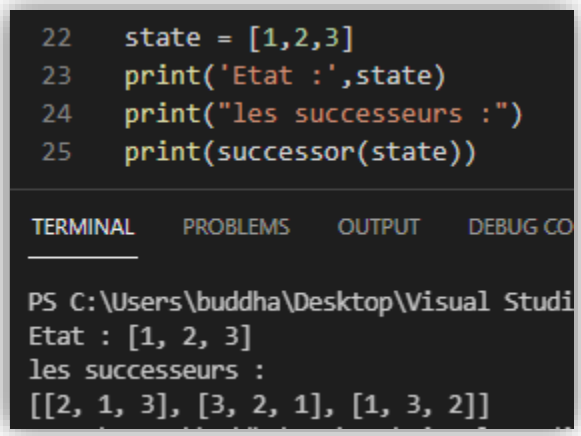
Pour trouver les successeurs d'un état, on va permuter chaque deux colonnes.
De cette façon on va tenir les contraintes d'une dame par colonne et par ligne valable.
De cette manier, chaque état aura $\frac{N*(N-1)}{2}$ successeurs, avec N la taille du problème.

Python :

```python
def successor(state: State) -> State_Space:
    L = []
    lenght = len(state)
    for i in range(lenght-1):
        for j in range(i+1,lenght):
            s=state.copy()
            s[i],s[j]=s[j],s[i]
            L+=[s]
    return L
```

La fonction prend un état en paramètre et retourne une liste des listes (les successeurs).

Exemple :



2. Développement :

Dans cette méthode on va utiliser trois méthodes :

- Generate_state_space
- Fitness
- successor

Dans cette méthode on va implémentés le concept de l'algorithme :

Les entrées : les trois fonctions, la taille du faisceau et le nombre d'itération maximal.

Les sorties : un état, le nombre d'itération avant le programme s'arrête et la valeur de l'évaluation de l'état retourné.

Python :

```python
def run(fitness_funct, successor_funct, generate_State_Space_funct, SizeSpace, MaxIteration=200):

    Space = generate_State_Space_funct(SizeSpace)   #generation d'un population

    for i  in range(MaxIteration):                  #vérifier le nombre de l'itération
        Space = sorted(Space,key=lambda state: fitness_funct(state)) #trier les etats on se basant
                                                                     #sur leurs evaluation

        if fitness_funct(Space[0])==0: break        #Vérifier l'etat but

        Space = Space[:SizeSpace]                   #choisir les meilleur K états

        successor = []
        for state in Space:
            successor+=successor_funct(state)       #les successeurs des K états
        Space = successor

    Space = sorted(Space,key=lambda state: fitness_funct(state)) #si on n'a pas trouvé une solution

    return Space[0],i,fitness_funct(Space[0])
```

On va suivre les mêmes étapes de l'organigramme :

- Générer une population aléatoirement
- Vérifier le nombre d'itération
- Trier les états croissement en se basant sur leurs évaluation (Fitness)
- Puisque le trie est croissant, le meilleur état est le premier, on vérifie s'il est état but
- Sinon, on choisit K meilleur état de l'espace, de 0 inclusif à SizeSpace exclusif puisque les états son déjà trier.
- On revient à l'étape 2

On refait un triage après la boucle, pour nous aider à voir le meilleur état atteint si on n'a pas eu une solution.

3. Test :

On teste l'algorithme avec le code suivant : (random.seed(100))

```
SizePuzzle = 8

start = time.time()
state,iteration,value = run(fitness_funct=fitness,
                            successor_funct=successor,
                            generate_State_Space_funct=generate_State_Space,
                            SizeSpace=SizePuzzle,
                            MaxIteration=200)
end = time.time()

print('Etat :',state)
print('Nb d\'itération :',iteration)
print('Evaluation (Nb attaque):',value)
print('Temps :',end - start)
```

Résultat, Pour N = 8 :

