Projet 3A Moca

Algorithmes génétiques

février 2021

CROGUENNEC Guillaume

DUPONT Ronan

Enseignant : M. Maire

Table des matières

1 Introduction

Quand on regarde dans la nature, on s'aperçoit que tout est bien fait. Toute population est adaptée aux problèmes qu'elle rencontre. Quand une population présente une caractéristique étrange, ce n'est pas seulement dû au hasard, mais c'est bien parce que cela lui est utile ou lui a été utile. Mais pourtant ces "réponses" apportées par la nature à ces populations n'est pas toujours ce à quoi on aurait "logiquement" pensé. Ces "réponses" sont arrivées au hasard, mais ont perduré car elles fonctionnaient mieux que les autres proposées. C'est ce que l'on appelle l'évolution. C'est cette évolution qui trouve parfois en quelques siècles des réponses que l'on n'auraient jamais peut-être trouvées par nous-même.
Mais tout cela concerne la biologie, la manière dont évoluent les espèces. Cela n'a à première vue aucun rapport avec les autres sciences. Mais, dans la seconde moitié du XXème siècle, des numériciens ont eu l'idée d'appliquer méthode de recherche de solutions à des problèmes autres que biologiques.
Leur idée est que pour tout problème où l'on cherche la solution la plus près d'être parfaite (problème d'optimisation), il suffit de proposer une population de solutions créées au hasard, puis de ne garder que les meilleures pour les répliquer plusieurs fois en changeant un peu chaque copie, et de répéter ce procéder autant de fois qu'il le faut. Ainsi à chaque génération de cette population, on aura des individus (qui sont les solutions proposées) de plus en plus adaptés au problèmes.
Avec du temps et de la pratique, on s'est rendu compte que pour de nombreux problèmes cette méthode numérique est la plus précise et la plus rapide.
Parmi ces problèmes, il peut y en avoir où le problème est en fait plusieurs problèmes couplés, ou encore d'autres problèmes où l'on a plusieurs population couplées. Un problème simple qui peut se résoudre par un algorithme génétique est celui de l'optimisation de trajet, ou encore celui de la résolution de sudokus. Mais il existe également des problèmes plus complexes comme des problèmes de thermique où l'on a plusieurs contraintes, auxquelles on doit répondre en bougeant les sources de chaleur ou en modifiant leur température. Enfin, il existe même des problèmes qui peuvent se résoudre de manière plus optimale si l'on modifie un peu ce fonctionnement de l'algorithme génétique, et encore d'autres problèmes où la méthode génétique est loin d'être la plus rapide.

2 L'Algorithme génétique

2.1 Théorie de la méthode

Le principe des algorithme génétique repose sur la génétique des organismes vivants. D'un point de vu informatique, on peut traduire ces algorithmes par les étapes ci-dessous figure 1.

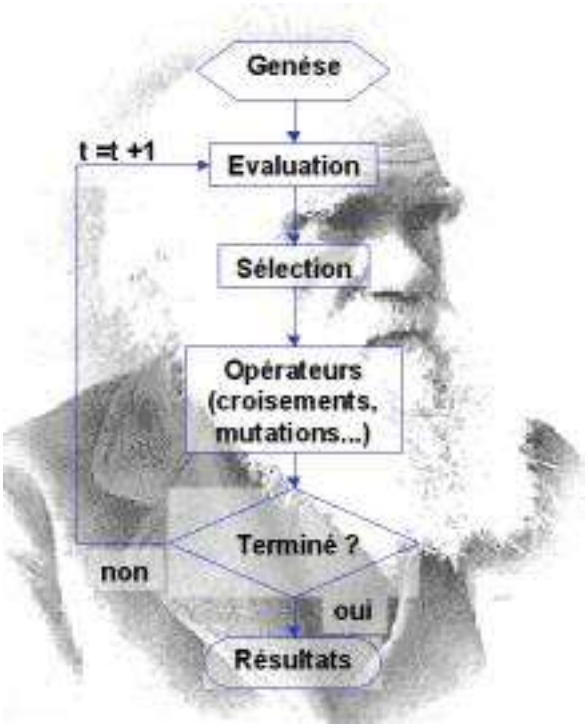

FIGURE 1 – Schéma de l'algorithme génétique

La première étape est donc la **Genèse** ou bien **Création de la population initiale**. Pour un problème donné, on créé une population de N individus où chacun est différent. Un individu est une réponse proposée. Dans le cas de l'optimisation de chemin, un individu est donc simplement un chemin qui relie tous les points incontournables d'une certaine manière. Dans le cas d'une résolution de sudoku, un individu est une certaine manière d'avoir la grille remplie (qui peut être fausse). Dans le cas d'un problème de thermodynamique, un individu est la liste des coordonnées des sources de chaleur et de leur température. Ou encore dans le cas d'une recherche d'un extremum d'une fonction, un individu est simplement une coordonnée.

La seconde étape est celle de l'**Évaluation**. On teste les individus pour savoir s'ils sont de bons candidats afin de ne garder que les meilleurs. Pour la recherche d'extremum c'est simple à faire, plus l'individu donne une valeur grande (ou petite) pour la fonction, mieux il est. Mais pour des problèmes comme celui de la résolution d'un sudoku, il faut procéder un peu différemment. En effet, il faut compter le nombre d'erreur dans la grille, et moins il y en a et mieux est l'individu.

La troisième étape est celle de la **Sélection**. Il y a plusieurs manières de procéder mais on retient principalement la sélection sous forme de "**duel**" entre deux individus, car son coût en calcul est moindre. Le principe est d'évaluer les caractéristiques de deux individus pour ne garder que le meilleur des deux : "celui qui a les meilleures gênes". Avec l'exemple de la recherche du minimum d'une fonction, on évalue entre deux individus tirés aléatoirement dans la population celui qui donne le plus petit résultat dans la fonction, et on le garde. Les autres modes de sélection répandus que l'on n'utilise pas à cause de leur coût en calcul sont la sélection par rang et la

sélection par probabilité proportionnelle à l'adaptation. Le principe de la première est que l'on classe tous les individus et on ne garde que les meilleurs. Le principe de la seconde est que chaque individu a une probabilité d'être gardé qui est proportionnelle à son adaptation. C'est sûrement cette sélection qui s'inspire le plus de la vrai sélection naturelle.

La quatrième étape est celle de l'**Opérateur**. C'est la manière dont on choisit les individus qui vont venir remplacer les inadaptés dans la population. Il existe plusieurs types d'opérateurs comme :
- **Les croisements** : on peut faire un mélange de gênes entre plusieurs individus gardés (souvent les plus adaptés) en prenant une certaine proportion des gênes de l'un et une certaine de l'autre.
- **Le clonage** : On prends tous les gênes d'un individu pour les mettre sur un autre. C'est très souvent suivi d'une mutation car sans cela il n'y a pas d'évolution de la population
- **Les mutations** : La suite du clonage. On change quelques caractéristiques de chaque clone en espérant faire un individu meilleur que l'original. Par exemple si on se place dans le cas du duel, on prends l'individu le moins adapté des deux et on en fait une mutation du meilleur des deux. A savoir que dans le cas de la recherche d'un extremum d'une fonction, une mutation est une coordonnée proche de la coordonnée gagnante.

Enfin, dans les cas où on cherche une solution parfaite (comme pour la résolution d'un sudoku), on regarde à chaque génération (donc chaque itération) s'il existe un individu dans la population qui satisfait les conditions de notre problème. Si tel est le cas, on arrête les itérations et on choisit l'individu candidat. Si ce n'est pas le cas, on continue les itérations comme on peut le voir sur le schéma figure 1.
Mais dans la majorité des cas, on ne peut pas dire exactement si notre candidat est la solution ou non. En effet, parfois on ne connaît pas la solution exacte à l'avance, donc on a du mal à évaluer si le candidat est le meilleur ou non. Dans ces cas-là, on attend que toute la population ait convergé vers la même solution. Mais en reprenant encore une fois la recherche du minimum, comment savoir si il s'agit du minimum global ou d'un minimum local ?
En fait, on ne peut pas simplement le deviner. La méthode est d'avoir une population assez grande pour être sûr que statistiquement un des individus de départ n'est pas trop loin de la solution. Mais il ne faut pas non plus que la population soit trop grande sinon il faudra beaucoup d'itération pour que la population converge vers la solution.
En résumé, ce qui change entre les différents algorithmes génétiques c'est la sélection, l'opérateur, le nombre d'individus choisis lors de la genèse et, dans certains problèmes, la manière dont l'on quantifie l'erreur.

2.2 Temps de calcul

Empiriquement, on sait que la fonction du temps de calcul d'un algorithme génétique en fonction de la taille de la population est convexe. En effet, au début plus la population est grande, et plus le temps de calcul est court, jusqu'à atteindre son minimum. Puis à partir de ce minimum, plus la taille de la population augmente, et plus le temps de calcul augmente également.

3 Application simple : Problème du voyageur de commerce

En informatique, le problème du voyageur de commerce (ou d'optimisation de trajet) a une importance cruciale car il est à la base de la théorie des graphes qui a des applications dans de nombreux domaines comme les réseaux sociaux, les réseaux informatiques, les télécommunications, etc...
Et sa résolution par algorithme génétique est l'une des plus efficaces.

3.1 Description du problème

Le problème du voyageur de commerce est un problème d'optimisation très connu. L'idée du problème est qu'un commerçant souhaite parcourir N villes pour faire des ventes. Or, celui-ci souhaite revenir à son point de départ après son voyage et parcourir le moins de distance possible. Le problème consiste donc à relier un nombre de villes N entre elles pour en faire le circuit le plus court.
On peut illustrer ce problème avec 4 villes par l'illustration obtenue dans [2] ci-dessus figure 2.

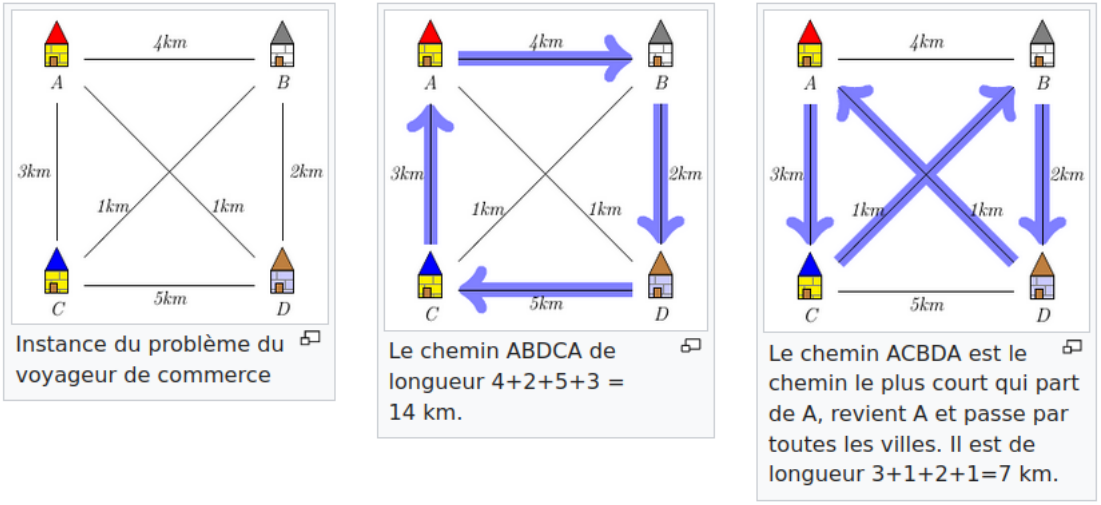

FIGURE 2 – Problème du voyageur pour 4 villes

Ce problème peut paraître simple en apparence, mais c'est un réel problème d'optimisation du fait de sa complexité.
En effet, plus on augmente le nombre de villes, plus on aura du mal à trouver le chemin le plus court. Si l'on pose le nombre de villes N, alors il existera un nombre total de chemin égal à $N!$. Or, le chemin de départ ne peut pas changer la longueur du chemin (par exemple, aller du point A à A n'a aucun intérêt...) et le sens du chemin n'a pas d'importance. Il y a donc $\frac{1}{2}(N-1)!$ chemins candidats. Si l'on voulait résoudre cet problème par une méthode "naïve" (en testant tous les chemins candidats possibles), on serait très rapidement limité ! Par exemple pour 15 villes, on aurait 43 589 145 600 chemins candidats ; sachant que l'ordinateur utilisé effectue (sous fortran), 8 millions d'opérations à la seconde... Il faudrait des heures pour calculer toutes les distances de tous les circuits. Deux scientifiques, *Held* et *Karp* ont réussi à l'aide de la programmation dynamique à résoudre le problème avec une complexité en $O(n^2 2^n)$ ce qui est nettement mieux mais toujours insuffisant...
C'est pourquoi on utilise une méthode qui est des plus optimisée pour résoudre ce problème : celle des algorithmes génétiques.

