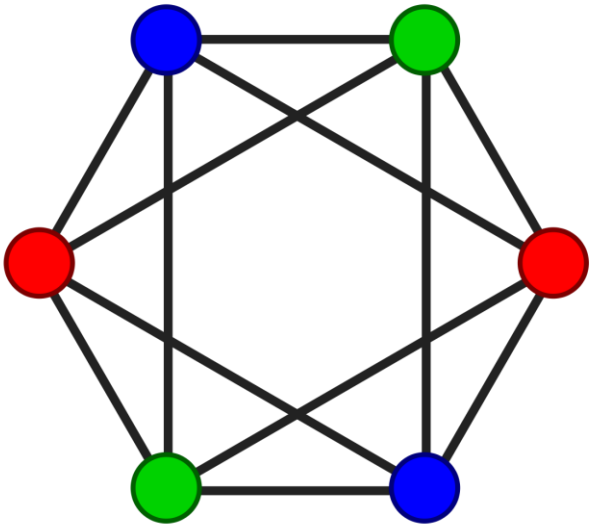
Ecole Polytechnique de Tunisie



RECHERCHE OPERATIONNELLE 2 : APPLICATION DE L'ALGORITHME GENETIQUE POUR LA COLORATION D'UN GRAPHE

Réalisé par :

Heni Masmoudi

Meher Marwani

Élèves ingénieurs en 2éme année à l'école polytechnique de Tunisie

Sommaire:

1. Introduction :

Les graphes sont des modèles très utiles pour représenter des réseaux complexes, et en particulier, la coloration des graphes est l'une des principaux problèmes des mathématiques discrètes, attirant des chercheurs en mathématiques et en ingénierie en raison de ses défis théoriques et de ses applications.

Les algorithmes exacts peuvent actuellement résoudre uniquement des graphes aléatoires de petite taille, avec jusqu'à 80 sommets. Ainsi, les méthodes heuristiques dominent la littérature pour des problémes de coloration générique sur des graphes ayant plus de quelques centaines de sommets. Les algorithmes heuristiques appartiennent essentiellement à trois approches de résolution :

1. Construction séquentielle : méthodes trés rapides mais pas particuliérement efficaces.

2. Recherches locales : Recherche à voisinage variable ou espace de recherche variable.

3. Méthodes à base de populations ou distribuées – Algorithmes hybrides

Les heuristiques peuvent être réparties en trois classes : les heuristiques de construction, les heuristiques d'amélioration et les algorithmes composés (qui combinent les deux premières). Les heuristiques de construction élaborent graduellement la tournée en ajoutant une ville (nœud) à chaque étape. Elles s'arrêtent dès que la solution est trouvée et n'essayent pas de l'améliorer.

La coloration des graphes est l'une des thèmes de recherche actifs dans le Recherche opérationnelle.

Dans ce projet, nous donnons un processus de réflexion à une machine en développant un algorithme génétique permettant de colorer un graphe pour l'utiliser par la suite dans des différentes applications réeles.

2. Définition du probléme de Coloration :

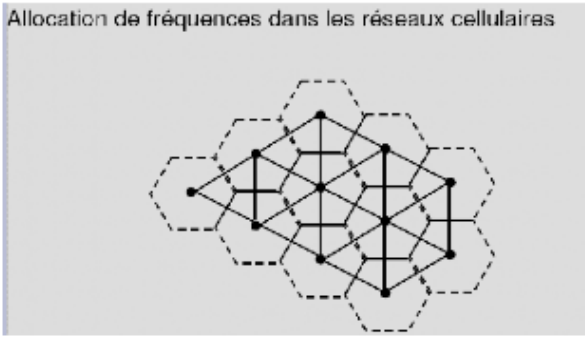
Le problème de coloration des graphes est l'un des problèmes NP-difficiles les plus étudiés et peut être défini de manière informelle comme suit. Étant donné un graphe non orienté, on souhaite colorer avec un nombre minimal de couleurs les nœuds du graphe de manière à ce que deux couleurs affectées à deux nœuds adjacents soient différentes.

