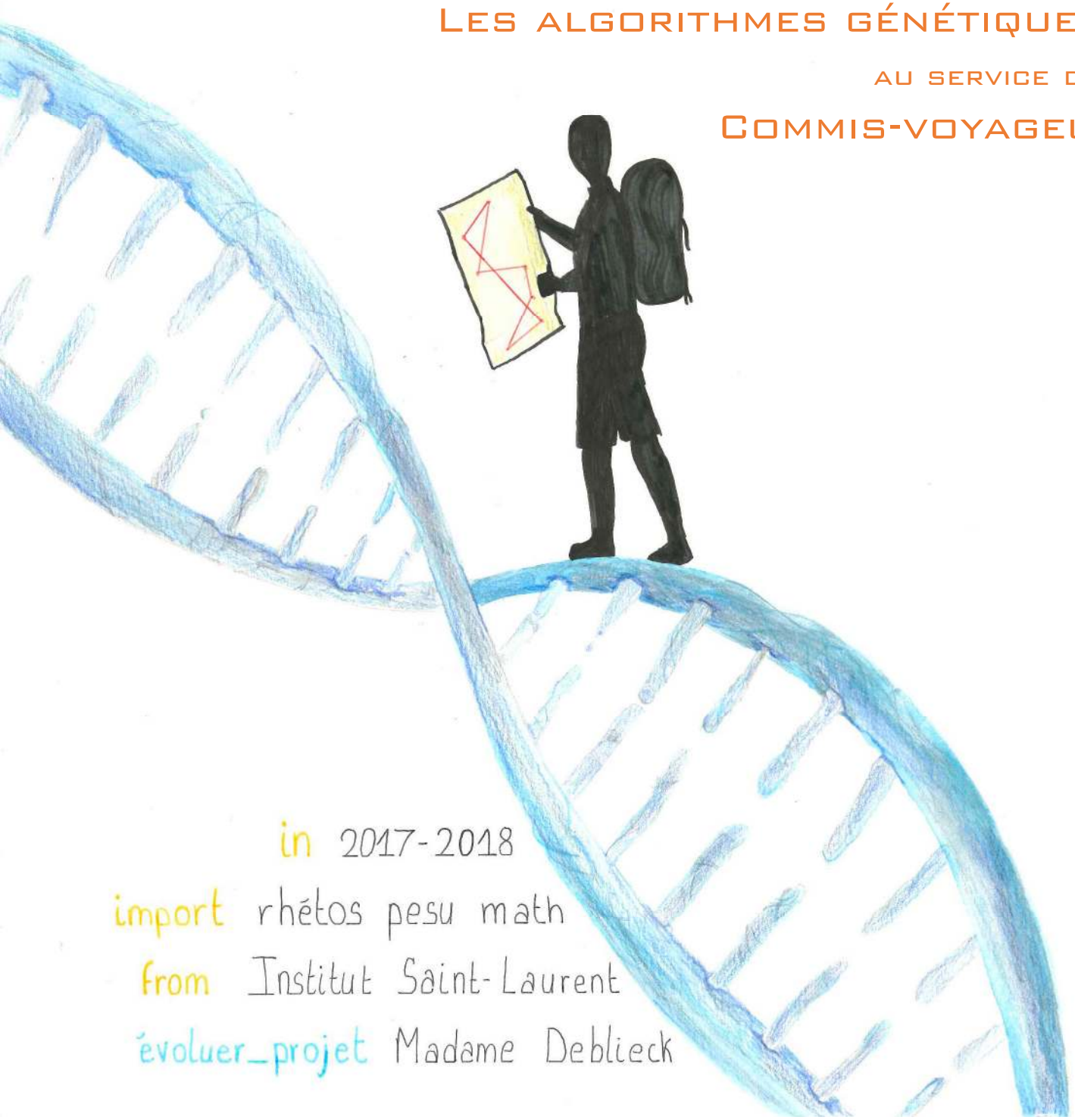
OPTiMATHiSATiON pythonienne

Les algorithmes génétiques au service du Commis-voyageur



in 2017-2018
import rhétos pesu math
from Institut Saint-Laurent
évoluer_projet Madame Deblieck