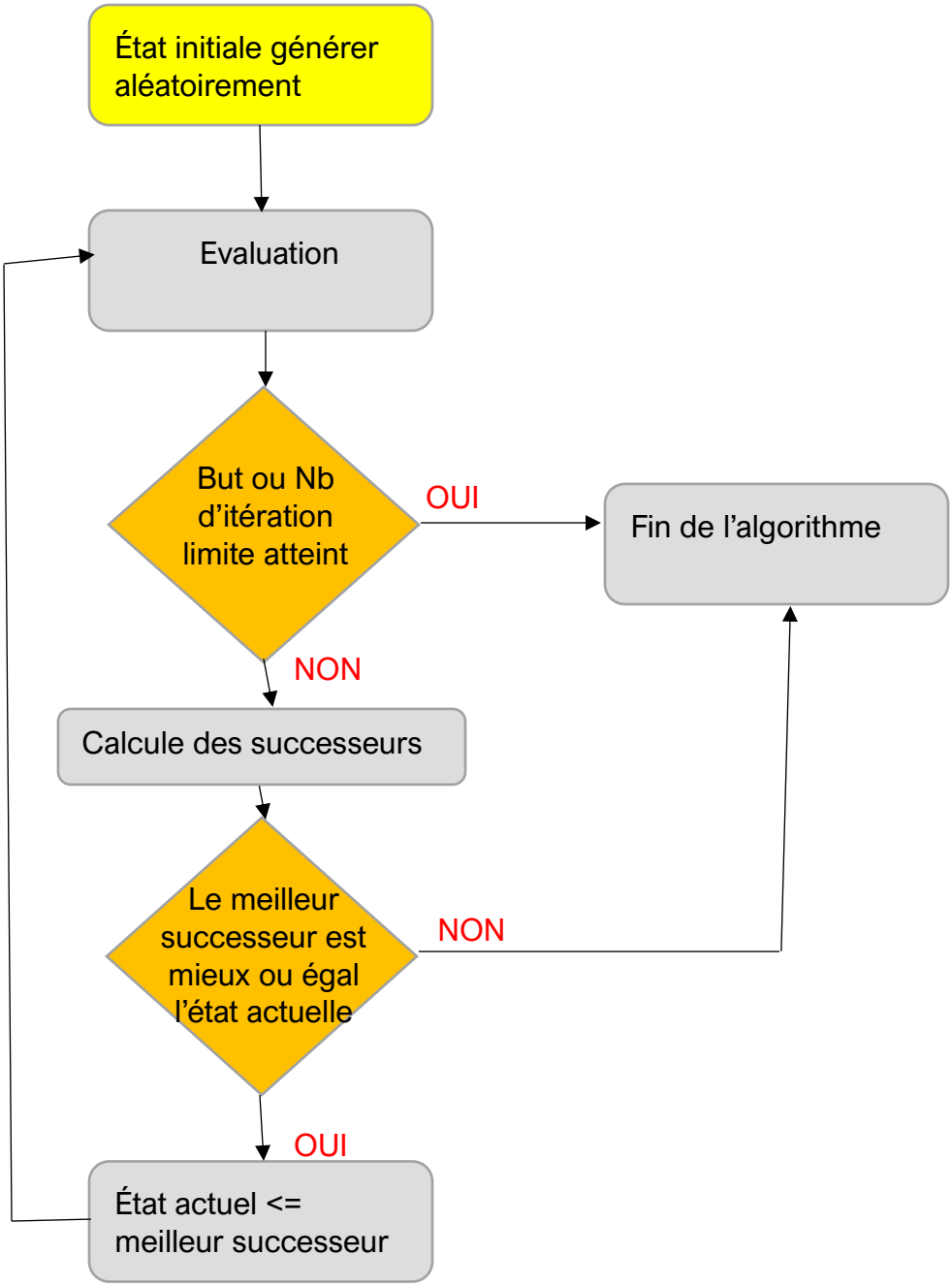
```
Etat : [6, 4, 7, 1, 8, 2, 5, 3]
Nb d'itération : 2
Evaluation (Nb attaque): 0
Temps : 0.01 sec
```

Résultat, Pour N = 30 :

```
Etat : [18, 23, 2, 4, 13, 16, 14, 9, 15, 5, 22, 8, 25, 12, 1, 20, 6, 10, 21, 17, 24, 7, 11, 3, 19]
Nb d'itération : 6
Evaluation (Nb attaque): 0
Temps : 2.696 sec
```

V. L'algorithme Hill Climbing :

La méthode hill-climbing est une méthode d'optimisation permettant de trouver un optimum local parmi un ensemble de configurations.