If $k = \{1, 2, 3...\}$ and $P(G, k)$ is the number of possible solutions for coloring the graph G with k colors, then

$$\chi(G) = min(k: P(G, k) > 0) \qquad (1)$$

3. Applications réelles du probléme de coloration :

- Le problème d'attribution des fréquences se pose dans le contexte de l'attribution de fréquences radio aux stations d'émission afin de minimiser le brouillage total dans le réseau. Une variante du problème consiste à décrire les fréquences sous forme de couleurs et les transmetteurs sous forme de sommets avec des interférences potentielles résultant de la diffusion de la même fréquence depuis deux émetteurs adjacents
- C'est un probléme de la classe du coloriage de graphe (Graph Coloring Problem).
  Soit un graphe G = (X, E) defini par:
  X: L'ensemble des sommets du graphe représentent les transmetteurs (TRX).
  E: L'ensemble des arêtes du graphe représentent les risques d'interférences.
  Il existe une arête [xi, xj] de E ssi xi est voisin avec x .
  Pour résoudre le probléme d'affectation de fréquences, on utilise la coloration des sommets qui consiste à affecter à tous les sommets du graphe une couleur (fréquence) de façon que chaque paire de sommets soit de couleurs differentes; En d'autre termes, s'il existe une arête [xi, xj] de E alors on a c(i) =6 c(j) (hexagone).
  Le nombre minimum de couleurs nécessaires pour colorier ce graphe en respectant cette contrainte, est appelé le nombre chromatique XG.
  L'application de la théorie des graphes va nous permettre de trouver le nombre minimal de fréquences allouées aux stations de bases et qui minimise l'intégralité des interférences.



Allocation des fréquences dans les réseaux cellulaires

Outre les activités de coloriage au sens propre, la coloration de graphe est utilisée dans de nombreux domaines : attribution d'horaire, organisation d'examens, incompatibilité d'humeurs, approvisionnement de chantiers par différentes routes

4. Perspective géneral des Algorithmes génétiques :

4.1 Définition :

Les algorithmes génétiques (AGs) sont des algorithmes d'optimisation stochastique fondés sur les mécanismes de la sélection naturelle et de la génétique. Leur fonctionnement est extrêmement simple. On part avec une population de solutions potentielles (chromosomes) initiales arbitrairement choisies. On évalue leur performance (fitness) relative. Sur la base de ces performance on crée une nouvelle population de solutions potentielles en utilisant des opérateurs évolutionnaires simples : la sélection, le croisement et la mutation. On recommence ce cycle jusqu'à ce que l'on trouve une solution satisfaisante. Les AGs ont été initialement développés par John Holland (1975). C'est au livre de Goldberg (1989) que nous devons leur popularisation. Leurs champs d'application sont très vastes. Outre l'économie, ils sont utilisés pour l'optimisation de fonctions (De Jong (1980)), en finance (Pereira (2000)), en théorie du contrôle optimal (Krishnakumar et Goldberg (1992), Michalewicz, Janikow et Krawczyk (1992) et Marco et Al. (1996)), ou encore en théorie des jeux répétés (Axelrod (1987)) et différentiels (Özyildirim (1996, 1997) et Özyildirim et Alemdar (1998)). La raison de ce grand nombre d'application est claire : simplicité et efficacité. Bien sûr d'autres techniques d'exploration stochastique existent, la plus connue étant le recuit simulé (simulated annealing – voir Davis (1987) pour une association des deux méthodes). Pour résumer, Lerman et Ngouenet (1995) distinguent 4 principaux points qui font la différence fondamentale entre ces algorithmes et les autres méthodes :