Institut Saint-Laurent

PESU Mathématique

OptiMATHisation pythonienne

Les algorithmes génétiques au secours du commis-voyageur

Auteurs :
Chenoy Firmin, Collard Thomas, Dubois Adélaïde, Fagny Damien, Goetz Isabeau, Grollinger William, Huet Camille, Joye Cyril, Jumpertz Sacha, Lannoy Robin, Lavens Fiona, Magnette Numa, Marchetto Alessio, Martin Florian, Rolland Hugo, Senger Antoine.

Professeur : Mme De Blieck

Date : 17 février 2018

Remerciements

Nous tenons tout d'abord à remercier Monsieur Hadrien Mélot, chargé de cours au département informatique de la Faculté des sciences de l'UMons et directeur du service d'Algorithmique. C'est lui qui nous a lancés dans l'aventure en intervenant dans notre école pour nous faire découvrir ce formidable outil que sont les algorithmes génétiques.

Nous voulons également exprimer toute notre gratitude à Monsieur Alexandre Mayer, chercheur qualifié au laboratoire de physique du solide de l'Université de Namur, qui a accepté de partager son expertise des algorithmes génétiques et qui, en répondant à nos questions, nous a éclairés lors de la phase d'évaluation de notre algorithme.

Table des matières

Table des figures

Introduction

L'optimisation est omniprésente et se manifeste partout autour de nous. Nous sommes souvent confrontés à des problèmes d'optimisation et notre but est toujours de vouloir le mieux plutôt que le plus : quel est le chemin qui me prendra *le moins* de temps ? quel investissement me rapportera *le plus* d'argent ? comment ranger *le plus* de chaussures dans mon dressing ?

Notre cours de mathématiques nous offre une méthode analytique imparable permettant de résoudre ce type de problème : la dérivée ! Cependant, d'une part l'utilisation de la dérivée nécessite la condition particulière de continuité du problème et d'autre part, la quête d'optimisation peut s'avérer très complexe dans de nombreux domaines industriels, économiques, biologiques, ménagers ou encore ludiques. La complexité de ces problèmes nous oblige à nous séparer de l'outil de dérivation. C'est là que les algorithmes entrent en jeu ; particulièrement les algorithmes génétiques, qui seront notés **AGs** dans la suite de ce document. Mais un peu de patience, parlons d'abord d'un problème d'optimisation bien connu : le *Problème du Voyageur de Commerce*.

La première partie de notre document est donc consacrée au *Problème du Voyageur de Commerce* (rebaptisé **PVC** dans ce travail), aussi appelé en anglais *The Travelling Salesman Problem*. Ce problème est posé en ces termes : étant donné n villes, quel est le chemin le plus court passant une et une seule fois par ces n villes et revenant au point de départ ? Ce problème est en fait la quête commune au livreur de pizzas, au facteur ou encore aux oniomaniaques : : « *comment optimiser mon itinéraire ?* ». Aucune méthode analytique connue ne permet de résoudre cet épineux problème. Par contre, de nombreux algorithmes tels que les AGs offrent une solution satisfaisante.

Revenons donc à nos algorithmes génétiques qui font l'objet de la deuxième partie de ce document...

Les algorithmes génétiques sont des algorithmes itératifs (basés sur la répétition) et stochastiques (en référence au hasard). Ils reposent sur les principes du néodarwinisme : la sélection naturelle et la recombinaison génétique. D'une part, la sélection naturelle, selon Charles Darwin, énonce que les individus les mieux adaptés sont plus aptes à survivre et à devenir parents de la prochaine génération. D'autre part, Gregor Mendel, père de la génétique, explique que des mutations au niveau des gènes permettent l'évolution des espèces. Ainsi, les mutations les plus adaptées à l'environnement sont les plus transmises et grâce à cet héritage génétique, les espèces actuelles deviennent des "versions optimisées" de leurs ancêtres.