1. Les successeurs :

La fonction successeurs est similaire à celle de Beam Search.

2. Développement :

On va utiliser trois méthodes :

- Generate_state (génère un seul état)
- Fitness
- Successor

Dans cette méthode on va implémentés le concept de l'algorithme.

Les entrées : les trois fonctions et le nombre d'itération maximal.

Les sorties : un état, le nombre d'itération avant le programme s'arrête et la valeur de l'état retourné.

Python

```python
def run(fitness_funct, successor_funct, generate_State_funct, MaxIteration=200):
    current = generate_State_funct()              #l'état initiale

    for i in range(MaxIteration):           #verification du nombre des itérations

        if  fitness_funct(current) == 0: break                #vérification de l'état but

        successor = successor_funct(current)                  #trouver les successeurs
        successor = sorted(successor,key=lambda state: fitness_funct(state))   #trier par valeur d'evalution

        if fitness_funct(current) < fitness_funct(successor[0]) : #vérifier si le meilleur successeur est moins bon que l'actuelle
            break

        current = successor[0]                #passage état_actuel <= nouveau_état

        #current = random.choices(population=successor,weights=[len(S) - fitness_funct(x) for x in successor],k=1)[0]

    return current,i,fitness_funct(successor[0])
```

On va suivre les mêmes étapes de l'organigramme :

- Générer un état initial aléatoirement
- Vérification du nombre d'itération
- Evaluer l'état actuelle s'il est l'état but
- Trouver les successeurs et les triés
- Vérifier si aucune des successeurs n'est mieux que l'actuelle
- Affectation un nouveau état
- On revient à l'étape 2

3. Test :

On teste l'algorithme avec le code suivant : (random.seed(100))

```
sizePuzzle = 8

start = time.time()
state, iteration, value = run(fitness_funct= fitness,
                              successor_funct= successor,
                              generate_State_funct= partial(generate_Stat,lenght=sizePuzzle),
                              MaxIteration=200)
end =  time.time()

print('Etat :',state)
print('Nb d\'itération :',iteration)
print('Evaluation (Nb attaque):',value)
print('Temps :',round(end - start,3),'sec')
```

Le résultat pour N = 8 :

```
Etat : [4, 2, 7, 3, 6, 8, 5, 1]
Nb d'itération : 2
Evaluation (Nb attaque): 0
Temps : 0.001 sec
```

Le résultat pour N = 30 :