3.2 Résolution par algorithme génétique

Le principe de la résolution de ce problème par algorithme génétique consiste à générer une population de N circuits et à chaque itération, tirer deux circuits parmi les N, puis sélectionner le chemin le plus long pour y affecter une mutation du chemin le plus long. Ici la mutation consistera simplement à permuter deux villes dans l'ordre du circuit.
On essaye, dans un premier temps, de résoudre ce problème sur un domaine simple en 2D : $[0,1]^2$. Pour ce qui est des villes, on les génère simplement avec une subroutine *random* générant des aléatoires dans [0,1]. La résolution est assez simple, et on se rend compte que même en appliquant le problème à un domaine en trois dimensions cela reste quasiment aussi simple.
Pour appliquer ce problème à un cas plus concret, on prend le cas de la France. Dans cette application on peut faire varier le nombre de villes utilisées pour observer les résultats comme on peut le voir ci-dessous figure 3.

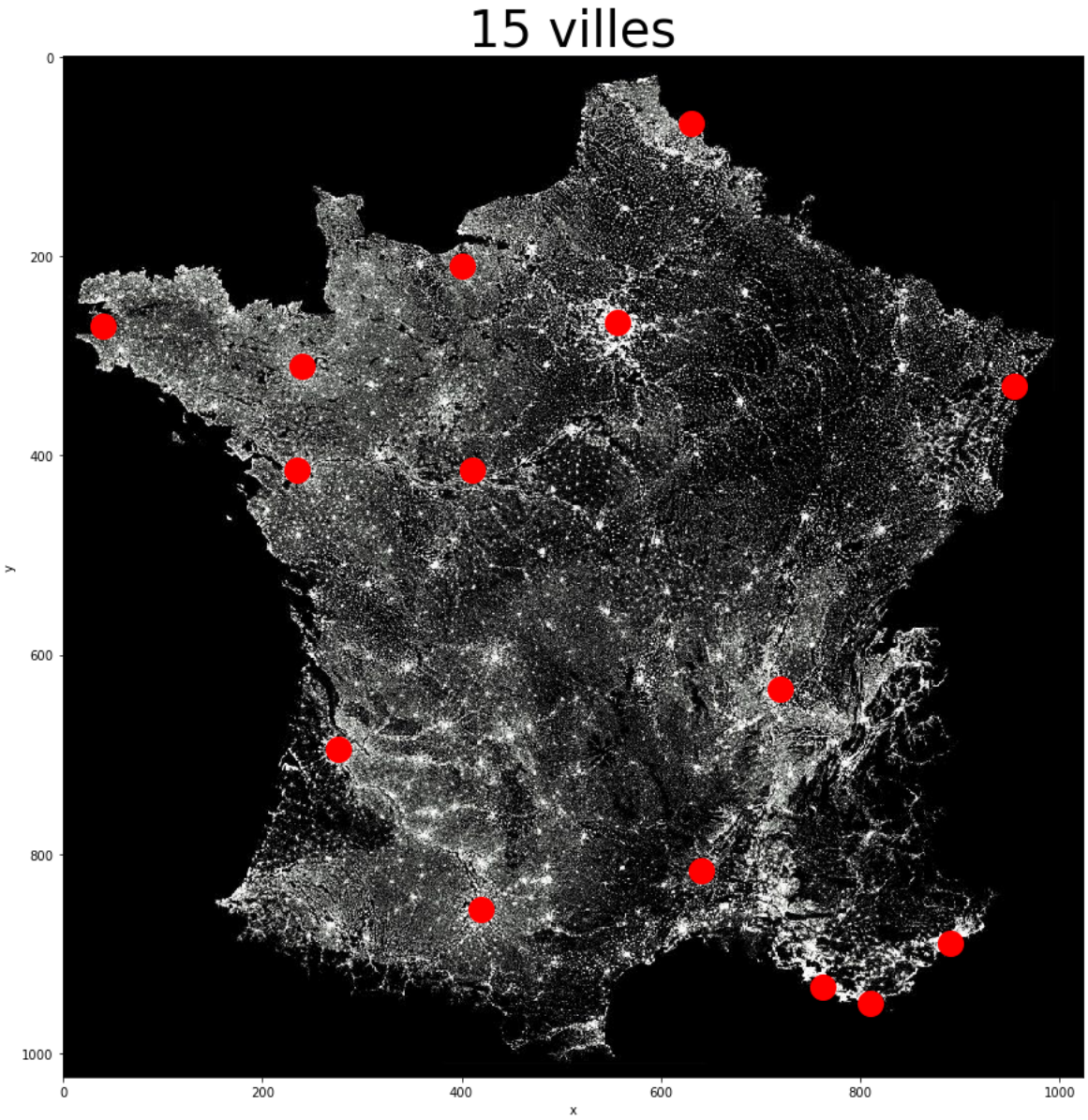

FIGURE 3 – La France avec 15 grandes villes

3.2.1 Génèse

Pour ce problème, il est nécessaire de prendre une population initiale assez grande étant donné le nombre de chemins possibles qui augmente très rapidement. On fixe cette population initiale à 1000 circuits différents. Cela est assez grand pour aboutir à la solution, mais pas trop grand pour que la population converge vers cette solution.

3.2.2 Sélection et opérateur

Pour ce problème, l'idée de la mutation est de provoquer un duel entre deux chemins, sélectionner le chemin le plus long, le cloner en l'autre et permuter 2 villes. Cela fonctionne mais on peut améliorer les mutations comme par exemple en échangeant et en inversant l'ordre d'ensembles aléatoires de 2 villes consécutives (par exemple les villes 3-4 avec les villes 7-8 en échangeant 3 avec 8 et 4 avec 7).Une autre idée serait de permuter plus de 2 villes ou de définir des hybridations, mais les résultats ne sont pas satisfaisants.

3.3 Résultats

Le but est donc de relier les villes de manière à avoir le plus court chemin à "vol d'oiseau". Mais, au départ, on ne sait pas jusqu'à combien de villes le problème pourra être résolue rapidement.

3.3.1 Choix du nombre d'itérations sur le domaine [0,1]

Pour cela, on commence par regarder pour 10 villes (car il y a seulement 181 440 chemins possibles) les types de chemins trouvés en fonction du nombre d'itérations. Voici ci-dessous figure 4 les résultats obtenues pour une population initiale de 1000.

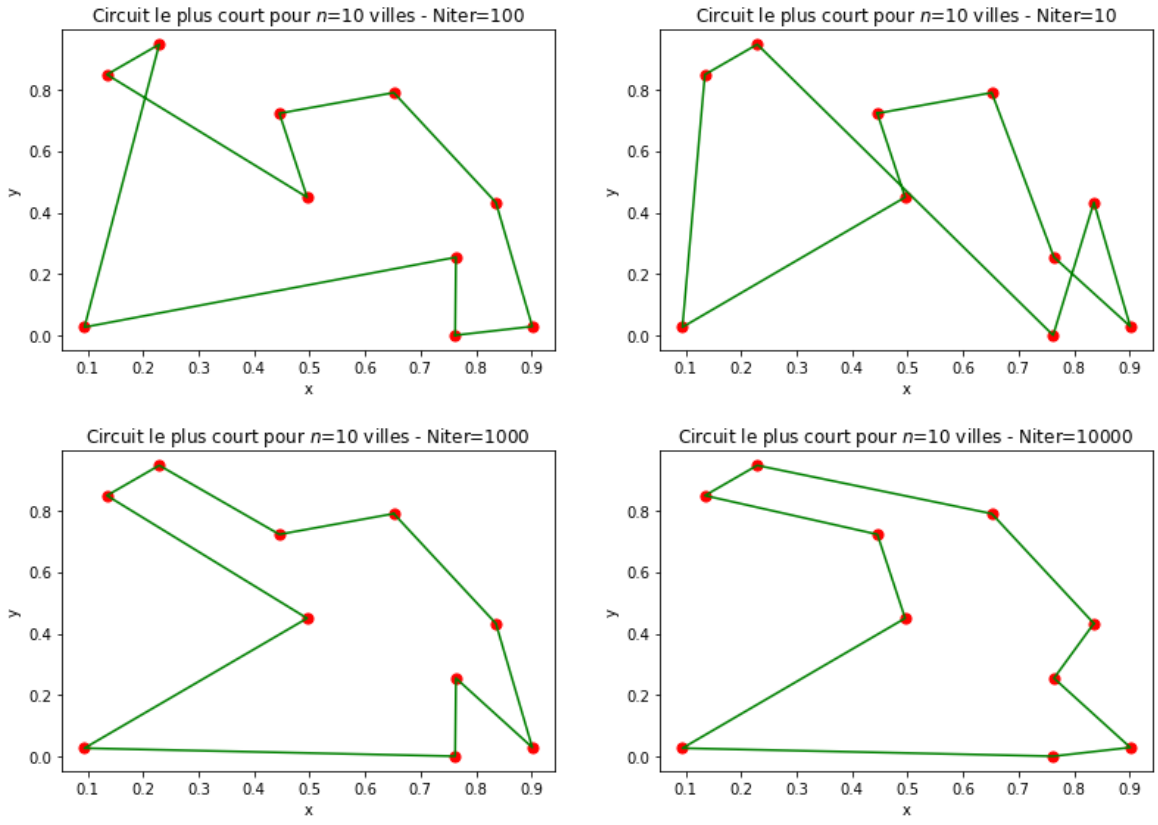

FIGURE 4 – Circuit le plus court en fonction des itérations

Les deux premiers circuits ne peuvent pas être les plus courts car il y a un ou plusieurs croisements (et qu'on traville ici sur des chemins directs où le plus court trajet entre deux villes est le segment qui les relie). En continuant les itérations, on remarque que le circuit à 1000 itérations est bien le plus court.

3.3.2 Influence du nombre de villes

Le fait de savoir qu'il faut un millier d'itérations pour résoudre le problème à 10 villes nous donne un ordre de grandeur de ce à quoi on peur s'attendre. On fixe donc le nombre d'itérations à 1 000 000 et la population initiale à 1 000 puis en faisant simplement varier le nombre de villes (20, 30, 50 et 100) on obtient les chemins suivants figure 5 :

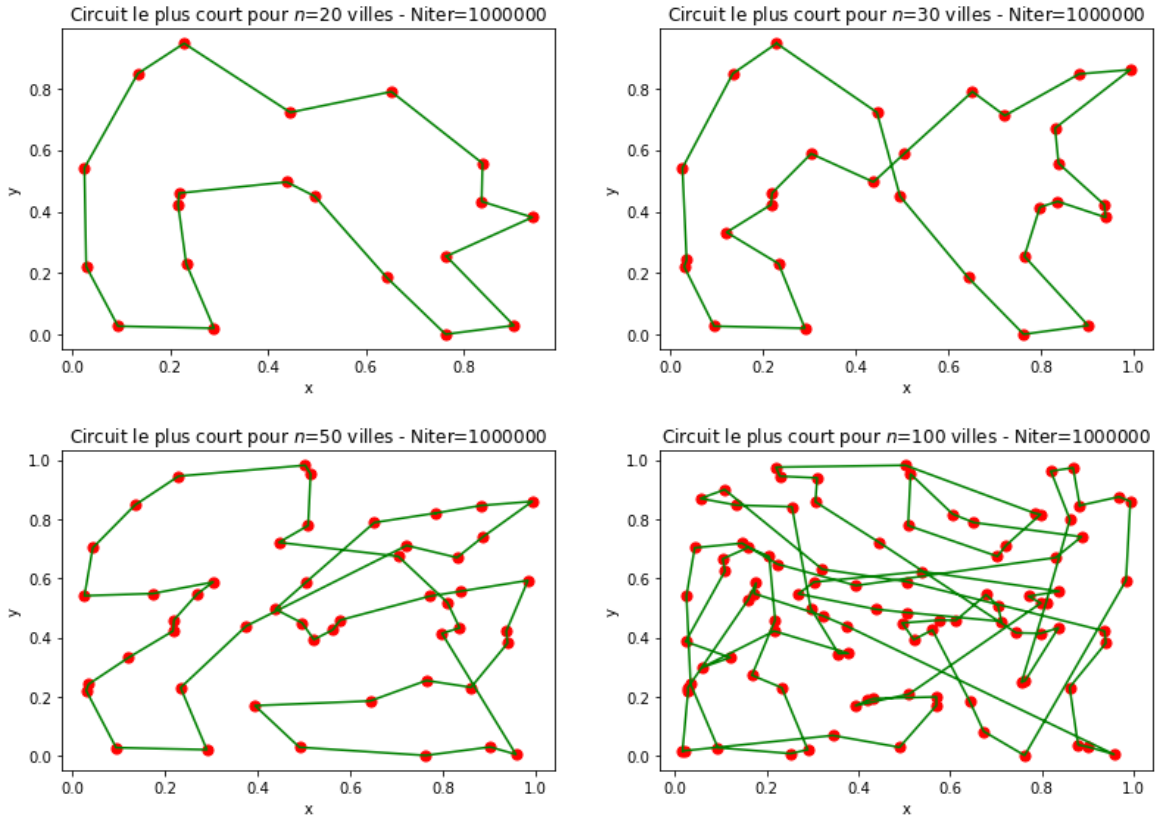

FIGURE 5 – Circuit le plus court en fonction du nombre de villes

On remarque que les chemins deviennent rapidement aberrants dès 30 villes. Même si l'on n'observe pas toujours de croisement, on voit bien que le chemin n'est pas le plus court. Si l'on avait à trouver le chemin le plus court pour 100 villes, cela prendrait sûrement des centaines d'heures de calculs étant donné que pour trouver ce chemin loin d'être bon il a fallu plus d'une minute.