Les phénomènes biologiques ont été une grande source d'inspiration pour les informaticiens. Le parfait exemple est donné par les algorithmes génétiques. Ceux-ci ont été initiés dans les années 1970 par le psychologue et scientifique John Holland. Ils imitent au sein d'un programme les mécanismes d'évolution dans la nature : croisement, mutation, sélection. En effet, dans un algorithme génétique, on part d'une population de solutions potentielles au problème, initialement choisies aléatoirement. Les solutions sont ensuite évaluées afin de déterminer leur capacité de survie et de désigner celles qui transmettront leurs gènes à la génération suivante. Les solutions changent donc constamment, les plus adaptées survivent et se reproduisent, les autres disparaissent. On recommence ce cycle jusqu'à obtenir une solution satisfaisante au problème.

Comment cette méthode bioinspirée permet-elle de résoudre le problème du voyageur de commerce ? C'est la troisième partie de ce document qui vous éclairera sur cette question. Loin de nous prendre pour des dieux de l'informatique, nous avons néanmoins souhaité expérimenter le fonctionnement d'un algorithme génétique en résolvant un PVC grâce à un programme que nous avons codé en langage Python. Vous découvrirez la structure de notre AG et la façon dont nous avons réalisé et testé celui-ci. Nous avons choisi comme exemple de PVC, le problème résolu par Georges Dantzig et son équipe en 1954. Le circuit à effectuer passe par 49 villes des Etats-Unis. Notre choix est motivé, d'une part, par l'aspect historique de ce problème - à l'époque, sa résolution fut considérée comme un exploit technologique - et d'autre part, par la possibilité de disposer d'éléments de comparaison. En effet, le nombre d'instances à traiter, bien que ridicule par rapport à ce qui se pratique aujourd'hui dans le domaine des PVC, semble raisonnable à notre niveau d'apprentis programmeurs. Par ailleurs, comme l'usage d'un AG ne garantit en rien une solution exacte, c'est grâce à la solution trouvée par la méthode de Dantzig que nous avons pu évaluer la performance de notre algorithme.

Si vous êtes novices des problèmes d'optimisation combinatoire ou des algorithmes, vous devrez sans doute, comme nous, "dormir dessus" pour bien comprendre les enjeux de l'optimisation et de sa résolution par AG. Cependant, nous avons mis tout notre cœur à trouver les explications les plus claires. C'est donc avec plaisir que nous vous invitons à la découverte d'une forme de darwinisme artificiel, qui pourrait bien vous aider à devenir plus rapides, plus forts voire meilleurs.

Première partie

I

L'épineux problème du voyageur de commerce

Chapitre 1

Un petit détour par la théorie des graphes

La théorie des graphes est le cadre naturel pour représenter le problème du voyageur de commerce. Aussi, nous faisons une petite parenthèse dans ce travail pour en exposer quelques principes. La théorie des graphes a pour but de faciliter la résolution d'un problème en représentant par un modèle abstrait les relations entre différents objets. Pour un seul et même problème, il existe donc une multitude de graphes possibles.

1.1 Quelques définitions

Définition 1 *Un graphe fini G =(V,E) est défini par l'ensemble fini* $V = \{v_1, v_2, ..., v_n\}$ *dont les éléments sont appelés* ***sommets*** *(parfois nœuds, cellules ou vertices en anglais), et par l'ensemble fini* $E = \{e_1, e_2, ..., e_m\}$ *dont les éléments sont appelés* ***arêtes*** *(ou liens, arcs ou encore edges en anglais).*

Une **arête** relie deux sommets entre eux et se note *(x,y)* avec x et y les extrémités de l'arête. Deux sommets reliés par une arête sont dits adjacents. Deux arêtes ayant un sommet en commun sont dites adjacentes. Les arêtes peuvent parfois être orientées, dans ce cas, elles se présentent sous forme de flèches qui définissent un sens de circulation.