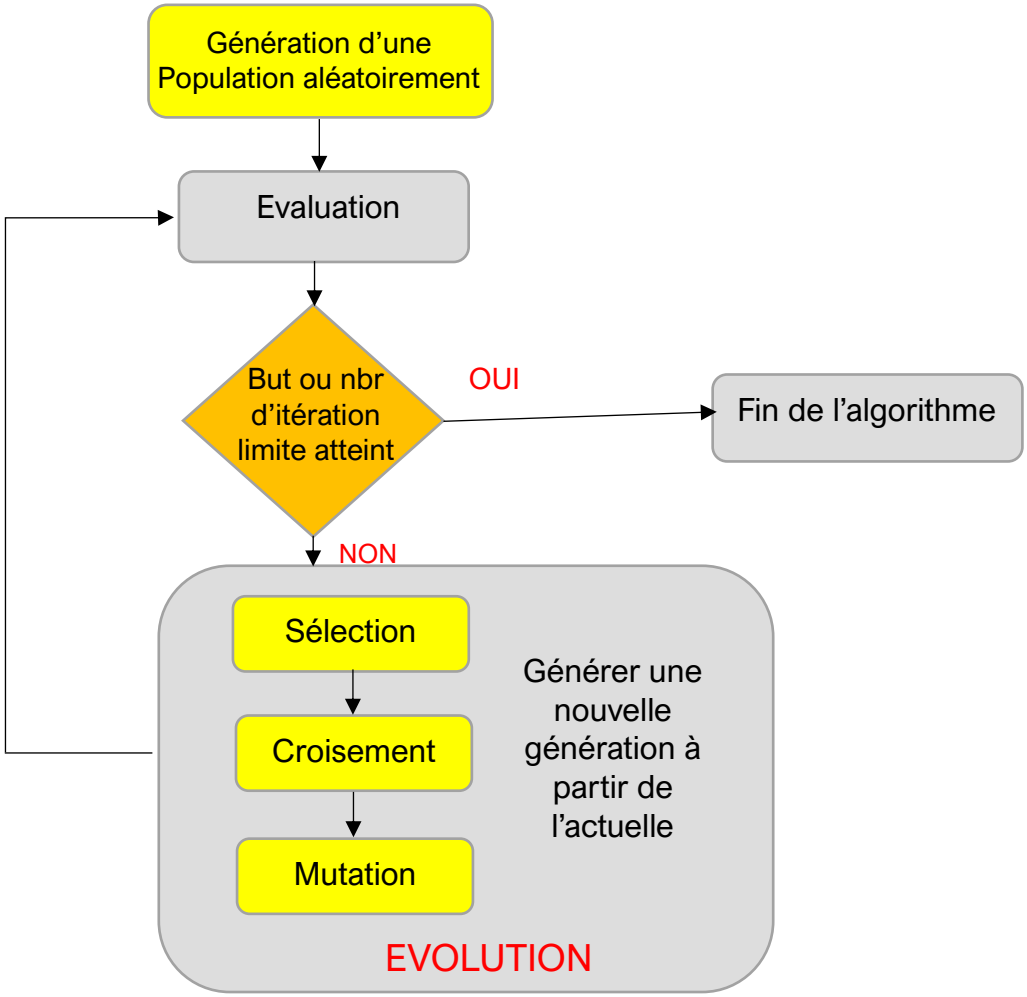
```
Etat : [20, 24, 6, 25, 5, 23, 13, 17, 10, 12, 4, 16, 29, 3, 28, 30, 1, 26, 15, 19, 11,
Nb d'itération : 199
Evaluation (Nb attaque): 1
Temps : 6.287 sec
```

Pour N=30, on n'a pas pu trouver une solution avant atteindre le nombre d'itération limite.

VI. L'algorithme génétique :

Cet algorithme est de la famille des algorithmes dans le principe s'inspire de la théorie de l'évolution pour résoudre des problèmes divers.

Le concept de cet algorithme est le même des autres algorithmes, le fait de générer une population, l'évaluer puis créer une nouvelle génération. Mais L'algorithme génétique est un peu compliqué dans la phase de la génération d'une nouvelle génération, ce qu'on appelle L'évolution.



1. La sélection :

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.

Il existe plusieurs techniques de sélection. Voici les principales utilisées:

- **Sélection par rang :** Cette technique de sélection choisit toujours les individus possédant les meilleurs scores d'adaptation.
- **Sélection par tournoi :** Cette technique utilise la sélection proportionnelle sur des paires d'individus, puis choisit parmi ces paires l'individu qui a le meilleur score d'adaptation.
- **Sélection uniforme :** La sélection se fait aléatoirement, uniformément et sans intervention de la valeur d'adaptation.
- **Roue de la fortune :** pour chaque individu, la probabilité d'être sélectionné est proportionnelle à son adaptation au problème. (Valeur d'évaluation)

Pour notre fonction on va utiliser la roue de la fortune pour sélectionner deux génomes

Python :

```
def selection_pair(population :Population,fitness) -> Tuple[Genome,Genome]:  #selection de deux genomes
    length = len(population[0])
    return random.choices(population=population,weights=[length - fitness(genome) + 1  for genome in population],k=2)
```

Si on se base juste sur la valeur de l'évaluation, qui représente le nombre d'attaque entre les dames, l'état qu'a un grand nombre d'attaque aura une grande chance d'être sélectionné, ce qui est faut.