Étant donné que 30 villes est trop pour obtenir une solution rapidement, on peut choisir de résoudre ce problème sur 15 villes de France.

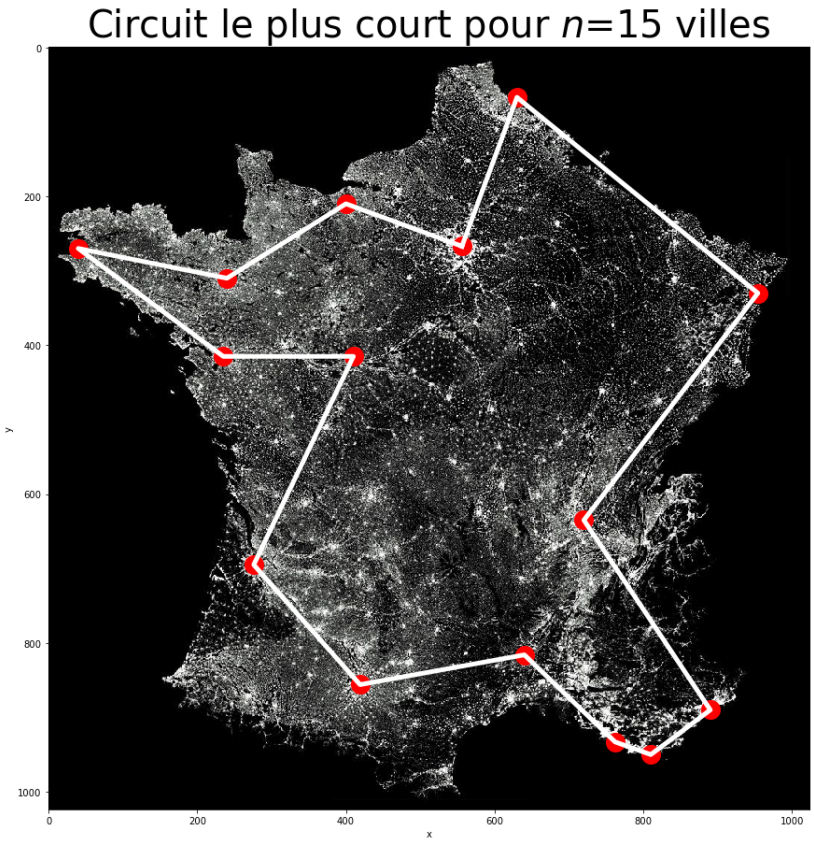

FIGURE 6 – La France avec 15 grandes villes

On observe sans trop de surprises le chemin ci-dessus. Il faut rappeler que ce problème est une optimisation de trajet à "vol d'oiseau" qui ne tient donc pas compte des routes de France. Cette distance minimale pour 15 villes est donc de 3 368 km.
On pourrait aller plus loin en travaillant sur des cartes routières de France mais cela demanderait sûrement un travail de traitement d'image assez laborieux pour forcer les traits à suivre la route.

4 Cas du sudoku : l'algorithme génétique est-il toujours le plus rapide ?

Le sudoku est un jeu en forme de grille apparu en 1979. Le but du jeu est de remplir la grille avec une série de chiffres tous différents, qui ne se trouvent jamais plus d'une fois sur une même ligne, dans une même colonne ou dans une même bloc. La plupart du temps, les symboles sont des chiffres allant de 1 à 9, les blocs étant alors des carrés de 3×3. Quelques symboles sont déjà disposés dans la grille, ce qui autorise une résolution progressive du problème complet.
Sa résolution par algorithme génétique est intéressante car il existe environ 6 671 milliards de milliards de grilles de sudoku possibles. De plus, chaque grille de sudoku a une difficulté notée avec un coefficient qu'on appelle SER (Sudoku Explainer Rating). Utiliser une méthode de type "naïve" en essayant chaque combinaison serait donc inutile pour résoudre ce genre de problème.
On résout donc dans un premier temps ce problème par la méthode génétique, avant de comparer cette résolution avec une autre bien connue pour ce genre de problèmes : le backtracking.

4.1 Méthode génétique

Encore une fois, la difficulté de cette méthode est de choisir la bonne genèse, la bonne sélection et le bon opérateur comme expliqué dans la partie 2 à l'aide de [3].

4.1.1 Genèse

Pour cette étape, on créé N individus qui sont des grilles générées aléatoirement mais avec des chiffres fixés dans certaines cases pour respecter l'énoncé du sudoku. L'idée est qu'en initialisant le programme, un tableau des coordonnées des cases non vides soit sorti, auquel est associé un autre tableau avec tous les chiffres possibles en fonction de la grille initiale. Cela permet d'alléger la complexité du programme pour la suite.

4.1.2 Sélection

Pour évaluer si une grille est bonne ou mauvaise, on ne peut pas comparer avec la grille exacte car elle n'est pas sue. On se base donc sur les règles du sudoku pour créer une fonction **coût**. Cette fonction calcule sur chaque ligne, colonne et bloc combien de chiffre il manque. Et pour chaque chiffre manquant l'erreur augmente de 1.
Mais on l'améliore pour que le programme soit plus rapide. Pour cela on ne calcule à chaque itération que les erreur soit sur les lignes, soit sur les colonnes, soit sur les blocs, et cela de manière alternée. Et quand il y a une erreur sur l'un des trois qui vaut 0 on commence à calculer les erreurs sur les trois cumulés. Cela permet à chaque itération d'être bien plus rapide.
Ensuite, à chaque itération, chaque grille (ou individu) est en duel avec une autre grille. Et c'est la grille avec l'erreur moindre qui gagne le duel.

4.1.3 Opérateur

Puis l'opérateur choisi est le clonage suivi de la mutation. C'est-à-dire que le gagnant du duel reste le même, tandis que le perdant devient une presque-copie du gagnant, autrement dit il est une copie qui a subi une mutation.
Cette mutation se traduit par un changement de quelques cases tirés au hasard, mais ce doit être forcément des cases qui ne sont pas fixes. A force d'essaies on s'aperçoit que le mien est de changer 3 cases non fixes aléatoirement. Mais il est en fait possible d'améliorer cette méthode en choisissant aléatoirement de changer, soit 2, soit 3 cases non fixes au hasard.
Toutes ces étapes sont donc réitérées jusqu'à ce qu'une grille ait une erreur totale de 0. Et cette grille est donc la solution du sudoku.

4.2 Résultats de la méthode génétique

Afin de quantifier les résultats en terme de temps de calcul, nombre d'itérations, etc... On a besoin de quantifier la difficulté des sudokus. Pour cela, il existe un coefficient appelé SER. Mais par manque de documentation sur le net, il n'est pas possible d'exploiter ce coefficient.
On invente donc deux critères de difficulté des sudoku. Il s'agit du nombre de cellules non remplies sur la grille ainsi que du nombre de possibilités moyennes de chiffres possibles sur chaque cellule non remplie.Avec ces deux coefficients, il est donc possible de quantifier la difficulté d'un sudoku.

4.2.1 Choix du nombre de grilles (ou d'individus) N dans la population initiale

Après avoir essayé avec de nombreuses grilles, on s'aperçoit qu'il est nécessaire d'avoir un nombre de grilles initiales suffisant afin d'obtenir la convergence de la population. Pour ce qui est des grilles faciles, on peut avoir une convergence vers la solution exacte à partir de 3 grilles de départ ; mais cela n'est pas suffisant pour les grilles les plus difficiles.
Il est en fait nécessaire d'avoir à minima 40 grilles pour obtenir la convergence de n'importe quelle grille. En prenant un nombre $N \in [|20, 60|]$, on a très souvent une convergence des plus rapides quelque soit le sudoku.

4.2.2 Temps de calcul et nombre d'itérations

Afin de comparer les temps de calculs et nombres d'itérations jusqu'à convergence vers la grille exacte, on travaille sur les grilles suivantes figures 7 8 9 10 11 où l'on indique leur niveau de difficulté en dessous. Leur temps de calcul et leur nombre d'itérations est inscrit en-dessous de chacun.

4.2.2.1 Grille facile

4	8	7						2
1				7			5	8
3			4		8	1	7	
		3		2	5		6	
	9		6	8				3
		6	3		4	5		7
		5			6	7		
	3	8	7				1	5
2	7			9	1		8	

Sudoku non résolu

4	8	7	*1*	*5*	*9*	*6*	*3*	2
1	*6*	*9*	*2*	7	*3*	*4*	5	8
3	*5*	*2*	4	*6*	8	1	7	*9*
7	*4*	3	*9*	2	5	*8*	6	*1*
5	9	*1*	6	8	*7*	*2*	*4*	3
8	*2*	6	3	*1*	4	5	*9*	7
9	*1*	5	*8*	*3*	6	7	*2*	*4*
6	3	8	7	*4*	*2*	*9*	1	5
2	7	*4*	*5*	9	1	*3*	8	*6*

Sudoku résolu

FIGURE 7 – **Grille facile** - Cases non fixes : 42 - Nombre moyen de chiffres possibles : 2.55

Grille facile 7	Nombre d'itérations	Temps de calcul
$N = 10$	4998	4.1943000000000001E-002 s
$N = 25$	1191	1.0731999999999998E-002 s
$N = 40$	1702	1.5347000000000000E-002 s
$N = 100$	8187	8.3151000000000003E-002 s
$N = 1000$	36091	0.30575999999999998 s

Pour ce sudoku, on remarque qu'il y a très rapidement convergence quelque soit le nombre N de grilles initiales. Il est cependant inutile de générer un grand nombre de grilles dans la population.

4.2.2.2 Grille moyenne

	8					3		5
6	9		3	4		1		
		1		6			4	9
	3				9		8	
	7	2			1		9	
			6	8		2		3
9	1				4			8
		4	8	1			3	
2		8			3	5		

Sudoku non résolu

4	8	*7*	*1*	*9*	*2*	3	*6*	5
6	9	*5*	3	4	*8*	1	*2*	*7*
3	*2*	1	*7*	6	*5*	*8*	4	9
5	3	*6*	*4*	*2*	9	*7*	8	*1*
8	7	2	*5*	*3*	1	*4*	9	*6*
1	*4*	*9*	6	8	*7*	2	*5*	3
9	1	*3*	*2*	*5*	4	*6*	*7*	8
7	*5*	4	8	1	*6*	*9*	3	*2*
2	*6*	8	*9*	*7*	3	5	*1*	*4*

Sudoku résolu

FIGURE 8 – **Grille moyenne** - Cases non fixes : 46 - Nombre moyen de chiffres possibles : 3.13

Grille moyenne 8	Nombre d'itérations	Temps de calcul
$N = 10$	81234	0.58185299999999995 s
$N = 25$	2660	2.1691999999999999E-002 s
$N = 40$	568090	3.7907160000000002 s
$N = 100$	77185	0.53762599999999994 s
$N = 1000$	46325	0.34854099999999999 s

Pour celui-ci, on remarque qu'il y a rapidement convergence quelque soit le nombre N de grilles initiales. Cependant, on remarque qu'à partir de $N = 25$, il y a une convergence nettement plus rapide.

4.2.2.3 Grille difficile

5		1	2					7
	2		4					
3					5	9	2	
4	3				7			1
					2		5	4
		5	6	4	8			
		8				3	6	
		6		9		1		
7			8	2				

Sudoku non résolu

5	*9*	1	2	*6*	*3*	*8*	*4*	7
6	2	*7*	4	*8*	*9*	*5*	*1*	*3*
3	*8*	*4*	*7*	*1*	5	9	2	*6*
4	3	*2*	*9*	*5*	7	*6*	*8*	1
8	*6*	*9*	*1*	*3*	2	*7*	5	4
1	*7*	5	6	4	8	*2*	*3*	*9*
9	*4*	8	*5*	*7*	*1*	3	6	*2*
2	*5*	6	*3*	9	*4*	1	*7*	*8*
7	*1*	*3*	8	2	*6*	*4*	*9*	*5*

Sudoku résolu

FIGURE 9 – **Grille difficile** - Cases non fixes : 51 - Nombre moyen de chiffres possibles : 3.52

Grille difficile 9	Nombre d'itérations	Temps de calcul
$N = 15$	425700	2.9116449999999996 s
$N = 20$	225904	1.5805190000000000 s
$N = 25$	87356	0.63403599999999993 s
$N = 200$	131725	1.0339170000000002 s
$N = 500$	4797724	35.857928000000001 s

Pour ce sudoku, on remarque que la convergence est parfois plus longue. Il est donc nécessaire de choisir minutieusement un bon nombre de grilles N au départ.