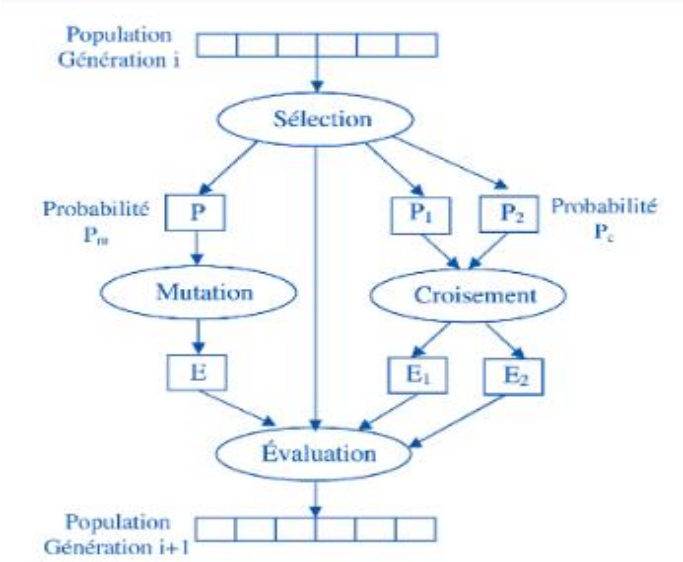
- Les algorithmes génétiques utilisent un codage des paramètres, et non les paramètres eux mêmes.
- Les algorithmes génétiques travaillent sur une population de points, au lieu d'un point unique.
- Les algorithmes génétiques n'utilisent que les valeurs de la fonction étudiée, pas sa dérivée, ou une autre connaissance auxiliaire.
- Les algorithmes génétiques utilisent des règles de transition probabilistes, et non déterministes.

La simplicité de leurs mécanismes, la facilité de leur mise en application et leur efficacité même pour des problèmes complexes ont conduit à un nombre croissants de travaux en économie ces dernières années.

Selon Lerman et Ngouenet (1995) un algorithme génétique est défini par :
– Individu/chromosome/séquence : une solution potentielle du problème.
– Population : un ensemble de chromosomes ou de points de l'espace de recherche.
– Environnement : l'espace de recherche.
– Fonction de fitness : la fonction positive que nous cherchons à maximiser.

Avant d'aller plus loin il nous faut définir quelques termes importants généralement définis sous l'hypothèse de codage binaire.

Définition 1 : (Séquence/Chromosome/Individu (Codage binaire)).
Nous appelons une séquence (chromosome, individu) A de longueur l(A) une suite A ={a1,a2,...,al} avec $\forall i \in [1,l]$, ai $\in$ V ={0,1}. Un chromosome est donc une suite de bits en codage binaire, appelé aussi chaîne binaire. Dans le cas d'un codage non binaire, tel que le codage réel, la suite A ne contient qu'un point, nous avons A ={a} avec $a \in \Re$.
Définition 2 : (Fitness d'une séquence). Nous appelons fitness d'une séquence toute valeur positive que nous noterons f(A), où f est typiquement appelée fonction de fitness.
La fitness (efficacité) est donc donnée par une fonction à valeurs positives réelles. Dans le cas d'un codage binaire, nous utiliserons souvent une fonction de décodage d qui permettra de passer d'une chaîne binaire à un chiffre à valeur réelle : $d : \{0,1\}^l \rightarrow \Re$ (où l est la longueur de la chaîne). La fonction de fitness est alors choisie telle qu'elle transforme cette valeur en valeur positive, soit $f : d(\{0,1\}^l) \rightarrow \Re^* +$. Le but d'un algorithme génétique est alors simplement de trouver la chaîne qui maximise cette fonction f. Bien évidemment, chaque problème particulier nécessitera ses propres fonctions d et f. Les AGs sont alors basés sur les phases suivantes :



1. Initialisation : Une population initiale de N chromosomes est tirée aléatoirement.
2. Évaluation : Chaque chromosome est décodé, puis évalué.
3. Sélection : Création d'une nouvelle population de N chromosomes par l'utilisation d'une méthode de sélection appropriée.
4. Reproduction : Possibilité de croisement et mutation au sein de la nouvelle population.
5. Retour à la phase d'évaluation jusqu'à l'arrêt de l'algorithme.

Voyons maintenant plus en détail les autres phases de l'algorithme génétique. Nous présentons maintenant ces opérateurs.

Les opérateurs génétique :

Opérateur d'initialisation :

Cet opérateur est utilisé pour générer la population initiale de l'algorithme génétique. La population initiale doit contenir des chromosomes qui soient bien répartis dans l'espace des solutions pour fournir à l'algorithme génétique un matériel génétique varié. La façon la plus simple est de générer aléatoirement les chromosomes.

Opérateur de Sélection :

Cet opérateur est peut-être le plus important puisqu'il permet aux individus d'une population de survivre, de se reproduire ou de mourir. En règle général, la probabilité de survie d'un individu sera directement reliée à son efficacité relative au sein de la population.

