Année 2022/2023
Résolution de Problèmes
Master 1, MIV

Série de Travaux Pratiques N°4
Résolution du problème des 8 reines avec l'algorithme génétique

Les 8 reines est un problème informatique classique. Trouver des arrangements possibles de 8 reines sur un échiquier standard 8 x 8 de sorte qu'aucune reine ne se retrouve dans une configuration offensive. Maintenant, si l'on connaît les bases du jeu d'échecs, on peut dire qu'une reine peut voyager horizontalement, verticalement ou en diagonale. Par conséquent, pour que 2 reines ou plus soient dans un état d'attaque, elles doivent se trouver soit horizontalement, soit verticalement, soit en diagonale alignées avec une autre reine. Figure 1 montre l'un des arrangements possibles qui servent de solution au problème des 8 reines.

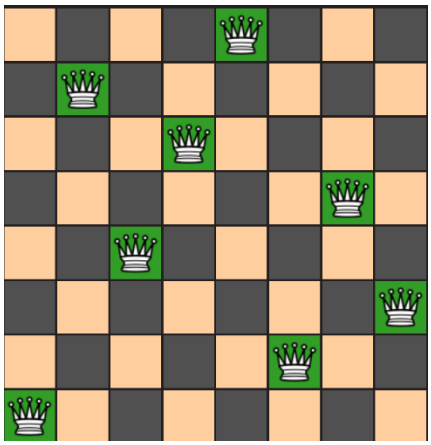

Figure 1 Solution au problème des 8 reines

Il existe différentes méthodes pour résoudre le problème des 8 reines. Le plus courant étant le BackTracking. Il peut également être résolu en utilisant une variété d'approches telles que l'escalade (Hill climbing), l'algorithme génétique, etc. Dans ce TP, nous allons résoudre le problème des 8 reines en utilisant l'algorithme génétique.

L'algorithme génétique est une méthode de résolution de problèmes d'optimisation avec ou sans contrainte qui est basée sur la sélection naturelle, le processus qui conduit l'évolution biologique. L'algorithme génétique modifie à plusieurs reprises une population de solutions individuelles. À chaque étape, l'algorithme génétique sélectionne des individus de la population actuelle comme parents et les utilise pour produire les enfants de la génération suivante. Au fil des générations successives, la population « évolue » vers une solution optimale.

I. **Définition d'un individu :** Un individu est caractérisé par un ensemble de paramètres (variables) appelés Gènes. Les gènes sont réunis dans une liste pour former un chromosome (solution).

Pour cela, nous allons définir une classe **Chromosome**, qui contient les attributs et les fonctions suivants :

1. L'attribut **sequence** : Pour représenter les positions des 8 reines sur l'échiquier, nous allons utiliser une liste de taille 8, où l'indice de chaque case représente la colonne de l'échiquier et le contenu de la case représente la ligne où se trouve la reine. Figure 2 est une représentation de l'échiquier de la Figure 1.

7	1	4	2	0	6	3	5

Figure 2 Représentation de l'échiquier de la Figure 1

2. L'attribut **fitness** : Le score de fitness détermine le niveau de forme d'un chromosome (la capacité d'un chromosome à rivaliser avec d'autres chromosomes). La probabilité qu'un individu soit sélectionné pour la reproduction est basée sur son score de fitness.

3. La fonction de création d'un nouveau chromosome est définie comme suit (voir - *Algorithme 1* -):

```
constructor(seq = None)
Begin
  If seq == None :   // Creation of a new chromosome in the initial population
    sequence = list()
    For i in range(N_QUEENS):
      add(random_number(0, N_QUEENS), sequence)
  else:   // Creation of a new chromosome with crossover
    sequence = seq
  fitness = None
End.
```

- Algorithme 1 -

4. La fonction **setFitness()** : La fonction de fitness est proportionnelle au nombre d'affrontements entre les reines. Il y a $N_{QUEENS} \times (N_{QUEENS} - 1)/2$ affrontements possibles dans un échiquier $N_{QUEENS} \times N_{QUEENS}$. Donc, pour un échiquier 8×8, il y a 28 affrontements. Par conséquent, un chromosome avec une valeur de fitness élevée, il aura un nombre minimal d'affrontements. Le cas idéal peut produire jusqu'à 28 arrangements de paires non attaquantes (Eq 1). Donc, la valeur de fitness maximale est de 28.

$$fitness(chromosome) = N_{QUEENS} \times (N_{QUEENS} - 1)/2 - nombre\ d'affrontements \qquad (1)$$

Pour déterminer le nombre total d'affrontements, pour chaque reine, on doit vérifier 2 types d'affrontements (voir - *Algorithme 2* -):

- Affrontements de lignes ;
- Affrontements de diagonales ;

```
setFitness ()
Begin
  // Number of horizontal collisions
  horizontal_collisions = 0
  For i in range(0, N_QUEENS-1):
    For j in range(i+1, N_QUEENS):
      If sequence[i] == sequence[j]:
```

```
        horizontal_collisions += 1

  // Number of diagonal collisions
  diagonal_collisions = 0
  For i in range(0, N_QUEENS-1):
    For j in range(i+1, N_QUEENS):
      If abs(i-j) == abs(sequence[i]-sequence[j]):
        diagonal_collisions += 1

  // for 8x8 chess: fitness = 28 – (number of collisions)
  Fitness = N_QUEENS*( N_QUEENS-1)/2 – (horizontal_collisions + diagonal_collisions)
End.
```

- Algorithme 2 -

Figure 3 montre un exemple d'un échiquier où le nombre de collisions est égale à 17, d'où la fonction fitness est égale à 11.