Ces 3 sudokus sont des plus classiques et on peut les trouver dans n'importe quel livre de sudoku. On peut même voir que la première est excessivement facile car à chaque case vide, on a le choix en moyenne entre seulement 2.55 chiffres.
Cependant, il existe des sudokus encore plus difficiles qu'on appelle "sudokus diaboliques" et même d'autres encore plus difficiles. Par exemple le sudoku "AI" figure 10 a longtemps été revendiquée comme le plus difficile par le finlandais Arto Inkala (d'où AI) ; mais depuis des sudokus plus difficiles ont été mis en lumière comme celui figure 11 qui a un SER plus élevé que le AI.

4.2.2.4 Grille AI

1					7		9	
	3			2				8
		9	6			5		
		5	3			9		
	1			8				2
6					4			
3							1	
	4							7
		7				3		

Sudoku non résolu

1	*6*	*2*	*8*	*5*	7	*4*	9	*3*
5	3	*4*	*1*	2	*9*	*6*	*7*	8
7	*8*	9	6	*4*	*3*	5	*2*	*1*
4	*7*	5	3	*1*	*2*	9	*8*	*6*
9	1	*3*	*5*	8	*6*	*7*	*4*	2
6	*2*	*8*	*7*	*9*	4	*1*	*3*	*5*
3	*5*	*6*	*4*	*7*	*8*	*2*	1	*9*
2	4	*1*	*9*	*3*	*5*	*8*	*6*	7
8	*9*	7	*2*	*6*	*1*	3	*5*	*4*

Sudoku résolu

FIGURE 10 – **Grille anciennement plus dure du monde, AI SER=10.5** - Cases non fixes : 58 - Nombre moyen de chiffres possibles : 4.46

Grille AI 10	Nombre d'itérations	Temps de calcul
$N = 40$	24229170	191.42896100000002 s
$N = 45$	8202729	66.567461999999992 s

Pour la grille AI, il est encore nécessaire de bien choisir la taille de la population N afin d'obtenir la solution exacte. Si l'on change seulement de 5 le nombre de grilles initiales, on a un rapport de 3 sur le temps de calcul.

4.2.2.5 Grille la plus difficile au monde

							3	9
					1			5
		3		5		8		
		8		9				6
	7				2			
1			4					
		9		8			5	
	2					6		
4			7					

Sudoku non résolu

7	*5*	*1*	*8*	*4*	*6*	*2*	3	9
8	*9*	*2*	*3*	*7*	1	*4*	*6*	5
6	*4*	3	*2*	5	*9*	8	*7*	*1*
2	*3*	8	*1*	9	*7*	*5*	*4*	6
9	7	*4*	*5*	*6*	2	*3*	*1*	*8*
1	*6*	*5*	4	*3*	*8*	*9*	*2*	*7*
3	*1*	9	*6*	8	*4*	*7*	5	*2*
5	2	*7*	*9*	*1*	*3*	6	*8*	*4*
4	*8*	*6*	7	*2*	*5*	*1*	*9*	*3*

Sudoku résolu

FIGURE 11 – **Grille la plus dure du monde, SER=11.9** - Cases non fixes : 60 - Nombre moyen de chiffres possibles : 4.71

Grille la plus dure du monde 11	Nombre d'itérations	Temps de calcul
$N = 35$	104713171	970.00205099999994 s
$N = 35$	79876986	761.57016699999997 s
$N = 40$	5079672	37.313479000000001 s
$N = 41$	30292802	482.82095900000002 s

Voici le sudoku étant actuellement considéré comme le plus complioqué au monde. Encore une fois il y a quasiment toujours convergence vers la solution exacte quelque soit la taille de la population N. Cependant, on peut voir qu'en changeant uniquement de 1 le nombre d'individus initiaux, on a un rapport 10 sur le temps de calcul.
Ainsi tous les sudokus peuvent être résolus relativement facilement avec la méthode génétique, et cela bien plus rapidement qu'en utilisant la méthode "naïve" qui consiste à essayer tous les cas. Mais est-ce pour autant la méthode la plus efficace pour ce problème ?

4.2.2.6 Optimisation de la fonction coût

Il est toujours possible d'optimiser le temps de le temps de calcul d'un programme. On cherche donc à diminuer le temps de calcul de notre fonction coût dans le programme . L'idée est de vérifier à chaque itération soit les lignes, soit les colonnes pour ressortir une erreur sur les lignes ou colonnes (à la différence de l'ancienne version qui vérifiait les 3 d'un coup). Puis si l'erreur est à 0, alors le programme vérifie sur les colonnes, lignes, blocs s'il n'y a pas d'autres erreurs.
Après avoir fait des tests avec cette fonction, on s'aperçoit rapidement que cette fonction n'est pas la meilleure car il n'y a convergence que pour les sudoku faciles ou moyens. De plus, les temps de calculs ne sont pas les meilleurs.

4.3 Méthode backtracking

L'utilisation de la méthode du backtracking pour la résolution de sudokus est très répandue, et elle est réputée comme étant très efficace. Donc on choisit de la comparer avec la méthode génétique.
Le backtracking, en français "retour sur trace" est un algorithme qui consiste à revenir légèrement en arrière sur des décisions prises afin de sortir d'un blocage. C'est-à-dire que l'on fonce jusqu'à se retrouver bloquer, puis on revient à l'étape précédente pour changer la dernière décision, et s'il n'y en a plus de non essayée on revient à l'étape de encore avant. Et cela ainsi de suite jusqu'à obtenir la bonne solution. On peut illustrer cela avec le schéma ci-dessous figure 12.

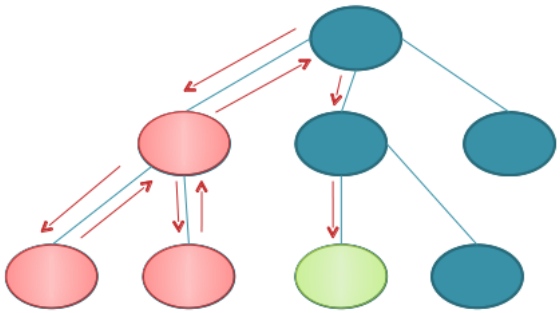

FIGURE 12 – Principe du backtracking

Pour l'appliquer à la résolution de sudoku, on a besoin dans un premier temps de créer des fonctions pour savoir s'il y a la possibilité de mettre un chiffre sur une certaine ligne, une certaine colonne ou un certain bloc (car s'il y est déjà on ne peut pas le mettre une deuxième fois). Ensuite, on a besoin de créer une fonction récursive qu'on appelle `estValide`. Cette fonction sait à chaque instant à quelle étape on est est, c'est-à-dire la combientième de case du sudoku est en train d'être remplie. Quand l'étape 81 est validée, le sudoku est complété.
Pour revenir à l'explication de cette méthode, à chaque étape un chiffre est rentré au hasard, et s'il n'était pas déjà dans la ligne, la colonne ou le bloc on passe à l'étape suivante. Mais s'il y était déjà, on revient à cette étape pour en mettre un autre. Et si tous les chiffres ont déjà été testés on revient à l'étape de encore avant pour mettre au hasard un autre chiffre qui n'a pas été testé.
Pour comprendre plus facilement, il est conseillé de s'aider du code en annexe 9.3.

4.4 Résultats backtracking

On résout donc les 5 sudokus précédents avec cette méthode afin d'obtenir leur temps de calcul et de pouvoir les comparer avec ceux de la méthode génétique. On obtient les résultats suivants :

	Temps de calcul
Grille facile	7.7e-05 s
Grille moyenne	8.7e-05 s
Grille difficile	0.000571 s
Grille AI	0.002518 s
Grille plus dure au monde	0.07401 s

On remarque que les temps de calculs sont nettement plus petits que ceux qui résultent de la méthode génétique. La grille la plus dure du monde est résolue en moins de 0.08s avec la méthode du backtracking, mais en plus de 37s avec la méthode génétique. Cependant, certains sudokus sont spécialement faits pour contrer la méthode du backtracking et demandent jusqu'à 30s de calcul. Mais cela reste inférieur aux 37s de la méthode génétique.

Ainsi, même si la méthode génétique fonctionne de manière efficace pour tous les problèmes d'optimisation, elle est parfois loin d'être la plus efficace. Comme dans le cas de la résolution de sudokus. Il faut donc être capable d'analyser les problèmes d'optimisation et de connaître les autres méthodes pour leur résolution, afin d'être capable de déterminer quand la méthode génétique n'est pas la plus adaptée.

5 Thermodynamique inversée : deux problèmes en un

Dans les applications, la population des solutions proposées ne devait s'adapter qu'à un seul problème. Mais en réalité, dans la sélection naturelle, il y a tout un tas de problème auxquels les espèces doivent s'adapter en même temps. Et d'ailleurs, il est très importants que les population s'adaptent en-même temps aux différents problèmes et pas un à un. Faisons une analogie avec la recherche du minimum d'une fonction 2D. Si on fixe un y et que l'on cherche le x pour lequel cette fonction est minimum à ce y fixé ; puis que l'on fixe à ce x la fonction et que l'on cherche son minimum en essayant tout les y, on ne tombe pas logiquement sur le minimum de la fonction pour tout (x,y). En effet, pour le trouver il faut le chercher en bougeant simultanément les deux coordonnées.
Il en va de même pour l'adaptation des espèces avec la sélection naturelle. Si un individu s'adapte très bien à un des problème, mais très mal à un autre, il ne pourra pas survivre. Il faut faut qu'il s'adapte le mieux possible à tous les problèmes réunis.
Pour un problème d'optimisation où il y a plusieurs contraintes, il faut donc procéder à la méthode génétique en ayant un opérateur qui prenne en compte les deux contraintes. La difficulté supplémentaire est donc de bien doser cet opérateur pour que la population n'ait pas tendance à mieux s'adapter à une des contraintes au détriment d'une autre.

5.1 Description du problème

Le problème a plusieurs contraintes que l'on choisit de résoudre est un problème de thermodynamique inversé. On a une pièce carrée dans un domaine $D = [0, 1]^2$, dont les murs sont maintenus à une certaine température, et suivant le problème que l'on choisit on a N radiateurs d'une taille donnée accrochés aux murs pour chauffer la pièce.
La première contrainte imposée est qu'à M endroits dans la pièce, on veut une certaine température. Et la seconde contrainte imposée est que l'on veut que les radiateurs soient aux températures les plus petites possibles.
Pour mesurer la température à un endroit donné, on utilise la méthode stochastique de la marche sur les cercles, sachant que, u étant la dissipation thermique, on a $\Delta u = 0$.

5.2 Méthode marche sur les cercles

Pour déterminer la température en un point, on utilise donc la méthode de la **marche sur les cercles** [4]. Le principe de la méthode est illustré figure 13.

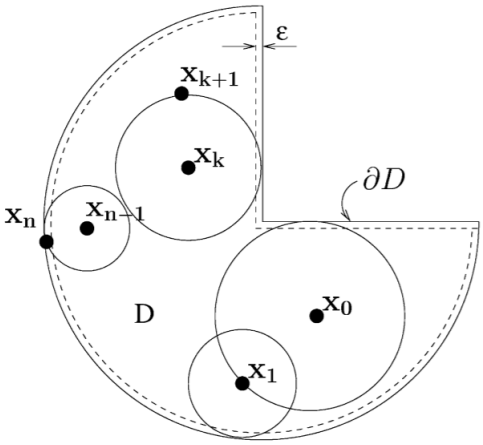

FIGURE 13 – Méthode de marche sur les cercles

On part d'un point $M_0 = (x, y)$. On calcule $r_{i+1} = d(M_i, \partial D)$ et on définit :

$$\begin{cases} x_{i+1} = x_i + r_{i+1}\cos(\theta_{i+1}) \\ y_{i+1} = y_i + r_{i+1}\sin(\theta_{i+1}) \end{cases} \tag{1}$$

où $\theta_{i+1} \sim U[0, 2\pi]$ tant que $r_i > \epsilon$. Une fois que $r_i < \epsilon$, on calcule la valeur de g au projeté de M_i sur le bord ∂D (voir figure 13). Il ne reste plus qu'à répéter ce procédé n fois puis faire la moyenne des valeurs de g au bord pour obtenir l'approximation de u. Le nombre moyen de cercles pour obtenir une précision ϵ est un $O(\ln(\epsilon))$.
Quant à la condition de sortie, on estime que dès que le marcheur est suffisamment proche d'un bord (comme à moins de 10^{-5}) c'est comme s'il était sorti. On ajoutera ensuite la température associé au mur de sortie à un compteur. Au bout de N itérations, on divisera le compteur par N pour obtenir la température en un point de la pièce.