Opérateur de Croisement :

permet la création de nouveaux individus selon un processus fort simple. Il permet donc l'échange d'information entre les chromosomes (individus). Tout d'abord, deux individus, qui forment alors un couple, sont tirés au sein de la nouvelle population issue de la reproduction.

Puis un (potentiellement plusieurs) site de croisement est tiré aléatoirement (chiffre entre 1 et l−1). Enfin, selon une probabilité Pc que le croisement s'effectue, les segments finaux (dans le cas d'un seul site de croisement) des deux parents sont alors échangés autour de ce site. Cet opérateur permet la création de deux nouveaux individus. Toutefois, un individu sélectionné lors de la reproduction ne subit pas nécessairement l'action d'un croisement. Ce dernier ne s'effectue qu'avec une certaine probabilité. Plus cette probabilité est élevée et plus la population subira de changement.

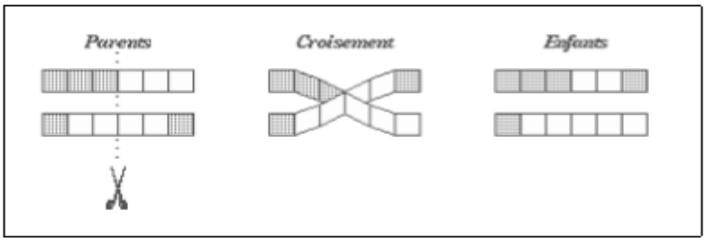


FIG. 2: Le croisement en codage binaire

Opérateur de Mutation :

Le rôle de la mutation consiste à faire apparaître de nouveaux gènes. Cet opérateur introduit une diversité nécessaire à l'exploration de l'espace de recherche en permettant de générer des points dans des régions a priori sans intérêt .La mutation la plus simple sur un chromosome change un un bit de façon aléatoire. Il a de fait un double rôle : celui d'effectuer une recherche locale et/ou de sortir d'une trappe (recherche éloignée).

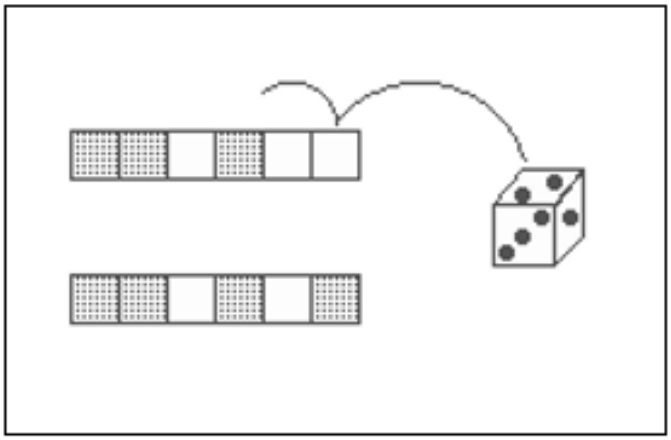


FIG. 3: La mutation en codage binaire

Opérateur Remplacement :

Cette dernière étape du processus itératif consiste en l'incorporation des nouvelles solutions dans la population courante. Les nouvelles solutions sont ajoutées à la population courante en remplacement (total ou partiel) des anciennes solutions. Généralement, les meilleures solutions remplacent les plus mauvaises ; il en résulte une amélioration de la population.

Lorsque la nouvelle population n'est constituée que de nouvelles solutions, on parle d'algorithme génétique générationnel.

Autres paramétres :