Définition 2 *Le* ***degré*** *d'un sommet x de G est le nombre d'arêtes incidentes à x et est noté d(x).*

Définition 3 *Le nombre de sommets d'un graphe G est appelé* ***ordre*** *du graphe.*

Définition 4 *Un graphe est* ***connexe*** *s'il est possible, à partir de n'importe quel sommet, de rejoindre tous les autres sommets en suivant les arêtes.*

Définition 5 *Un graphe est* ***complet*** *si chaque sommet est adjacent à tous les autres.*

Le graphe complet d'ordre n est noté K_n. Dans ce graphe, chaque sommet est de degré *n-1.*

Définition 6 *Un graphe est* ***valué*** *lorsqu'à chaque arête ou arc est associé un nombre réel.*

Si ce nombre est positif, on parle de **poids** et de **graphe pondéré**. Lorsque les nombres (si compris entre 0 et 1) représentent des probabilités, on peut parler de graphe probabiliste. Soit un graphe $G = (V, E)$. Chaque arête e_i (arc s'il s'agit d'un graphe orienté) est munie d'un poids p_i.

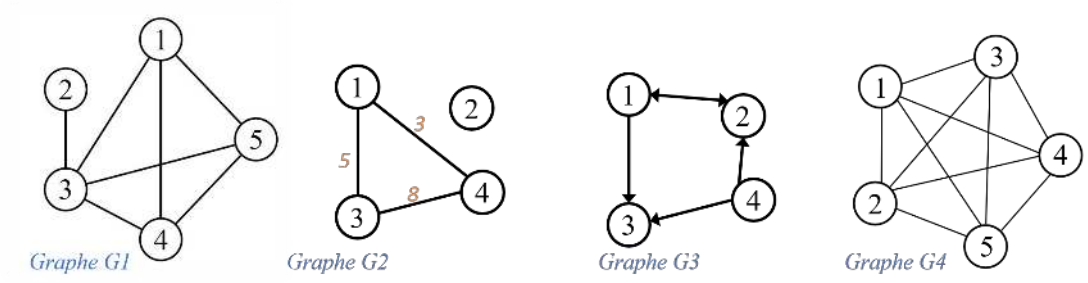


FIGURE 1.1 – G1 est un graphe connexe dans lequel le sommet 4 est de degré 3. G2 n'est pas connexe car le sommet 2 est isolé mais il est pondéré. G3 est un graphe orienté. G4 est un graphe complet d'ordre 5.

1.2 Chemins, cycles et hamiltonicité

Le problème du voyageur de commerce est assimilé à une promenade sur un graphe. Ceci amène les définitions mathématiques de *chemin* et de *cycle*.

Définition 7 *Un **chemin** ou chaîne est une liste $p = (v_1, v_2, \ldots, v_n)$ de sommets telle qu'il existe dans le graphe G(V,E) une arête entre chaque paire de sommets successifs, c-à-d $\forall i = 1, \ldots, n-1$: $(v_i, v_{i+1}) \in E$.*

La longueur du chemin correspond au nombre d'arêtes parcourues : *n-1*. Un chemin p est dit **simple** si chaque arête du chemin est empruntée une seule fois.

Définition 8 *Un **cycle** $c = (v_1, v_2, \ldots, v_n)$ est un chemin simple finissant à son point de départ, c-à-d tel que $v_i = v_n$.*

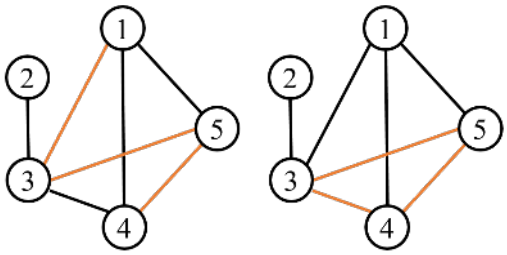


FIGURE 1.2 – Dans la représentation de gauche, $p = (1, 3, 5, 4)$ est un chemin de longueur 3 qui relie le sommet *1* au sommet *4*. Dans la représentation de droite, $c = (3, 4, 5, 3)$ est un cycle.