5.3 Résolution par algorithme génétique

Pour la résolution de ce problème par algorithme génétique, il faut donc créer une population qui s'adapte à deux contraintes. Mais il faut surtout savoir que si on espère arriver à une solution exacte pour avoir les températures que l'on veut aux points donnés, il ne faut pas qu'il y ait plus d'équations que d'inconnues. Où il y a autant d'équations que de points où l'on veut une température précise ; et le double d'inconnues qu'il y a de radiateurs, car chaque radiateur a comme inconnues son emplacement et sa température.
Donc s'il y a deux fois plus de points que de radiateurs il n'y a qu'une solution (mais par symétrie, cette solution peut en fait se transformer en deux solutions), s'il y en a plus de deux fois plus il n'y a pas de solutions, et s'il y en a moins de deux fois plus il y a beaucoup de solutions. Et c'est dans ce dernier cas où la deuxième contrainte est importante car elle va contraindre à trouver la solution où l'addition des températures des radiateurs vaut le moins possible.

5.3.1 Genèse

La population pour cette résolution génétique est constituée de pièces. En effet, chaque individu est une pièce avec ses N radiateurs disposés à des endroits au hasard et avec la température de chacun décidée au hasard dans un intervalle paraissant adéquat.

5.3.2 Sélection

Pour son coût en calcul moindre, c'est la sélection par duel qui est adoptée dans cette résolution de problème. Mais pour cela, il faut donc être capable de quantifier l'erreur. C'est-à-dire à quel point cette disposition de pièce est près d'être la bonne.
L'erreur d'une pièce pourrait simplement être l'addition des différences entre les températures que l'on a aux points et celles que l'on désire. Mais l'erreur doit également prendre en compte les températures des radiateurs pour qu'elles soient les plus petites possibles. Pour cela, on rajoute à l'erreur précédente un centième de la somme des températures des radiateurs. On modélise cela par la formule suivante :

$$J = \varepsilon \sum_{i=1}^{n_{rad}} Trad_i + \sum_{i=1}^{n_{zone}} \left| \int_{Z_i} (u(x) - T_i)dx \right|$$

avec $\varepsilon = 0.01$. Pour notre cas, cela se traduit plus simplement par :

$$J = \varepsilon Trad + \left| \int_{Z_1} (u(x) - T_1)dx \right| + \left| \int_{Z_2} (u(x) - T_2)dx \right|$$

avec $T_1 = T_2 = 20°C$. Cela permet à l'algorithme génétique de converger vers la pièce avec les radiateurs les moins chauds, sans pour autant vraiment s'éloigner des températures désirées en certains points.

5.3.3 Opérateur

L'opérateur choisi pour cette méthode génétique est encore une fois le clonage suivi de la mutation. Donc la pièce qui gagne le duel reste la même tandis que celle qui le perd devient une presque-copie de l'autre.
La mutation du perdant après le clonage est que ses positions de radiateurs changent un peu, et leurs températures également. Mais dans ce problème, il faut également que la taille de la fourchette dans laquelle le radiateur peut bouger ou changer sa température diminue à chaque itération. Sinon, quand il y a un duel entre deux pièces à faible erreur, l'individu perdant ne pourrait en fait presque que s'éloigner de la bonne solution. Donc la fourchette est en fait divisée par $n^{\frac{1}{2}}$, n étant le numéro de l'itération. C'est la puissance de n la plus adaptée pour cela car si la puissance n'était pas inférieure à 1 la fourchette diminuerait trop vite. Cette puissance est trouvée par dichotomie.

5.4 Résultats

En faisant tourner le programme génétique pour résoudre ce problème, on se rend compte qu'il est très très coûteux en calculs. Pour pallier cela, on se focalise donc sur des problèmes où l'on n'a seulement deux points où l'on désire une certaine température, et seulement un radiateur. Mais, même comme cela, le programme tourne à chaque fois plus de 5 minutes avant d'obtenir une erreur inférieure à 1 (l'addition des deux erreurs aux deux points vaut moins de 1°C).
Pour vérifier si les résultats obtenus sont logiques, on place les deux points de manière symétrique (par rapport à la verticale centrale), et on demande la même température en ces deux points, soit 19°C. Donc logiquement, le radiateur devrait être placé pile entre les deux points sur le mur le plus proche, et on peut trouver par dichotomie avec la méthode de la marche sur les cercles que sa température doit être de 25.5°C.
Et c'est bien ce que l'on obtient avec le programme génétique, avec une population initiale de 100 individus (par dichotomie on obtient que c'est le plus optimum). Le programme met quand même plus de cinq minutes avant de renvoyer un individu avec une erreur de 0.2 figure 14, qui a son radiateur placé à la coordonnée 0.51 sur le mur le plus proche (les coordonnées de chaque mur allant de 0 à 1), et dont la température est 25.55°C.

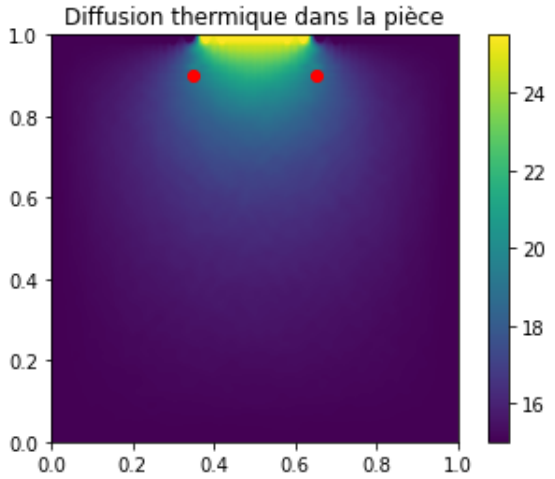

FIGURE 14 – Diffusion dans la pièce pour les deux zones choisit

Ainsi, pour un tel problème de thermodynamique inversé où il y a plusieurs contraintes, la méthode génétique fonctionne toujours. On se rend néanmoins compte qu'elle demande un coût très important en calcul.
Dans cette application, il n'y a que deux contraintes. Mais pour un problème avec des centaines de contraintes, une méthode génétique serait bien trop coûteuse en calcul (on comprend donc pourquoi les évolutions d'espèces se font en plusieurs millions d'années). Cependant, ce problème est quand même particulièrement coûteux en calcul, car à chaque itération il faut faire autant de méthode de marche sur les cercle que la moitié de la population.Donc avec un problème moins coûteux en calcul, il faudrait des dizaines de contraintes pour arriver à un tel temps de calcul.

6 Coloration de graphe : amélioration de la méthode génétique

Ce qui rend la sélection naturelle si efficace, et surtout si sûr pour arriver à la meilleure solution, c'est que les caractéristiques des individus changent quelque part au hasard. Ainsi, toutes les possibilités sont exploitées, et seules les meilleures sont gardées. Mais que se passerait-il si pour chaque caractéristique, la mutation avait lieu à l'endroit où cela pose problème ? Ne serait-ce pas plus efficace ? Pour cela, il faut bien sûr savoir où la caractéristique est inadaptée, et c'est la raison pour laquelle la sélection naturelle ne peut pas fonctionner comme cela. Mais dans certains problèmes, il est possible de savoir précisément où est l'erreur qui rend l'individu inadapté.

6.1 Description du problème

Le principe est qu'il y a des zones, reliées entre elles, qui doivent être coloriées d'une couleur. Mais la contrainte est que deux zones reliées entre elles ne peuvent pas être de la même couleur, et qu'il faut utiliser le moins de couleurs possibles.
Par exemple, cette coloration est la plus optimale qui soit pour ce schéma figure 15 :

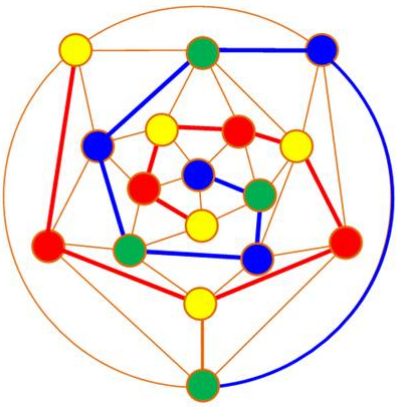

FIGURE 15 – Description du problème : coloration de graphe

Ce type de problème peut s'appliquer à plusieurs choses, comme la coloration d'une carte où il ne faut pas que les zones voisines soient de la même couleur, ou encore, en le modifiant un peu, à la répartition d'espèces animales sur des territoires, quand certaines ne doivent pas être en contact.

6.2 Méthode génétique

Comme cas test, on utilise les schémas ci-dessous qui ne sont pas trop complexes, mais le sont quand même assez pour être intéressants. Bien sûr, ils vont être résolus très rapidement par la méthode génétique, mais ils servent des cas de comparaison.

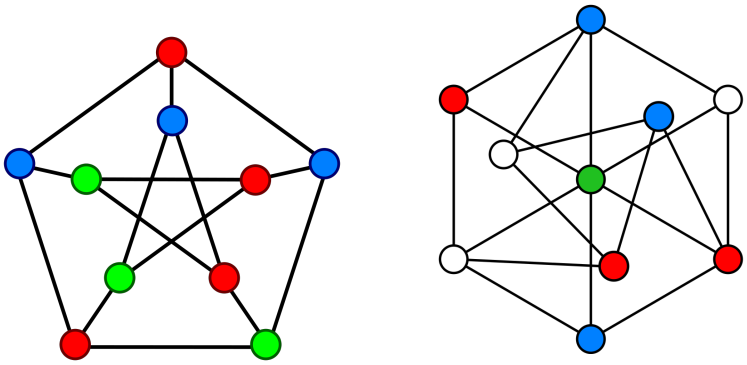

FIGURE 16 – Cas 1 et 2 de coloration de graphe

Par dichotomie, on obtient que les populations les plus optimales pour ces problèmes sont de quatre individus pour le premier, et de six individus pour le second. Où chaque individu est un schéma avec des couleurs mises au hasard dans chaque zone.
Le mode de sélection choisi est celui du duel, où le gagnant est celui avec la plus petite erreur. Sachant que l'erreur est telle que, pour chaque zone qui a un ou plusieurs voisines de la même couleur, elle augmente de 1.
Le gagnant reste donc le même, tandis que le perdant devient comme le gagnant, sauf qu'il a deux zones où la couleur change.
Ces paramètres de la méthode génétique donnent le meilleur résultat, car les deux problèmes sont résolus en respectivement treize et vingt-trois itérations.

6.3 Méthode pseudo-génétique

Même si la méthode génétique est très efficace sur ce type de problème, on se doute qu'elle perd du temps en modifiant parfois la couleur de zones qui étaient déjà parfaites. Il serait donc possible d'obtenir un programme plus optimal en ciblant les zones à erreur.
L'idée qui en découle alors est la suivante : il suffirait de se contenter d'une population à un seul individu, de relever quelles zones sont fausses (quand elles ont un voisin de la même couleur), puis de changer la couleur de cette zone. De plus, pour que la méthode soit plus optimale, il n'y a en fait qu'une chance sur deux pour que la couleur d'une zone fausse change. Cela est dû au fait qu'une seule liaisons fausse entre deux zones entraîne deux erreurs.
Avec cette méthode, le premier cas est résolu en seulement six itérations, et le second cas en seulement neuf itérations. De plus, chaque itération de cette méthode est bien moins coûteuse en calcul qu'une itération de la méthode génétique.
Même si cette méthode n'a une population que de un individu, elle s'apparente à la méthode génétique car l'individu concerné transmet à son descendant ses bons gênes, et seuls les mauvais gênes vont muter.
On voit donc bien qu'il existe encore des variantes à la méthode génétique qui reste assez méconnues, mais qui pourtant sont très efficaces pour certains problèmes.

7 Pour aller plus loin

Si on ne prend pas en compte le temps de calcul, tout problème d'optimisation peut en fait être résolu par la méthode génétique. Pour cela, il suffit de trois choses. Il faut obtenir par dichotomie la bonne population initiale lors de la genèse pour avoir le coût en calcul le plus petit possible. Il faut également choisir le bon opérateur, pour que la population s'adapte bien au problème. Mais même avec l'opérateur le plus optimal, la population converge vers la solution au problème. Et surtout, il faut choisir la bonne manière de quantifier l'erreur, par rapport au problème, de chaque individu de la population, afin d'avoir une bonne sélection. Sans cela, la méthode génétique ne peut pas fonctionner.
Quand la méthode génétique ne fonctionne pas sur un problème d'optimisation, c'est qu'elle est mal paramétrée. Et, dans la plupart des cas, c'est donc le calcul de l'erreur qui est mal paramétré. En effet, l'erreur est parfois compliquée à calculer ou même à cerner.
Dans les cas étudiés précédemment, les erreurs étaient relativement simples à cerner puis à calculer, mais il existe des cas où elles sont bien plus compliquées.