Alors le poids des génomes est :

$$le\ poids = Le\ nb\ des\ dames - le\ nb\ d'attaque + 1$$

Avec

$$Le\ nb\ des\ dames == la\ longueure\ d'un\ genome$$

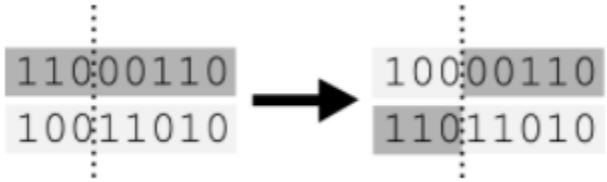
Alors le poids du plus mauvais état (Tous les dames s'attaquent) est 1, et le poids du meilleur état (solution) et N+1.

2. Le croisement

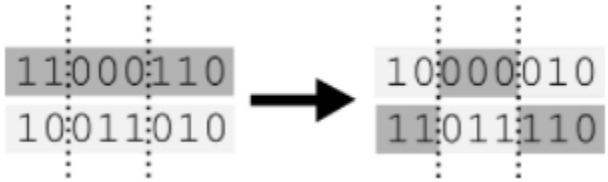
Le croisement ou crossing-over, est le résultat obtenu lorsque deux chromosomes partagent leurs particularités.
Celui-ci permet le brassage génétique de la population.

Il existe deux méthodes de croisement, simple ou double enjambement :

- Le simple enjambement consiste à fusionner les particularités de deux individus à partir d'un pivot.



- Le double enjambement repose sur le même principe, sauf qu'il y a deux pivots :



Dans notre fonction on va utiliser le premiers type de croisement, Mais avec une petite amélioration, pour assurer l'unicité des dames par ligne et colonne lors de du croisement.

On sélectionne un indice aléatoirement entre 2 et N-1, on coupe le premier parent par cet indice.
La première partie du première parent est reporté au fils, et la deuxième partie du fils sera complétés par les gènes du deuxième parent et qu'ils sont absent dans le fils.

Exemple : L'indice = 4

Parent 1 = {3, 2, 4, 1, 5}

Parent 2 = {~~2~~, ~~3~~, 5, 1, ~~4~~}

Fils 1 = {3, 2, 4, 5, 1}

Et on refait la même chose pour le deuxième parent pour produire le deuxième fils.

Exemple : L'indice =4

Parent 1 = {3, 2, 4, ~~1~~, 5}

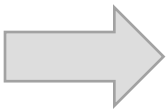
Parent 2 = {2, 3, 5, 1, 4}

Fils 2 = {3, 2, 5, 1, 4}

Résumé :

Parent 1 = {3, 2, 4, 1, 5}

Parent 2 = {2, 3, 5, 1, 4}



Fils 1 = {3, 2, 4, 5,1}

Fils 2 = {3, 2, 5, 1, 4}

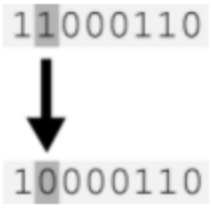
Et ce croisement est fait avec une probabilité P (0.9).

Python:

```python
def single_point_crossover(genome1 :Genome,genome2 :Genome, prob :float=.9) -> Tuple[Genome,Genome]:

    p = random.randint(1,len(genome1)-2)   #nombre aléatoire entre 2 et N-1

    if  random.random() < prob :
        genome1 = genome1[0:p] + [gene for gene in genome2 if gene not in genome1[0:p]]
        genome2 = [gene for gene in genome1 if gene not in genome2[p:]] + genome2[p:]

    return genome1, genome2
```

3. La mutation :

Nous définissons une mutation comme étant l'inversion d'un bit dans un chromosome. Cela revient à modifier aléatoirement la valeur d'un paramètre du dispositif. Les mutations jouent le rôle de bruit et empêchent l'évolution de se figer. Elles permettent d'assurer une recherche aussi bien globale que locale, selon le poids et le



nombre des bits mutés. De plus, elles garantissent mathématiquement que l'optimum global peut être atteint.

La mutation dans notre problème est le remplacement d'une valeur de la liste aléatoirement avec une valeur aléatoire entre 1 et N.

Mais encore, comme la fonction de croisement, on va améliorer cette fonction pour assurer une dame par ligne.

La mutation dans notre cas est la permutation de deux colonnes aléatoirement, càd, choisir deux indices distincts entre 1 et N et permuter leurs valeurs.