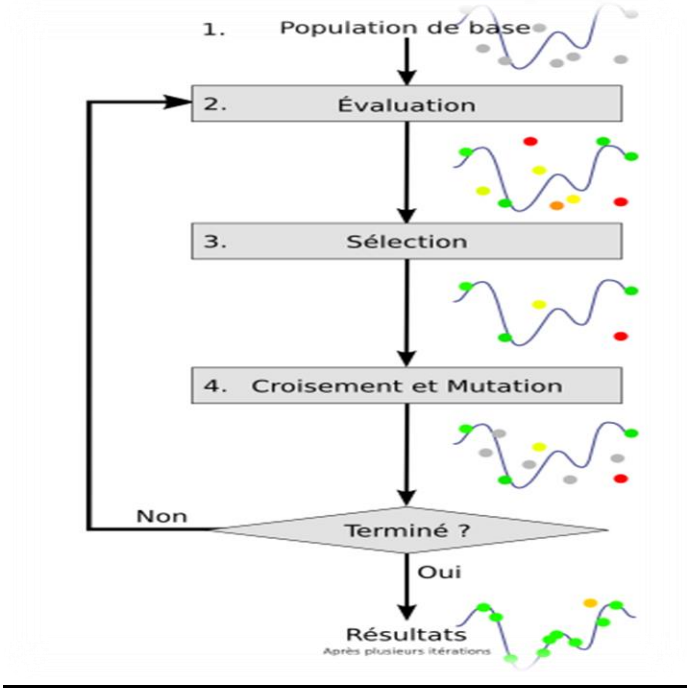
N : la taille de la population,
Pc : la probabilité de croisement,
Pm : la probabilité de mutation,
MAX : le nombre maximal de générations ou la durée maximale attribuée à l'algorithme ou le nombre maximal de générations sans améliorer la meilleure solution trouvée
Q : le nombre de nouveaux chromosomes créés par croisement ou mutation.
D'autres paramètres devront être définis pour la sélection et pour les adaptations spécifiques à un problème donné.

4.2 Avantages et inconvénients :

- Avantages :
  - Le grand avantage des algorithmes génétiques est qu'ils parviennent à trouver de bonnes solutions sur des problèmes très complexes, et trop éloignés des problèmes combinatoires classiques pour qu'on puisse tirer profit de certaines propriétés connues.
  - ce sont des méthodes robustes à l'initialisation (c'est-à-dire que leurs convergences ne dépendent pas de la valeur initiale), qui permettent de déterminer l'optimum global d'une fonctionnelle ou de s'en approcher.
- Inconvénients :
  - Paramètres difficiles à fixer (taille population, % mutation)
  - Choix de la fonction d'évaluation délicat
  - Pas assuré que la solution trouvée est la meilleure, mais juste une approximation de la solution optimale
  - Problèmes des optimums locaux si paramètres mal évalués

5. L'application de l'algorithme génétique au problème de coloration :

5.1 Implémentation de l'algorithme génétique :



1-Population de base :

Dans notre projet, nous avons considéré des chromosomes représentés comme un simple tableau de taille=nombre de nœuds du graphe, dont ces cases contiennent des couleurs aléatoirement générés, telle que leur nombre maximal a été fixé.
Le choix de population de base est une sélection aléatoire des chromosomes de tailles égaux.

```
5     #population = randomly generated chromosomes
6     population=[]
7     for i in range(population_size):
8         cr=[]
9         for i in range(gr.n):
10            #ramdomly choose table of colors
11            cr.append(random.randint(1,numberof_colors))
12        #add to population
13        population.append(cr)
14
```

2-évaluation (Fitness scoring) :

- une mauvaise arrête est définie lorsque deux nœuds ont la même couleur.
- La finesse est le nombre des mauvaises arrêtes (number of bad edges).
- Nous avons crée la fonction fitness pour calculer le nombre de mauvaises arrête, elle prend comme input les paramétres (le graphe, le chromosome(tableau de couleurs)).

```
#define fitness scoring by calculating number of bad edges
def fitness(gr,sample):
        score=0
        for i in range(gr.n):
            for j in range(gr.n):
                if ( (gr.adj[i,j]==1) and (sample[i]==sample[j]) and (i!=j) ):
                    score=score+1
        return int(score/2)
```

Pour trouver le chromosome qui a la plus bonne fitness nous avons crée la fonction Bestfitness qui prend comme paramétres la population et notre graphe :

```
def Bestfitness(population,gr):
    population_size=len(population)
    #evaluate fitness of all population
    allfitnessscore=[]
    for i in range(population_size):
        allfitnessscore.append( fitness(gr,population[i]) )
    bestfitness=min(allfitnessscore)
    return bestfitness,allfitnessscore.index( bestfitness )
```

3- sélection :

La sélection dépend de la valeur de fitness on a à prit comme seuil de la fitness la valeur 4,
Si la valeur de fitness du chromosome est supérieure à 4, on sélectionne les parents par la méthode **parentSelection1**, sinon par la méthode **parent Selection2**.