Chaînes et cycles eulériens

Définition 9 *On appelle **chaîne eulérienne** d'un graphe G une chaîne passant une et une seule fois par chacune des arêtes de G.*

Définition 10 *On appelle **cycle eulérien** d'un graphe G un cycle passant une et une seule fois par chacune des arêtes de G.*

Un graphe est dit **eulérien** s'il possède un cycle eulérien. Un graphe est **semi-eulérien** lorsqu'il possède une chaîne eulérienne : on peut le tracer en passant par toutes les arêtes une et une seule fois (sans l'obligation de finir au sommet par lequel on a commencé). Dans un graphe eulérien, on ne peut passer sur chaque arête qu'une et une seule fois, mais on peut passer plusieurs fois par chaque sommet. L'adjectif « *eulérien* » doit son origine au grand mathématicien Léonhard Euler (1707-1783) qui énonça les théorèmes suivants.

Théorème 1 *Un graphe simple connexe G=(V,E) est eulérien si et seulement si tous ses sommets sont de degré pair : $\forall v \in V$, d(v) est pair.*

Théorème 2 *Un graphe simple connexe G=(V,E) est semi-eulérien ssi au plus 2 de ses sommets sont de degré impair.*

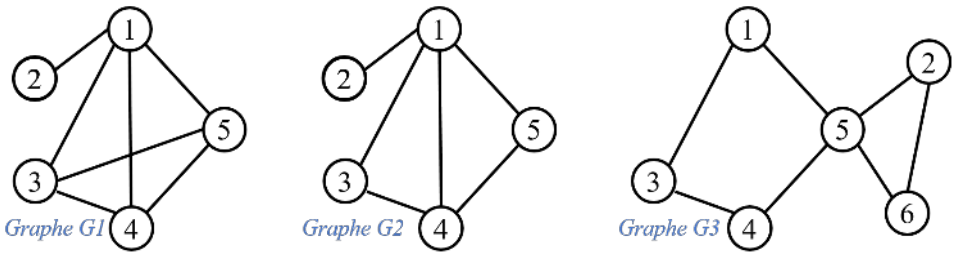


FIGURE 1.3 – G1 ne possède pas de chaîne eulérienne : 4 de ses sommets sont de degré 3. G2 possède la chaîne eulérienne (2,1,5,4,1,3,4). Il n'est cependant pas eulérien car 2 de ses sommets sont de degré impair. G3 est eulérien car tous ses sommets sont de degré pair. Un cycle eulérien est (5,2,6,5,1,3,4,5).

Chaînes et cycles hamiltoniens

Définition 11 *On appelle* ***chaîne hamiltonienne*** *d'un graphe G une chaîne passant une et une seule fois par chacun des sommets de G.*

Définition 12 *On appelle* ***cycle hamiltonien*** *d'un graphe G un cycle passant une et une seule fois par chacun des sommets de G.*

Un graphe est dit **hamiltonien** s'il possède un cycle hamiltonien. Un graphe ne possédant que des chaînes hamiltoniennes est **semi-hamiltonien**. Un graphe hamiltonien passe par tous les sommets, mais pas nécessairement par toutes les arêtes.

Il n'existe pas de critères précis pour déterminer un graphe hamiltonien, contrairement aux graphes eulériens. Il existe tout de même quelques propriétés et conditions. Cependant, les conditions ne sont que suffisantes et non pas nécessaires. Nous pouvons donc, grâce à cela, affirmer qu'un graphe est hamiltonien s'il répond à l'une des conditions mais il faut aussi savoir qu'un graphe ne répondant pas aux conditions, peut tout de même être hamiltonien.