Exemple : indice_1 = 2, indice_2 = 4

Avant = {2, 3, 5, 1, 4}

Apres = {2, 1, 5, 3, 4}

PS :

- La mutation n'est effectuée qu'avec une probabilité prob (0.5)
- On peut avoir une double mutation (deux mutations successif) avec prob/2
- On peut choisir le nombre de mutation

```python
def mutation(genome : Genome,number_of_mutation : int=1, prob : float=0.2, check_mutation :bool = True) -> Genome:

    for _ in range(number_of_mutation):
        #Deux nombres aleatoire distincts entre 1 et N
        ind1,ind2 = random.sample(list(range(len(genome))),k=2)

        if  random.random() < prob : #vérifier le seuil
            genome[ind1], genome[ind2] = genome[ind2], genome[ind1] #permutation
            if check_mutation:    #verifier si on est dans la premier mutation
                return mutation(genome,prob=prob*0.5,check_mutation=False) #tenter une deuxieme mutation

    return genome
```

4. Développement

La méthode du développement utilise plusieurs méthodes :

- Generate_population (utilise generate_genome)
- Fitness
- Selection_pair
- Single_point_crossover
- Mutation

Les entrées : la méthode fitness, generate_population et le nombre limite d'itération.

Les sorties : un état, le nombre d'itération avant le programme s'arrête et la valeur de l'état retourné.

Python :

```python
def run_evolution(fitness_funct, Generat_pop_funct,generation_limit :int=100):
    P = Generat_pop_funct() #generer une population

    for i in range(generation_limit):
        P = sorted(P, key=lambda genome: fitness_funct(genome),reverse=True)

        if fitness_funct(P[0])==len(P[0]): #verifier l'etat but
            break

        next_generation = P[0:2]  #elitism

        for j in range(len(P)//2 -1):
            parents = selection_pair(P,fitness_funct) #selection de deux genomes

            g1, g2 = single_point_crossover(parents[0],parents[1]) #croisement
            g1 = mutation(g1) #mutation du genome croisé
            g2 = mutation(g2)
            next_generation += [g1,g2] #ajout à la nouvelle generation

        P = next_generation

    P = sorted(P, key=lambda genome: fitness_funct(genome),reverse=True )

    return P[0],i,fitness_funct(P[0])
```

Dans cette évolution on va ajouter la technique élitiste, qui conserve K=2 meilleurs individues dans la génération suivante.
Cette technique permet de s'assurer que la fonction augmente (déminue) toujours si on maximise (minimise) une fonction.

5. Test :

On teste l'algorithme avec le code suivant : (random.seed(100))

```
sizeP = 20
sizePuzzle = 16

start = time.time()
genome,generation,h = run_evolution( fitness_funct=fitness,
                                     Generat_pop_funct=partial(generate_population,size=sizeP,lenght=sizePuzzle),
                                     generation_limit=200
                                   )
end = time.time()

print('Etat :',genome)
print('Nb d\'itération :',generation)
print('Evaluation (Nb attaque):',h)
print('Temps :',round(end - start,3),'sec')
```

Résultat : N = 8 , PopulationSize =10

```
Etat : [6, 8, 2, 4, 1, 7, 5, 3]
Nb d'itération : 27
Evaluation (Nb attaque): 0
Temps : 0.017 sec
```

Nb d'itération est le numéro de la génération.

Résultat : N = 30 , PopulationSize =50

```
Etat : [27, 21, 13, 15, 14, 6, 2, 22, 30, 26, 10, 5, 20, 18, 9, 4
Nb d'itération : 200
Evaluation (Nb attaque): 2
Temps : 15.744 sec
```