7.1 Profil NACA : erreur compliquée à calculer

Les profils NACA sont des profils aérodynamiques pour les ailes d'avions développés par le Comité consultatif national pour l'aéronautique (NACA, États-Unis). Il s'agit de la série de profils la plus connue et utilisée dans la construction aéronautique.

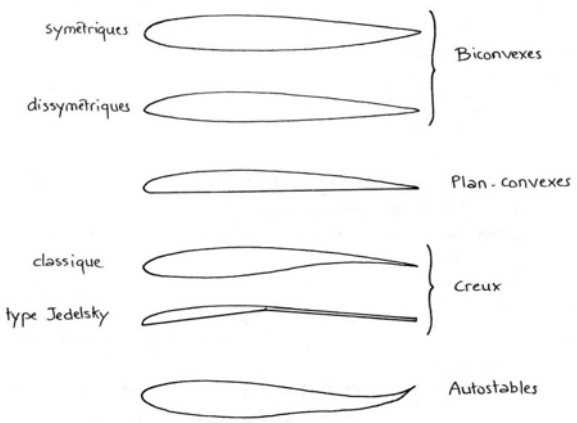

Figure 17 – Profils NACA

Avec la méthode génétique, il est possible de déterminer le profil d'aile le plus aérodynamique. La population est donc constituée d'individus qui sont des profils d'ailes créés au hasard, et en utilisant une sélection de duel, puis un opérateur de clonage/mutation fait converger la population vers le profil d'aile le plus aérodynamique.
Mais pour faire une sélection, il faut une moyenne de quantifier "l'erreur". Cette erreur choisie est en fait la force de traînée de ce profil d'aile, et plus elle est petite, mieux est le profil. Cependant, calculer cette force de traînée est extrêmement compliqué. Donc on se retrouve avec un problème qui peut se résoudre par méthode génétique avec un raisonnement assez, mais qui finalement peut se révéler très lourd en calcul.

7.2 Arrangement des feuilles d'une plante : erreur/contrainte compliquée à trouver

Quand on regarde la manière dont les feuilles poussent sur la tige d'une plante, on comprend directement que ce n'est pas vraiment dû au hasard. En effet, pour de nombreuse plante, il y

a le même angle entre chaque feuille qui se suivent. Et cet angle est le nombre d'or, qui vaut $\frac{1+\sqrt{5}}{2}$.

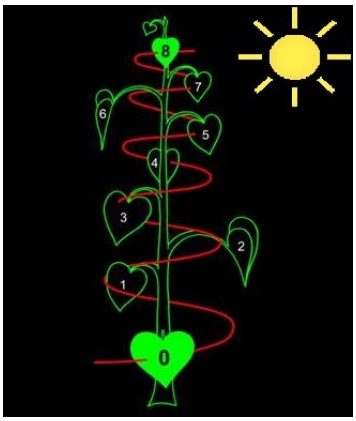

FIGURE 18 – Arrangement des feuilles d'une plante

On comprend bien qu'il y a une raison à cela. Si la sélection naturelle a rendu ces espèces comme cela, c'est que c'est la solution optimale à une contrainte. Donc dans ce problème d'arrangement des feuilles d'une plante, le plus compliqué est de trouver à quelle contrainte répond le problème, afin de pouvoir calculer une erreur pour la sélection.
Et cette contrainte est en fait une double contrainte. Il faut que vue du dessus les feuilles se chevauchent le moins possible. Et il faut également que les feuilles soient les plus éloignées possibles, afin que des bourgeons puissent se développer plus facilement.
Et, encore une fois, c'est une erreur extrêmement compliquée à calculer. Donc la méthode génétique est encore très adaptée, mais vraiment compliquée à mettre en place.

8 Conclusion

Ainsi, pour chaque problème d'optimisation, la méthode génétique est possible, et elle reste toujours relativement efficace. Dans certains cas comme par exemple le sudoku, cette méthode est loin d'être la plus efficace alors que dans d'autres comme le problème du voyageur de commerce, cela reste la plus optimisée. Il faut savoir reconnaître quand cette méthode s'impose, ou quand il faut en utiliser une autre. De plus, et très logiquement, plus le problème traité a des contraintes, et plus la méthode génétique va avoir besoin d'itérations pour obtenir une bonne solution. Sachant que chaque itération va demander plus de calculs. Mais elle reste plus simple (parfois plus efficace) à mettre en place que les méthodes de mise en équation / résolution d'équations physique par méthodes tel les éléments finis. Cependant, il faut également savoir que pour certains problèmes, même si ce n'est pas la méthode génétique la plus efficace, cela peut en être une variante plus simple à mettre en place.

Références

[1] Maire Sylvain. *Cours sur les méthodes Monte Carlo*, 2020.

[2] Wikipédia, Problème du voyageur de commerce. https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_voyageur_de_commerce

[3] Maire Sylvain. *Projet Sudoku automatique Seatech*, 2021.

[4] Chi-Ok Hwang, Michael Mascagni, James A. Given. *A Feynman–Kac path-integral implementation for Poisson's equation using an h-conditioned Green's function*, 2003. http://websrv.cs.fsu.edu/~mascagni/papers/RIJP2003_1.pdf

[5] Wikipédia, Coloration de graphe. https://fr.wikipedia.org/wiki/Coloration_de_graphe

9 Annexes

9.1 Voyageur de commerce sur la France en python

```
from matplotlib.pylab import * # importe aussi numpy sans alias
from scipy.sparse.linalg import spsolve
from scipy.sparse import spdiags
import imageio
import shutil
import matplotlib.cm as cm
from PIL import Image
import random as rd
from math import factorial
import time

def circuit(ville):
    return [ville[ind][0] for ind in range(nb_villes)]+[ville[0][0]],[ville[ind][1] for ind

def plot_circuit(ville):
    X,Y=circuit(ville)

    plot(X,Y,'white',linewidth=10)
    for i in range(len(X)-1):
        scatter(X[i],Y[i],marker="o",s=500,c="red")
    title("Circuit le plus court pour "+str(len(X)-1)+" villes",fontsize=40)
    xlabel("x")
    ylabel("y")
    return

def dist(ville):
    X,Y=circuit(ville)
    nb_villes=len(X)-1
    return sum([((X[i+1]-X[i])**2+(Y[i+1]-Y[i])**2)**0.5 for i in range(nb_villes)])

def perm2(ville):
    n_pop=len(ville)
    n1=rd.randint(0,n_pop-1)
    n2=rd.randint(0,n_pop-1)
    ville[n1],ville[n2]=ville[n2],ville[n1]
    return ville

def perm3(ville):
    n_pop=len(ville)
    n1=rd.randint(0,n_pop-1)
    n2=rd.randint(0,n_pop-1)
    n3=rd.randint(0,n_pop-1)
    ville[n1],ville[n2],ville[n3]=ville[n3],ville[n1],ville[n2]
    return ville

###############
# Paramètres #
###############
```

```
#Spatial
coef=1 #Coefficient de division de taille d'image: initialement de taille 1024 x 1024
n=int(Image.open('france2.jpg').size[0]/coef) #Longueur du côté du maillage.


x_seatech=int(810/coef)
y_seatech=int(950/coef)
v1=[x_seatech,y_seatech]

x_paris=int(556/coef)
y_paris=int(267/coef)
v2=[x_paris,y_paris]

x_lyon=int(720/coef)
y_lyon=int(635/coef)
v3=[x_lyon,y_lyon]

x_bordeaux=int(276/coef)
y_bordeaux=int(695/coef)
v4=[x_bordeaux,y_bordeaux]

x_toulouse=int(419/coef)
y_toulouse=int(856/coef)
v5=[x_toulouse,y_toulouse]

x_caen=int(400/coef)
y_caen=int(210/coef)
v6=[x_caen,y_caen]

x_marseille=int(762/coef)
y_marseille=int(933/coef)
v7=[x_marseille,y_marseille]

x_nantes=int(235/coef)
y_nantes=int(415/coef)
v8=[x_nantes,y_nantes]

x_lilles=int(630/coef)
y_lilles=int(67/coef)
v9=[x_lilles,y_lilles]

x_nice=int(890/coef)
y_nice=int(890/coef)
v10=[x_nice,y_nice]

x_strasbourg=int(954/coef)
y_strasbourg=int(330/coef)
v11=[x_strasbourg,y_strasbourg]
```

```
x_montpelier=int(640/coef)
y_montpelier=int(816/coef)
v12=[x_montpelier,y_montpelier]

x_tours=int(410/coef)
y_tours=int(415/coef)
v13=[x_tours,y_tours]

x_brest=int(40/coef)
y_brest=int(270/coef)
v14=[x_brest,y_brest]

x_rennes=int(240/coef)
y_rennes=int(310/coef)
v15=[x_rennes,y_rennes]

villes=[v1,v2,v3,v4,v5,v6,v7,v8,v9,v10,v11,v12,v13,v14,v15]
nb_villes=len(villes)
size=15

rd.seed(1)
nb_circuits_possibles=1/2*factorial(nb_villes-1)
print("Nombre de circuits possibles="+str(nb_circuits_possibles)+" | best candidat="+str(nb_
n_pop=1000
Niter=100000
pop_villes=[villes.copy() for i in range(n_pop)]

for i in range(n_pop):
    rd.shuffle(pop_villes[i])


# Colormap rouge
theCM = cm.get_cmap("Reds")
theCM._init()
alphas = abs(linspace(0, 1, theCM.N))
theCM._lut[:-3,-1] = alphas



#####################################################
# Importation d'image, redimenssion et inversion #
#####################################################

fig = plt.figure(figsize=(size,size))
img = Image.open('france2.jpg')
img = img.resize((int(img.size[0]/coef),int(img.size[1]/coef)))
img = 255 - np.asarray(img)
imgplot = plt.imshow(img)
imgplot.set_interpolation('nearest')

#Sauvegarde de la première image
```

```
i=0
# representation graphique de la solution
imgplot = plt.imshow(img)
imgplot.set_interpolation("nearest")


dmin=dist(pop_villes[0])
for i in range(Niter):
    n1=rd.randint(0,n_pop-1)
    n2=rd.randint(0,n_pop-1)
    ville1=pop_villes[n1]
    ville2=pop_villes[n2]
    d1=dist(ville1)
    d2=dist(ville2)
    if d1<d2:
        ville2=ville1.copy()
        pop_villes[n2]=perm2(ville2)
        if d1<dmin:
            dmin=d1
            ville_min=ville1.copy()
    else:
        ville1=ville2.copy()
        pop_villes[n1]=perm2(ville1)
        if d2<dmin:
            dmin=d2
            ville_min=ville2.copy()

plot_circuit(ville_min)
print(dmin)
fig.savefig("Nb_iter-"+str(Niter)+".png")
```