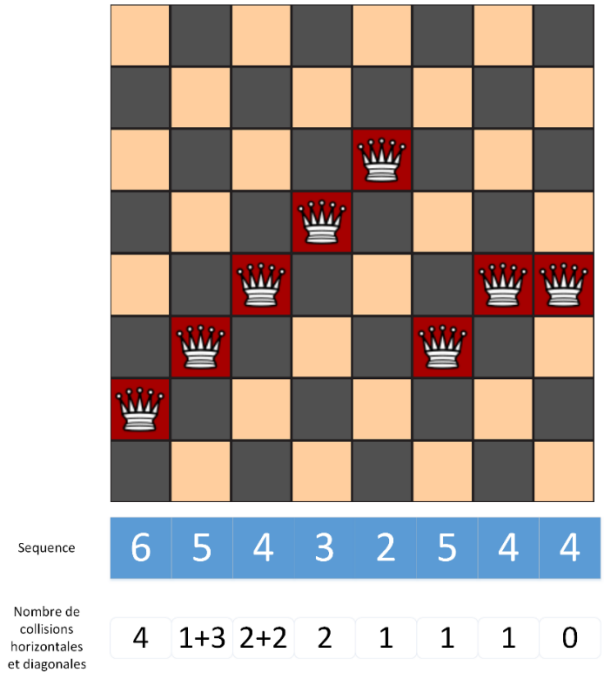

Figure 3 Exemple d'un échiquier avec collisions

5. La fonction **mutate()** : Chez certains nouveaux chromosomes, certains de leurs gènes peuvent subir une mutation avec une faible probabilité aléatoire **MUTATE_PROB** (voir - *Algorithme 3* -). Un exemple d'une mutation est illustré dans la Figure 4.

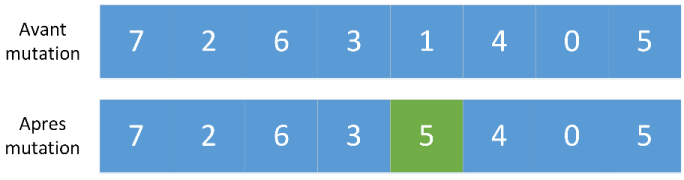

Figure 4 Exemple d'une mutation

```
mutate()
Begin
  p = random_probability()
  If p < MUTATE_PROB:
    col =  random_number(0, N_QUEENS)
    sequence[col] = random_number(0, N_QUEENS)
End.
```

- Algorithme 3 -

II. Etapes de l'algorithme génétique: Cinq phases sont considérées dans l'algorithme génétique :

- Génération de la population initiale ;
- Élitisme (optionnel);
- Sélection des parents ;
- Crossover ;
- Mutation ;

1. **Génération de la population initiale :** Le processus commence par un ensemble d'individus qui s'appelle une population. Chaque individu est une solution au problème que nous voulons résoudre. Pour la population initiale la séquence de chaque chromosome est générée d'une manière aléatoire (voir - *Algorithme 4* -).

```
generateInitialPopulation()
Begin
  population = list()
  For i in range(POPULATION_SIZE):
    chromosome = new Chromosome()
    add(chromosome, population)
    chromosome.setFitness()
End.
```

- Algorithme 4 -

2. **Élitisme :** Une stratégie dans les algorithmes évolutionnaires où la ou les meilleures solutions, appelées les élites, de chaque génération, sont insérées dans la suivante, sans subir aucune modification. Cette stratégie accélère généralement la convergence de l'algorithme (voir - *Algorithme 5* -).

```
elitism(population)
Begin
  sortedPopulation = sorted(population, key=chromosome.fitness, reverse=True)
  newPopulation = list()
  add(sortedPopulation[0], newPopulation)
  add(sortedPopulation[1], newPopulation)
  return  newPopulation
End.
```

- Algorithme 5 -

3. **Sélection des parents:** La sélection des parents se fait d'une manière stochastique en utilisant la méthode du Roulette Wheel. L'idée est de privilégier les individus ayant de bons scores de fitness et de leur permettre de transmettre leurs gènes aux générations successives. Pour cela, on doit calculer pour chaque individu de la population sa probabilité de survie **survivalProb** comme suit (Eq 2) :

$$survivalProb(chromosome_i) = \frac{fitness(chromosome_i)}{\sum_{j=1}^{POPULATION\_SIZE} fitness(chromosome_j)} \quad (2)$$

Après avoir calculer la probabilité de survie de chaque chromosome dans la population, on doit calculer sa probabilité de survie cumulative comme dans l'exemple de la Figure 5. Ensuite, un chromosome est sélectionné comme parent si sa probabilité de survie cumulative est supérieure ou égale à une probabilité aléatoire *p* (voir - *Algorithme 6* -).