- **parentSelection1 :**

Algorithm2: parentSelection1:
define: parent1, parent2, tempParents;

tempParents = two randomly selected chromosomes from the population;
parent1 = the fitter of tempParents;

tempParents = two randomly selected chromosomes from the population;
parent2 = the fitter of tempParents;

return parent1, parent2;

Notre implémentation :

```
def parentselection1(population,gr):

    #choose first parent
    x=random.choice(population)
    y=random.choice(population)
    if(fitness(gr,x) <= fitness(gr,y) ):
        parent1=x
    else:
        parent1=y
    #choose second parent
    x=random.choice(population)
    y=random.choice(population)
    if(fitness(gr,x) <= fitness(gr,y) ):
        parent2=x
    else:
        parent2=y

    return parent1,parent2

```

Cette méthode consiste à choisir aléatoirement deux parents (chromosomes). On prend deux pères et on sélectionne celui qui a le fitness la plus petite, c'est le parent1. On répète la même opération pour choisir la mère à partir de deux parents aléatoirement choisis, Parent2.

ParentSelection2 :

Algorithm3: parentSelection2:
define: parent1, parent2

parent1 = the top performing chromosome;
parent2 = the top performing chromosome;

return parent1, parent2;

Notre implémentation :

```
def parentselection2(population,gr):
    #choose  the top performing chromosome
    bestfitness,Index1=Bestfitness(population,gr)
    return population[Index1],population[Index1]

```

Cette fonction choisi les deux parents qui ont la fitnesse la plus petite. La différence entre le selection 1 et 2 est que la selection 2 nous permet de poursuivre la génération avec les deux parents les plus performants.

4-croissement :

Etant donné deux parents sélectionnés le croissement crée un enfant (chromosome de méme taille) qui va poursuivre la génération.

Algorithm4: crossover
define: crosspoint, parent1, parent2, child

crosspoint = random point along a chromosome;
child = colors up to and including crosspoint from parent 1 + colors after crosspoint to the end of the chromosome from parent2;

return child;

Nous avons crée une fonction Crossover qui prend comme input les deux parents sélectionnés, On choisi après aléatoirement **CrossPoint** inférieur au nombre maximale de couleurs, puis on crée l'enfant par le croissement des deux chromosomes séléctionnés. C'est l'union de la premiére partie de parent1 jusqu'à CrossPoint aléatoirement choisi, et la deuxiéme partie de Parent2 à partir de CrossPoint.

```
1  def Crossover(parent1,parent2):
2      CrossPoint = random.randint(0,len(parent1)-1)
3      Child = parent1[:CrossPoint+1] + parent2[CrossPoint+1:]
4      return Child
5
```

5-mutation :

Etant donné un chromosome, la mutation nous permet de génerer un autre chromosome qui peut étre plus performant.

Mutation 1 :

Algorithm5: mutation1:
define: chromosome, allColors, adjacentColors, validColors, newColor;

for each(vertex in chromosome) {
if (vertex has the same color as an adjacent vertex) {
adjacentColors = all adjacent colors;
validColors = allColors – adjacentColors;

newColor = random color from validColors;
chromosome.setColor(vertex, newColor)
}
}
return chromosome;

Notre Implémentation :

```
def mutation1(gr,numberof_colors,sample):
        all_colors=range(1,numberof_colors+1)
        for i in range(gr.n):
            adjacent_colors=[]
            valid_colors=[]
            for j in range(gr.n):
                if (gr.adj[i,j]==1) and (sample[i]==sample[j]):
                    for k in range(gr.n):
                        if (gr.adj[i,k]==1):
                            adjacent_colors.append(sample[k])
            adjacent_colors=list(set(adjacent_colors))
            valid_colors=list(set(all_colors)-set(adjacent_colors))
            if(len(valid_colors)==0):
                print("PROBLEM : Number of colors not enough to color all graph")
            new_color=random.choice(valid_colors)
            sample[i]=new_color
        return sample
```

Nous avons crée une fonction mutation1 qui retourne une mutation d'un chromosome sélectionné comme paramétre. La méthode de mutation est expliqué dans l'algorithme.

Ps : Si le nombre de couleur choisi est insuffisant la fonction déclanche un erreur.