9.2 Résolution sudoku par AG en fortran

9.2.1 Programme principal

```
program sudo
implicit none

integer                     :: N,i,ii,j,jj,k,kk,kposs,c,test,bool,n_non_remp,nb_double,n_cou
real ( kind = 8)            :: ix,na,nmax,u1,u2,u,t1,t2
integer, dimension (:), allocatable :: L_comp,L_poss,size_poss
integer, dimension (:,:), allocatable :: zones,grille_ini,ind_non_rempli,ind_zone,grille,gri
integer, dimension (:,:,:), allocatable :: all_grilles

allocate(zones(9,9),grille_ini(9,9),ind_non_rempli(81,2),ind_zone(9,18),grille1(9,9),grille(
allocate(L_comp(9),L_poss(9))

! Initialisation des nombres d'aléatoires
ix=51477.d0
na=16807.d0
nmax=2147483647.d0 ! nombre premier tres grand
```

```
!Les zones du sudoku

zones(1,:)=(/1,1,1,2,2,2,3,3,3/)
zones(2,:)=(/1,1,1,2,2,2,3,3,3/)
zones(3,:)=(/1,1,1,2,2,2,3,3,3/)
zones(4,:)=(/4,4,4,5,5,5,6,6,6/)
zones(5,:)=(/4,4,4,5,5,5,6,6,6/)
zones(6,:)=(/4,4,4,5,5,5,6,6,6/)
zones(7,:)=(/7,7,7,8,8,8,9,9,9/)
zones(8,:)=(/7,7,7,8,8,8,9,9,9/)
zones(9,:)=(/7,7,7,8,8,8,9,9,9/)


! Initialisation des indices des zones:
! par exemple pour obtenir les indices de la zone n, on
! parcours k de 1 à 9 pour récuperer les indices i=ind_zone(n,k) et j=ind_zone(n,k+9)

DO k=1,9
   c=0
   DO i=1,9
      DO j=1,9
         IF (zones(i,j)==k) THEN
            c=c+1
            ind_zone(k,c)=i
            ind_zone(k,c+9)=j
         ENDIF
      ENDDO
   ENDDO
ENDDO



!======================!
!= Une grille initiale =!
!======================!

!Grille livre 1ere FACILE - Fonctionne
! Résolution en        4998 itérations et   4.4006999999999998E-002 s avec          10 grill


grille_ini(1,:)=(/4,8,7,0,0,0,0,0,2/)
grille_ini(2,:)=(/1,0,0,0,7,0,0,5,8/)
grille_ini(3,:)=(/3,0,0,4,0,8,1,7,0/)

grille_ini(4,:)=(/0,0,3,0,2,5,0,6,0/)
grille_ini(5,:)=(/0,9,0,6,8,0,0,0,3/)
grille_ini(6,:)=(/0,0,6,3,0,4,5,0,7/)

grille_ini(7,:)=(/0,0,5,0,0,6,7,0,0/)
grille_ini(8,:)=(/0,3,8,7,0,0,0,1,5/)
```

```
grille_ini(9,:)=(/2,7,0,0,9,1,0,8,0/)



CALL cell_missing(n_non_remp) !Nombre de cellules manquantes et affectation au tableau des i
! Avec ceci on peut parcourir k de 1 à n_non_remp pour avoir les indices:
! i=ind_non_rempli(k,1) et j=ind_non_rempli(k,2) des cases non remplies


allocate(poss(n_non_remp,9)) !Liste des possibles à chaque cases non remplies
allocate(size_poss(n_non_remp)) !Taille de ces listes (nombre de possible par cases)

deallocate(L_poss)
DO kposs=1,n_non_remp
   allocate(L_poss(9))
   L_poss(:)=0
   CALL possible(ind_non_rempli(kposs,1),ind_non_rempli(kposs,2),L_poss)
   poss(kposs,:)=L_poss
   c=1
   DO i=1,9
      IF (L_poss(i)==0) THEN
         c=i-1
         EXIT
       ENDIF
   ENDDO
   size_poss(kposs)=c
   deallocate(L_poss)
ENDDO
allocate(L_poss(9))

! Ecrit les possibilités de chiffres pour chaque cellules
OPEN ( UNIT =48 , FILE = 'poss.dat')
DO i =1,n_non_remp
        WRITE (48 ,*) (poss (i , j ) , j =1 , size_poss(i))
ENDDO
CLOSE (48)


!======================!
!= Corps du programme =!
!======================!

n_cout=10
c=0
N=25   !Nombre de grilles initiales
CALL CPU_TIME(t1) !Timer 1
allocate(all_grilles(N,9,9)) !Tableau 3 dimensions des grilles

DO k=1,N !Generation de N grilles
   CALL create_grille(grille)
   all_grilles(k,:,:)=grille
```

```
ENDDO


DO WHILE (1==1)
   c=c+1 !Compteur de nombre d'itérations
   CALL rd_int(i1,1,N) !Prends un indice pour une premiere grille aléatoire
   CALL rd_int(i2,1,N) !Prends un indice pour une seconde grille aléatoire
   grille1=all_grilles(i1,:,:)
   grille2=all_grilles(i2,:,:)
   CALL cout(grille1,cout1) !Calcul des erreurs de la grille1
   CALL cout(grille2,cout2) !Calcul des erreurs de la grilel2
   n_cout=minval((/cout1,cout2/)) !Cout minimum

   ! Condition d'arret
   IF (n_cout<=0) THEN
      IF (n_cout==cout1) THEN
         ind_f=i1
      ELSE
         ind_f=i2
      ENDIF
      EXIT
   ELSE

   if (cout2>=cout1) THEN !Change la grille au plus fort cout
      CALL change(grille1)
      all_grilles(i2,:,:)=grille1 !Réaffecte la grille changée
      IF (n_cout>cout1) THEN
         n_cout=cout1 !Change le coût
      ENDIF
   ELSE
      CALL change(grille2) !Change la grille au plus fort cout
      all_grilles(i1,:,:)=grille2 !Réaffecte la grille changée
      IF (n_cout>cout2) THEN
         n_cout=cout2 !Change le coût
      ENDIF
   ENDIF
   ENDIF
ENDDO
CALL cpu_time(t2) !Temps après le calcul

grille=all_grilles(ind_f,:,:)
print*,"Grille initiale:"
call aff(grille_ini)
print*,"Grille résolue:"
call aff(grille)
call cout(grille,n_cout)
print*,"Résolution en",c,"itérations et",t2-t1,"s avec",N,"grilles de départ et"&
&,n_non_remp," cases non remplies. Vérification: nb erreurs",n_cout

! ====================================
! =========== SUBROUTINES ==============
```

```
! ====================================

CONTAINS
SUBROUTINE rd(u)  !Aléatoire entre 0 et 1
   real ( kind = 8)            :: u
   ix=abs(ix*na)
   ix=mod(ix,nmax)
   u=dble(ix)/dble(nmax)
   RETURN
END SUBROUTINE rd

SUBROUTINE rd_int(u,a,b) !Aléatoire entier entre a et b inclus
   integer                     :: u,a,b
   real ( kind = 8)            :: u1
   ix=abs(ix*na)
   ix=mod(ix,nmax)
   u1=dble(ix)/dble(nmax)
   u=int(u1*(b-a+1)+a)
   RETURN
END SUBROUTINE rd_int


SUBROUTINE nb_doublons(L,nb)  !Nombre de doublons dans une liste L
   integer, dimension (:), allocatable :: L
   integer                     :: nb
   nb=0
   DO i=1,9
      DO j=i+1,9
         IF (L(i)==L(j)) THEN
            nb=nb+1
         ENDIF
      ENDDO
   ENDDO
   RETURN
END SUBROUTINE nb_doublons

SUBROUTINE cout(grille,n_cout) !Fonction coût qui compte le nombre d'erreurs dans la grille
   integer, dimension (:,:), allocatable :: grille
   integer, dimension (:), allocatable :: L_double
   integer                     :: n_cout,nb_double,k,kk
   n_cout=0
   DO k=1,9
      L_double=grille(k,:)
      CALL nb_doublons(L_double,nb_double) !Cout sur les lignes
      n_cout=n_cout+nb_double

      L_double=grille(:,k)
      CALL nb_doublons(L_double,nb_double) !Cout sur les colonnes
      n_cout=n_cout+nb_double

      DO kk=1,9
```

```
         L_double(kk)=grille(ind_zone(k,kk),ind_zone(k,kk+9))
      ENDDO
      CALL nb_doublons(L_double,nb_double) !Cout sur les zones
      n_cout=n_cout+nb_double
   ENDDO
   RETURN
END SUBROUTINE cout

SUBROUTINE cout_partial(bool,grille,n_cout) !Fonction coût qui compte le nombre d'erreurs do
   integer, dimension (:,:), allocatable :: grille
   integer, dimension (:), allocatable :: L_double
   integer                      :: n_cout,nb_double,k,kk,bool
   n_cout=0
   IF (bool==1) THEN
          DO k=1,9
                 L_double=grille(k,:)
                 CALL nb_doublons(L_double,nb_double) !Cout sur les lignes
                 n_cout=n_cout+nb_double
          ENDDO
   ENDIF

   IF (bool==2) THEN
          DO k=1,9
                 L_double=grille(:,k)
                 CALL nb_doublons(L_double,nb_double) !Cout sur les colonnes
                 n_cout=n_cout+nb_double
          ENDDO
   ENDIF

   IF (bool==3) THEN
          DO k=1,9
                 DO kk=1,9
                        L_double(kk)=grille(ind_zone(k,kk),ind_zone(k,kk+9))
                 ENDDO
                 CALL nb_doublons(L_double,nb_double) !Cout sur les zones
                 n_cout=n_cout+nb_double
          ENDDO
   ENDIF
   RETURN
END SUBROUTINE cout_partial


SUBROUTINE change(grille) !Effectue les changements sur la grille
   integer, dimension (:,:), allocatable :: grille
   integer                      :: u1,v1,rg
   CALL rd_int(rg,2,3) !2 ou 3 changements
   DO k=1,rg
      CALL rd_int(u1,1,n_non_remp) !Aléatoire sur le nombre de cellules non remplies
          CALL rd_int(v1,1,size_poss(u1)) !Aléatoire sur le nombre de possibles sur cette ce
      grille(ind_non_rempli(u1,1),ind_non_rempli(u1,2))=poss(u1,v1) !Changement
   ENDDO
```

```
   RETURN
END SUBROUTINE change



SUBROUTINE aff(M) !Affichange de la grille
   integer, dimension (:,:), allocatable :: M
   DO i=1,9
      print*,M(i,1),M(i,2),M(i,3),M(i,4),M(i,5),M(i,6),M(i,7),M(i,8),M(i,9)
   ENDDO
   RETURN
END SUBROUTINE aff

SUBROUTINE cell_missing(n) !Nombre de cellules manquantes sur la grille initiale et implémen
   integer                      :: n
   n=0
   DO i=1,9
      DO j=1,9
         IF (grille_ini(i,j)==0) THEN
            n=n+1
            ind_non_rempli(n,1)=i
            ind_non_rempli(n,2)=j
         ENDIF
      ENDDO
   ENDDO
   RETURN
END SUBROUTINE cell_missing

SUBROUTINE in_T(T,a,bool) !Test d'appartenance d'un scalaire à une liste
   integer                      :: bool,a
   integer, dimension (:), allocatable :: T
   bool=0
   DO i=1,size(T)
      IF (T(i)==a) THEN
         bool=1
      ENDIF
   ENDDO
   RETURN
END SUBROUTINE in_T

SUBROUTINE complementaire(L_ini,L_comp) !Renvoie la liste complémentaire de chiffres (à 9).
   integer                              :: bool,a,c,k
   integer, dimension (:), allocatable :: L_ini,L_comp
   c=0
   DO k=1,9
      CALL in_T(L_ini,k,bool)
      IF (bool==0) THEN
         c=c+1
         L_comp(c)=k
      ENDIF
   ENDDO
```

```
   RETURN
END SUBROUTINE complementaire


SUBROUTINE create_grille(grille) !Création d'une grille
   integer                                 :: N,i,j,k
   real ( kind = 8)                 :: u
   integer, dimension (:), allocatable :: L_possi
   integer, dimension (:,:), allocatable :: grille
   grille=grille_ini
   DO k=1,n_non_remp
      i=ind_non_rempli(k,1)
      j=ind_non_rempli(k,2)
      CALL rd_int(n,1,size_poss(k)) !Entier entre 1 et le nombre de possibles pour la k-ieme
      grille(i,j)=poss(k,n)
   ENDDO
   RETURN
END SUBROUTINE create_grille


SUBROUTINE possible(i,j,L_poss) ! Donne les chiffres possibles pour la case i,j
   integer                                 :: c,i,j,k,kk
   integer, dimension (:), allocatable :: Tab_digits,L_poss
   allocate(Tab_digits(81))
   Tab_digits(:)=0
   c=0
   DO k=1,9
      IF (grille_ini(i,k)==0) THEN
      ELSE
         c=c+1
         Tab_digits(c)=grille_ini(i,k) !Test sur les lignes
      ENDIF

      IF (grille_ini(k,j)==0) THEN
      ELSE
         c=c+1
         Tab_digits(c)=grille_ini(k,j) !Test sur les colonnes
      ENDIF

      DO kk=1,9
         IF (grille_ini(ind_zone(k,kk),ind_zone(k,kk+9))==0) THEN
         ELSE
            c=c+1
            Tab_digits(c)=grille_ini(k,j) !Test sur les zones
         ENDIF
      ENDDO

   ENDDO
   CALL complementaire(Tab_digits,L_poss)
   deallocate(Tab_digits)
   RETURN
```

```
END SUBROUTINE possible


! ========== FIN DU PROGRAMME =============
end program  sudo
```

9.2.2 Optimisation fonction coût

```
SUBROUTINE cout(grille,n_cout)
   integer, dimension (:,:), allocatable :: grille
   integer, dimension (:), allocatable :: L_double
   integer                       :: n_cout,nb_double,rgbis
   real ( kind = 8)              :: rg

   n_cout=0
   CALL rd(rg)
   rgbis=int(rg*2) !0 ou 1

   IF (rgbis==0) THEN
      DO k=1,9
         L_double=grille(k,:)
         CALL nb_doublons(L_double,nb_double) !Cout sur les lignes
         n_cout=n_cout+nb_double
      ENDDO
   ENDIF

   IF (rgbis==1) THEN
      DO k=1,9
         L_double=grille(:,k)
         CALL nb_doublons(L_double,nb_double) !Cout sur les colonnes
         n_cout=n_cout+nb_double
      ENDDO
   ENDIF

   IF (n_cout==0) THEN

      DO k=1,9
         L_double=grille(k,:)
         CALL nb_doublons(L_double,nb_double) !Cout sur les lignes
         n_cout=n_cout+nb_double

         L_double=grille(:,k)
         CALL nb_doublons(L_double,nb_double) !Cout sur les colonnes
         n_cout=n_cout+nb_double

         DO kk=1,9
            L_double(kk)=grille(ind_zone(k,kk),ind_zone(k,kk+9))
         ENDDO
         CALL nb_doublons(L_double,nb_double) !Cout sur les zones
         n_cout=n_cout+nb_double
      ENDDO
```

```
   ENDIF

   RETURN
END SUBROUTINE cout
```