	Chromosome 1	Chromosome 2	Chromosome 3	Chromosome 4	Chromosome 5
fitness	28	18	14	9	26
probabilité de survie	0.295	0.189	0.147	0.095	0.274
probabilité de survie cumulative	0.295	0.484	0.631	0.726	1

Figure 5 Exemple de calcul de probabilité de survie cumulative

```
rouletteWheelSelection(population) :
Begin
  // Compute the sum of the fitness scores of the whole population
  sumFitness = 0
  for chrom in population:
    sumFitness += chrom.fitness
  // Compute the survival probability of each chromosome in the population
  survivalProbs = list()
  for chrom in population:
    add(chrom.fitness/sumFitness, survivalProbs)
  // Compute the cumulative survival probabilities
  cumSurvivalProbs = cumulative_sum(survivalProbs)

  // Generate a random probability
  p = random_probability()
  // Select a chromosome with a cumulative probability >= p
  For i in range(POPULATION_SIZE):
    If  cumSurvivalProbs[i] >= p:
      return population[i]
End.
```

- Algorithme 6 -

4. **Crossover:** Cela représente le croisement entre les individus de la population. Deux individus sont sélectionnés à l'aide de la méthode de sélection décrite ci-dessus (Roulette Wheel), et les sites de croisement sont choisis au hasard. Ensuite, les gènes de ces sites de croisement sont échangés, créant ainsi un individu complètement nouveau (voir - *Algorithme 7* -). Un exemple d'une opération de crossover est illustré dans la Figure 6.

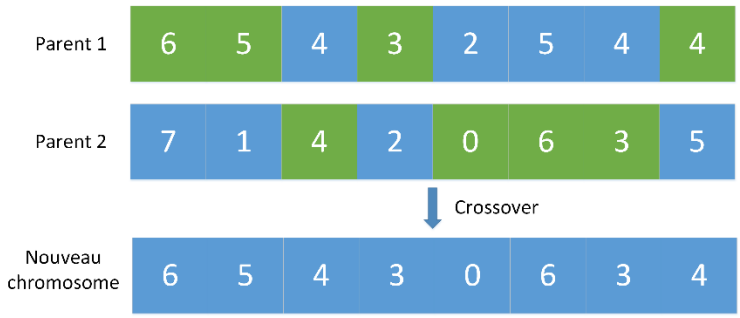

Figure 6 Exemple de crossover

```
crossover(parent1 , parent2)
Begin
  sequence = list()
  For i in range(N_QUEENS):
    p = random_probability()
    If p < 0.5:
      sequence[i] = parent1.sequence[i]
    Else:
       sequence[i] = parent2.sequence[i]
  newChrom = new Chromosome(sequence)
  newChrom.setFitness()
  return   newChrom
End.
```

- Algorithme 7 -

5. **Algorithme génétique:** la fonction de génération d'une nouvelle population avec l'algorithme génétique est définie comme suit (voir - *Algorithme 8* -):

```
GA(population, maxFitness)
Begin
  newPopulation = elitism(population)
  For i in range(POPULATION_SIZE-2):
    parent1 = rouletteWheelSelection(population)
    parent2 = rouletteWheelSelection(population)
    child = crossover(parent1, parent2)
    If MUTATE_FLAG :
      child.mutate()
    add(child, newPopulation)
    If child.fitness == maxFitness:
       break
  return   newPopulation
End.
```

- Algorithme 8 –

III. **Algorithme principal :** La population a une taille fixe. Au fur et à mesure que de nouvelles générations se forment, les individus les moins en forme meurent, laissant de la place à une nouvelle progéniture. La séquence de phases est répétée pour produire des individus dans chaque nouvelle génération qui sont meilleurs que la génération précédente. Ce processus générationnel est répété jusqu'à ce qu'une condition de terminaison soit atteinte. Ici, les deux conditions de terminaison

sont : une solution avec un score maximal de fitness est atteinte ou le nombre maximal d'itérations est atteint (voir - *Algorithme 9* -).

```
// Global variables
N_QUEENS = 8
POPULATION_SIZE = 100
MUTATE_FLAG = True
MUTATE_PROB = 0.1
MAX_ITER = 500
main()
Begin
  // Compute maxFitness value
  maxFitness = N_QUEENS * (N_QUEENS-1) /2

  // Initial population generation
  population = generateInitialPopulation ()

  // Searching for the best solution
  generation = 1
  While (maxFitness != max(population, key=chromosome.fitness))
          and (generation <= MAX_ITER):
     population = GA(population, maxFitness)
     generation += 1

  // Select the best solution from the population
  bestChromosome = max(population, key=chromosome.fitness)
  if bestChromosome.fitness == maxFitness:
     print ('Game solved in generation: ', generation-1)
     print ('Solution: ',  bestChromosome.sequence)
  else:
     print ('Game not solved. Best fitness: ', bestChromosome.fitness)
     print ('One of the best solution: ', bestChromosome.sequence)
End.
```

- Algorithme 9 -