Mutation 2 :

Algorithm6: mutation2:
define: chromosome, allColors

for each(vertex in chromosome) {
if (vertex has the same color as an adjacent vertex) {
newColor = random color from allColors;
chromosome.setColor(vertex, newColor)
}
}
return chromosome;

Notre implémentation :

```
def mutation2(gr,numberof_colors,sample):
        all_colors=range(1,numberof_colors+1)
        for i in range(gr.n):
            for j in range(gr.n):
                if (gr.adj[i,j]==1)and (sample[i]==sample[j]):
                    new_color=random.choice(all_colors)
                    sample[i]=new_color
        return sample
```

Nous avons crée une fonction mutation2 qui retourne une mutation d'un chromosome sélectionné comme paramétre. La méthode de mutation est expliqué dans l'algorithme.

Ps : Le critére de choix de la méthode de mutation est la valeur de fitness.

- Fitness > 4 : mutation1
- Fitness < 4 : mutation2

La mutation résulte un chromosome hybride.

6- L'algorithme génétique géneral :

Notre implémentation :

```
def general_genetic_algorithm(numberof_colors,max_generation_number):
    population_size=50
    mid_point=25
    #population = randomly generated chromosomes
    population=[]
    for i in range(population_size):
        cr=[]
        for i in range(gr.n):
            #ramdomly choose table of colors
            cr.append(random.randint(1,numberof_colors))
        #add to population
        population.append(cr)
    #run general genetic algorithm
    generation_number=0
    best_fitness=351286532
    while( (generation_number<max_generation_number) and (best_fitness!=0) ):
        #evaluate fitness of all population
        best_fitness,Index1=Bestfitness(population,gr)
        #create children
        children=[]
        if (best_fitness>=4):
            for i in range (len(population)+1):
                parent1,parent2=parentselection1(population,gr)
                child=Crossover(parent1,parent2)
                child=mutation1(gr,numberof_colors,child)
                children.append(child)
        else:
            for i in range (len(population)+1):
                parent1,parent2=parentselection2(population,gr)
                child=Crossover(parent1,parent2)
                child=mutation2(gr,numberof_colors,child)
                children.append(child)
        #evaluate fitness of all population
        best_fitness,Index1=Bestfitness(population,gr)
        #update population
        population=sortbyfitness(population,gr)
        for i in range(mid_point,population_size):
            population[i]=children[i-mid_point]
        #add number of generations
        generation_number=generation_number+1
        if(generation_number%100==0):
            print("best fitness found in " ,generation_number,"generation step :",best_fitness)
    print("the algorithm found a solution after :",generation_number,"generation steps with fitness :",best_fitness)
    return population
```

Algorithm1: General Genetic Algorithm
define: population, parents, child

population = randomly generated chromosomes;

while (terminating condition is not reached) {
gaRun();
}

// a single run of a genetic algorithm
function gaRun() {
parents = getParents();
child = crossover(parents);
child = mutate(child);
population.add(child);
}

7- Main code :

1ére Phase : (Choix de l'example)

```
#input example:
#define number of vertex
n=20
#define adjency matrix
adj=np.array(...)
print("adjency matrix:\n",adj)
#graph creation
gr=graph(n,adj)
```

2éme Phase : (Fixation de paramétres et génération de la polpulation finale)

```
#choose parameters
numberof_colors= 20
max_generation_number=1000

#run genetic algorithm with chosen parameters
final_population=general_genetic_algorithm(numberof_colors,max_generation_number)
```

3éme Phase : (éxecution de l'algorithme)

```
#results:
#find the bestchromoson that has fitness=0 with minimum number of colors
index_minimum_colors=0
test=0
for i in range( len(final_population) ):
    if(fitness(gr,final_population[i])==0):
        test=1
        actual_value=len(set(final_population[index_minimum_colors]))
        i_value=len( set(final_population[i]) )
        if( actual_value > i_value ):
            index_minimum_colors=i
#test if the algorithm failed to find correct solution
if(test==0) :
    print("Number of generations is not enough to find a correct solution")
else :
    print("a good solution for the problem using the chromatic number ",actual_value,"is:\n")
    print("befor correction:\n" ,final_population[ index_minimum_colors] )

    #colors rank optimazation
    colors=set( final_population[ index_minimum_colors] )
    final_solution=final_population[ index_minimum_colors]
    k=1
    for x in colors:
        for i in range(len(final_solution)):
            if(final_solution[i]==x):
                final_solution[i]=k
        k=k+1
    print("after correction:\n",final_solution)
    print("fitness error: ",fitness(gr,final_solution))
```

5.2. Test de l'algorithme génétique :

Ps : la correction finale consiste à optimiser le classement des couleurs utilisés.