9.3 Résolution sudoku backtracking en C

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#include <iostream>
// Fonction d'affichage
void affichage (int grille[9][9])
{
    for (int i=0; i<9; i++)
    {
        for (int j=0; j<9; j++)
        {
            printf( ((j+1)%3) ? "%d " : "%d|", grille[i][j]);
        }
        putchar('\n');
        if (!((i+1)%3))
            puts("------------------");
    }
    puts("\n\n");
}

bool absentSurLigne (int k, int grille[9][9], int i)
{
    for (int j=0; j < 9; j++)
        if (grille[i][j] == k)
            return false;
    return true;
}

bool absentSurColonne (int k, int grille[9][9], int j)
{
    for (int i=0; i < 9; i++)
        if (grille[i][j] == k)
            return false;
    return true;
}

bool absentSurBloc (int k, int grille[9][9], int i, int j)
{
    int _i = i-(i%3), _j = j-(j%3);  // ou encore : _i = 3*(i/3), _j = 3*(j/3);
    for (i=_i; i < _i+3; i++)
        for (j=_j; j < _j+3; j++)
            if (grille[i][j] == k)
                return false;
    return true;
```

```
}

bool estValide (int grille[9][9], int position)
{
    if (position == 9*9)
        return true;

    int i = position/9, j = position%9;

    if (grille[i][j] != 0)
        return estValide(grille, position+1);

    for (int k=1; k <= 9; k++)
    {
        if (absentSurLigne(k,grille,i) && absentSurColonne(k,grille,j) && absentSurBloc(k,gr
        {
            grille[i][j] = k;

            if ( estValide (grille, position+1) )
                return true;
        }
    }
    grille[i][j] = 0;

    return false;
}

int main (void)
{    clock_t t1, t2;

    t1 = clock();
    int grille[9][9] =
    {
   {0,0,0,0,0,0,0,3,9},
   {0,0,0,0,0,1,0,0,5},
   {0,0,3,0,5,0,8,0,0},
   {0,0,8,0,9,0,0,0,6},
   {0,7,0,0,0,2,0,0,0},
   {1,0,0,4,0,0,0,0,0},
   {0,0,9,0,8,0,0,5,0},
   {0,2,0,0,0,0,6,0,0},
   {4,0,0,7,0,0,0,0,0}
    };

    printf("Grille avant\n");
    affichage(grille);

    estValide(grille,0);

    printf("Grille apres\n");
    affichage(grille);
```

```
    t2 = clock();
    float temps;
    temps= (t2-t1);
    std::cout << "Temps de calcul: " <<temps/CLOCKS_PER_SEC << " secondes "<< "\n";

}
```

9.3.1 Thermodynamique inversée

```
ix=51477.d0
na=16807.d0
nmax=2147483647.d0

l=100

DO i=1,l
        ix=abs(ix*na)
        ix=mod(ix,nmax)
        x=(dble(ix)/dble(nmax))
        if (x<=0.25) then
                rad1=4
        elseif (x>=0.25 .and. x<=0.5) then
                rad1=4
        elseif (x>=0.5 .and. x<=0.75) then
                rad1=4
        else
                rad1=4
        endif

        ix=abs(ix*na)
        ix=mod(ix,nmax)
        rad2=(dble(ix)/dble(nmax))
        rad2=rad2*0.7+0.15

        ix=abs(ix*na)
        ix=mod(ix,nmax)
        temp=(dble(ix)/dble(nmax))
        temp=temp*10+20

        Un(i,1)=rad1
        Un(i,2)=rad2
        Un(i,3)=temp
enddo



N=50
dt=1.d0/100.d0

!do i=1,3
!print*,Un(i,3)
```

```
!ENDDO


do k=1,1000

do j=1,l

v=0.d0
r=0.d0
!print*,"i",i
rad1=Un(j,1)
rad2=Un(j,2)
temp=Un(j,3)
!print*,temp,"oo"

DO i=1,N
        x=0.35
        y=0.9
        ri=1
        do while (ri>0.00001)
                if (abs(x-0.5)>=abs(y-0.5)) then
                        ri=0.5-abs(x-0.5)
                else
                        ri=0.5-abs(y-0.5)
                endif

                if (ri>0.00001) then

                        ix=abs(ix*na)
                        ix=mod(ix,nmax)
                        c=(dble(ix)/dble(nmax))

                        ix=abs(ix*na)
                        ix=mod(ix,nmax)
                        t=(dble(ix)/dble(nmax))

                        x=x+ri*dcos(2*3.1415*c)
                        y=y+ri*dsin(2*3.1415*c)
                endif
        enddo
        if (rad1==1) then
                if (y<0.000001 .and. abs(x-rad2)<0.15) then
                        v=v+temp
                        !print*,"test"
                else
                        v=v+15.d0
                endif
        elseif (rad1==2) then
                if (x<0.000001 .and. abs(y-rad2)<0.15) then
                        v=v+temp
                else
```

```
                        v=v+15.d0
                endif
        elseif (rad1==3) then
                if (x>0.99999 .and. abs(y-rad2)<0.15) then
                        v=v+temp
                else
                        v=v+15.d0
                endif
        elseif (rad1==4) then
                if (y>0.99999 .and. abs(x-rad2)<0.15) then
                        v=v+temp
                        !print*,"test"
                else
                        v=v+15.d0
                endif
        endif
!Print*,v

ENDDO

v=v/N




DO i=1,N
        x=0.65
        y=0.9
        ri=1
        do while (ri>0.00001)
                if (abs(x-0.5)>=abs(y-0.5)) then
                        ri=0.5-abs(x-0.5)
                else
                        ri=0.5-abs(y-0.5)
                endif

                if (ri>0.00001) then

                        ix=abs(ix*na)
                        ix=mod(ix,nmax)
                        c=(dble(ix)/dble(nmax))

                        ix=abs(ix*na)
                        ix=mod(ix,nmax)
                        t=(dble(ix)/dble(nmax))

                        x=x+ri*dcos(2*3.1415*c)
                        y=y+ri*dsin(2*3.1415*c)
                endif
```

```
        enddo
        if (rad1==1) then
                if (y<0.000001 .and. abs(x-rad2)<0.15) then
                        r=r+temp
                else
                        r=r+15.d0
                endif
        elseif (rad1==2) then
                if (x<0.000001 .and. abs(y-rad2)<0.15) then
                        r=r+temp
                else
                        r=r+15.d0
                endif
        elseif (rad1==3) then
                if (x>0.99999 .and. abs(y-rad2)<0.15) then
                        r=r+temp
                else
                        r=r+15.d0
                endif
        else
                if (y>0.99999 .and. abs(x-rad2)<0.15) then
                        r=r+temp
                else
                        r=r+15.d0
                endif
        endif

ENDDO

r=r/N

!Print*,v,r

Un(j,4)=abs(v-19.d0)+abs(r-19.d0)

enddo

do i=1,l
        ix=abs(ix*na)
        ix=mod(ix,nmax)
        x=(dble(ix)/dble(nmax))
        if (Un(int(x*50)+1,4)<Un(i,4)) then
                Un(i,1)=Un(int(x*l)+1,1)

                ix=abs(ix*na)
                ix=mod(ix,nmax)
                x=(dble(ix)/dble(nmax))
                Un(i,2)=Un(int(x*l)+1,2) +x*0.1/(k)**(1/2)-0.05

                ix=abs(ix*na)
                ix=mod(ix,nmax)
```

```
                x=(dble(ix)/dble(nmax))
                Un(i,3)=Un(int(x*l)+1,3)+x/(k)**(1/2)-0.5

        endif
enddo

enddo

PRINT*,v,r
j=1
do i=1,l
        if (Un(i,4)<Un(j,4)) then
                j=i
        endif
enddo

print*,Un(j,1),Un(j,2),Un(j,3),Un(j,4)
```

9.3.2 coloration de graphe : méthode génétique

```
Un(1,1)=2
Un(1,2)=5
Un(1,3)=6

Un(2,1)=1
Un(2,2)=3
Un(2,3)=7

Un(3,1)=2
Un(3,2)=4
Un(3,3)=8

Un(4,1)=3
Un(4,2)=5
Un(4,3)=9

Un(5,1)=1
Un(5,2)=4
Un(5,3)=10

Un(6,1)=1
Un(6,2)=8
Un(6,3)=9

Un(7,1)=2
Un(7,2)=9
Un(7,3)=10

Un(8,1)=3
Un(8,2)=6
Un(8,3)=10
```

```
Un(9,1)=4
Un(9,2)=6
Un(9,3)=7

Un(10,1)=5
Un(10,2)=7
Un(10,3)=8

a=0


ix=51477.d0
na=16807.d0
nmax=2147483647.d0

do i=1,10

        ix=abs(ix*na)
        ix=mod(ix,nmax)
        x=(dble(ix)/dble(nmax))

        Un(i,4)=int(x*3)+1

enddo


do i=1,10
        Print*,Un(i,1),Un(i,2),Un(i,3),Un(i,4)
enddo

Print*,3

do k=1,9

do i=1,10

        Un(i,5)=0

        if (Un(i,4)==Un(Un(i,1),4) .or. Un(i,4)==Un(Un(i,2),4) .or. Un(i,4)==Un(Un(i,3),4))

                Un(i,5)=1

                ix=abs(ix*na)
                ix=mod(ix,nmax)
                x=(dble(ix)/dble(nmax))

                if (x<0.5) then

                        a=0

                        if (Un(i,4)==1) then
```

```
                                ix=abs(ix*na)
                                ix=mod(ix,nmax)
                                y=(dble(ix)/dble(nmax))

                                if (y<0.5) then
                                        a=2
                                else
                                        a=3
                                endif

                        elseif (Un(i,4)==2) then

                                ix=abs(ix*na)
                                ix=mod(ix,nmax)
                                y=(dble(ix)/dble(nmax))

                                if (y<0.5) then
                                        a=3
                                else
                                        a=1
                                endif

                        else

                                ix=abs(ix*na)
                                ix=mod(ix,nmax)
                                y=(dble(ix)/dble(nmax))

                                if (y<0.5) then
                                        a=1
                                else
                                        a=2
                                endif

                        endif

                        Un(i,4)=a

                endif

        endif

enddo


enddo
```

9.3.3 Coloration de graphe : méthode pseudo-génétique

```
ix=51477.d0
na=16807.d0
```

```
nmax=2147483647.d0


do k=1,p
do i=1,10

        ix=abs(ix*na)
        ix=mod(ix,nmax)
        x=(dble(ix)/dble(nmax))

        Wn(i,k)=int(x*3)+1

enddo
enddo

do l=1,10

do k=1,p

        ix=abs(ix*na)
        ix=mod(ix,nmax)
        x=(dble(ix)/dble(nmax))

        Wn(11,k)=0
        Wn(11,int(x*p)+1)=0

do i=1,10



        if (Wn(i,k)==Wn(Un(i,1),k) .or. Wn(i,k)==Wn(Un(i,2),k) .or. Wn(i,k)==Wn(Un(i,3),k))

                Wn(11,k)=Wn(11,k)+1

        endif

enddo

do i=1,10



        if (Wn(i,int(x*p)+1)==Wn(Un(i,1),int(x*p)+1) .or. Wn(i,int(x*p)+1)==Wn(Un(i,2),int(x
                &Wn(i,int(x*p)+1)==Wn(Un(i,3),int(x*p)+1)) then

                Wn(11,int(x*p)+1)=Wn(11,int(x*p)+1)+1

        endif

enddo
```

```
if (Wn(11,k)>=Wn(11,int(x*p)+1)) then
        do i=1,11
                Wn(i,k)=Wn(i,int(x*p)+1)
        enddo
        ix=abs(ix*na)
        ix=mod(ix,nmax)
        y=(dble(ix)/dble(nmax))
        ix=abs(ix*na)
        ix=mod(ix,nmax)
        z=(dble(ix)/dble(nmax))
        Wn(int(y*10)+1,k)=int(z*3)+1

!        ix=abs(ix*na)
!        ix=mod(ix,nmax)
!        y=(dble(ix)/dble(nmax))
!        ix=abs(ix*na)
!        ix=mod(ix,nmax)
!        z=(dble(ix)/dble(nmax))
!        Wn(int(y*10)+1,k)=int(z*3)+1
endif

enddo

enddo

k=Wn(11,1)
do i=2,p
        if (Wn(11,i)<k) then
                k=Wn(11,i)
        endif
enddo
```