Pas de solution, il reste deux attaques

VI. Comparaison:

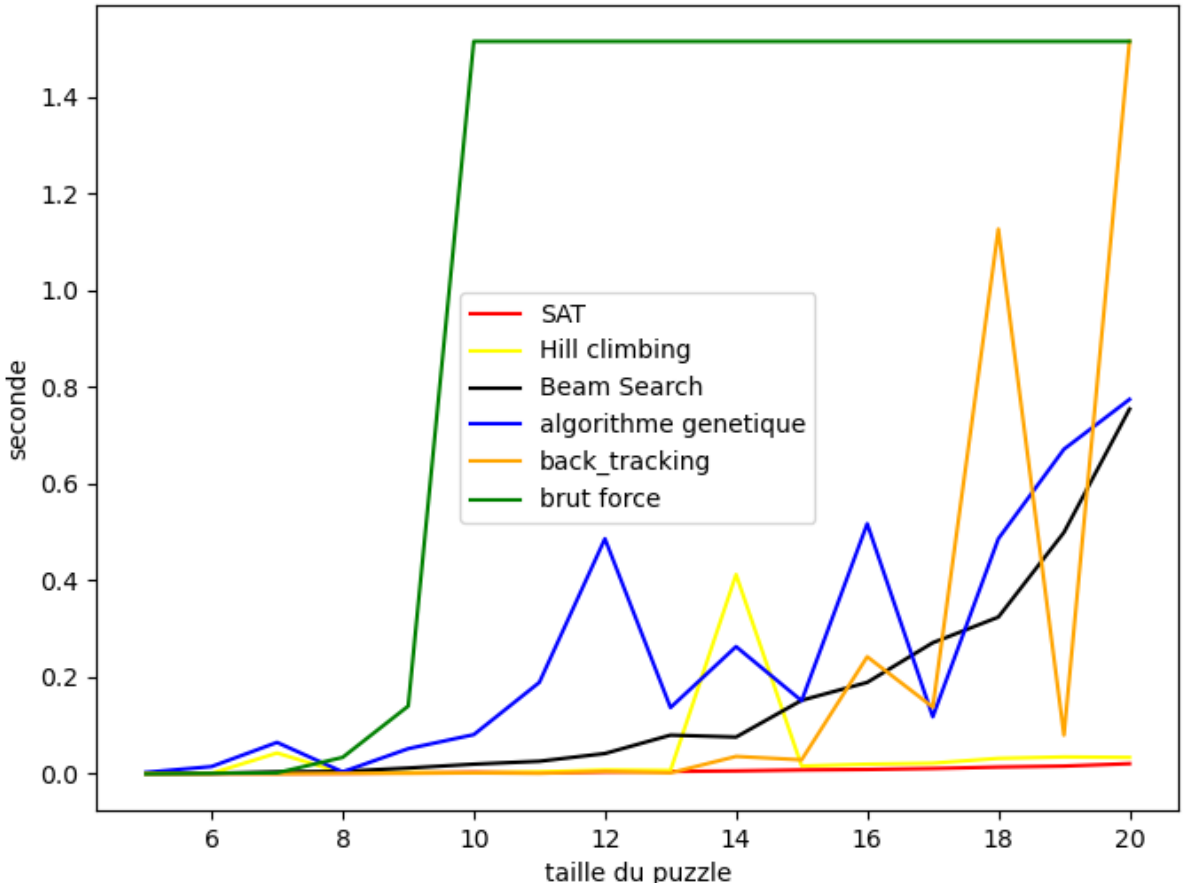
Dans cette partie on va faire une comparaison entre les 3 algorithmes, en plus du SAT Solver, brute force et BackTracking, sur l'intervale N = {5 , 20}

Pour la brute force j'ai utilisé une version un peu améliorer, commence avec un état = {1,2, 3, …, n} et trouver toutes les permutations possibles puis chercher la solution, de cette manier on réduit énormément la taille de l'espace d'état, car on s'assure qu'il a d'une seule dame par ligne et par colonne, pourtant le programme crache pour N = 12.

PS :

- Tous les codes sources des algorithmes sont écrit du zéro sauf le BackTracking.
- La taille de la population de l'algorithme génétique varie entre 10 et 21.

Le graphe :



Pour N = 50, il nous reste que deux candidats :

- Hill Climbing : 2.979 sec
- SAT : 0.329 sec

Pour N = 100, On a un vainqueur :

- SAT : 2,863 sec

Conclusion

J’ai présenté la modélisation du problème de N dames aux plusieurs modèles, Une simple implémentation a pu réduire le temps d’exécution de 3.4 mois (N=12) à une moyenne de 0.04 secondes.

Ce que nous permettre de dire que ces approches sont pratiquement efficaces dans ce type de problème.