Premier Test : **(Nombre de Nœuds = 10)**

Phase 1 :

```
nmuber of vertex:  10
adjency matrix:
 [[1 1 1 0 0 1 0 0 0 0]
 [1 1 0 1 0 0 1 0 0 0]
 [1 0 1 0 1 0 0 1 0 0]
 [0 1 0 1 1 0 0 0 1 0]
 [0 0 1 1 1 0 0 0 0 1]
 [1 0 0 0 0 1 0 0 1 1]
 [0 1 0 0 0 0 1 1 0 1]
 [0 0 1 0 0 0 1 1 1 0]
 [0 0 0 1 0 1 0 1 1 0]
 [0 0 0 0 1 1 1 0 0 1]]
```

Phase 2 :

```
the algorithm found a solution after : 5 generation steps with fitness : 0
```

Phase 3 :

```
a good solution for the problem using the chromatic number  3 is:

befor correction:
 [3, 1, 2, 2, 1, 2, 2, 1, 3, 3]
after correction:
 [3, 1, 2, 2, 1, 2, 2, 1, 3, 3]
fitness error:  0
```

L'algorithme trouve la solution exacte (fitness=0) après 5 générations, **la correction finale consiste à optimiser le classement des couleurs utilisés**.

Deuxiéme Test : **(Nombre de Nœuds = 15)**

Phase 1 :

```
nmuber of vertex:  15
adjency matrix:
 [[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]
```

Phase 2 :

```
best fitness found in  100 generation step : 1
the algorithm found a solution after : 115 generation steps with fitness : 0
```

Phase 3 :

```
a good solution for the problem using the chromatic number  15 is:

befor correction:
 [10, 20, 15, 2, 12, 6, 17, 19, 11, 7, 4, 8, 16, 14, 1]
after correction:
 [7, 15, 11, 2, 9, 4, 13, 14, 8, 5, 3, 6, 12, 10, 1]
fitness error:  0
```

Dans les deux tests la correction n'était pas utilisé parceque le nombre de couleur utilisé est Optimal.

6. Conclusion :

- Les algorithmes génétiques peuvent constituer une alternative intéressante lorsque les méthodes d'optimisation traditionnelles (méthode "grimpeur ", méthodes analytiques telle que les moindres-carrés) ne parviennent pas à fournir efficacement des résultats fiables.
- Cependant, comme nous l'avons souvent souligne au cours de ce projet, la plupart des mises en œuvre courantes des AG nécessitent un environnement (le processus ou un modèle de processus) suffisamment tolérant pour laisser à l'algorithme le "droit a l'erreur "
- L'algorithme génétique employé seul constitue un mécanisme de recherche souvent "trop aveugle " alors il peut rendu plus efficace si on le combine avec des méthodes de recherche traditionnelles, habituellement plus locales (régulateur classique, régulateur flou,..).
- Le résultat final est un progrès rapide grimper au sommet de la solution optimale globale.

Bibliorgaphie :

- Genetic Algorithm Applied to the Graph Coloring Problem Musa M. Hindi and Roman V. Yampolskiy Computer Engineering and Computer Science J.B. Speed School of Engineering Louisville, Kentucky
- Solving the graph coloring problem using genetic algorithm Justine W.Shen
- Présentation des algorithmes génétiques et de leurs applications en économie

Thomas Vallée] et Murat Yıldızo˘glu∗

] LEN-C3E Université de Nantes, LEA-CIL Chemin de la Censive du Tertre F-44312 NANTES Thomas.Vallee@sc-eco.univ-nantes.fr ∗IFREDE-E3i Université Montesquieu Bordeaux IV Avenue Léon Duguit F-33608 PESSAC yildi@montesquieu.u-bordeaux.fr

7 septembre 2001, v. 1.2