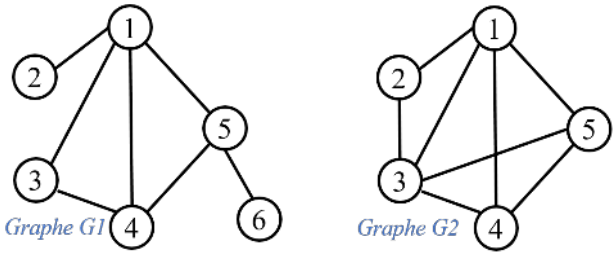


FIGURE 1.4 – Un graphe possédant un sommet de degré 1 ne pouvant être hamiltonien, G1 ne l'est pas. Par contre, il est aisé de voir le cycle hamiltonien (1,2,3,4,5,1) dans G2.

L'origine du problème hamiltonien remonterait au $XVIII^e$ siècle lorsqu'Euler se demanda comment un cavalier pourrait visiter toutes les cases d'un échiquier.

Cependant, nous devons la terminologie de ces graphes à W. R. Hamilton qui, en 1857, étudia ces cycles lorsqu'il inventa le casse-tête du « *Tour du monde* », connu aussi sous le nom d' *Icosian Game*. Le jeu se présentait sous la forme d'un dodécaèdre de bois, c'est-à-dire un polyèdre à 12 faces en forme de pentagone régulier. Les 20 sommets de ce dodécaèdre portaient les noms de différentes villes du monde. Le jeu consistait à partir d'une ville quelconque et à voyager le long des arêtes du dodécaèdre de manière à passer une fois seulement par les 19 autres villes, puis de revenir au point de départ. C'est le premier pas vers la formulation du problème du voyageur de commerce.

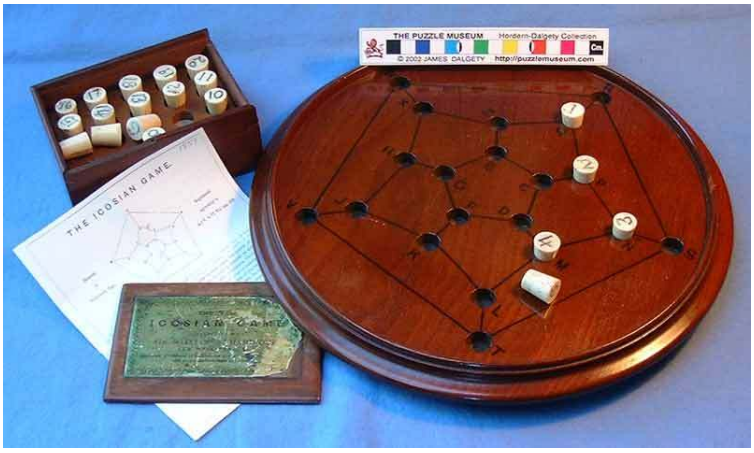


FIGURE 1.5 – L'*Icosian Game* : l'ancêtre du PVC [12].

1.3 Domaines d'application

La théorie des graphes a de nombreuses finalités. Réseaux de télécommunications (internet, téléphonie, transport,...), circuits éléctriques, organisation de fichiers informatiques, bases de données relationnelles, stockage de données, codage, représentation des molécules, ordonnance de tâches, ... sont autant de problèmes concrets qui se modélisent et se résolvent grâce à cette branche des mathématiques discrètes. Nous trouvons dans la théorie des graphes un intérêt particulier car le problème du voyageur de commerce compte au nombre de ces applications. En effet, le PVC n'est rien d'autre que la quête du cycle hamiltonien de poids minimum dans un graphe dont les sommets sont les villes à visiter.

Beaucoup de problèmes de tournée de distribution se formulent en termes de graphes. Un autre métier itinérant utilise également les principes de cette théorie : le *Postier Chinois*. C'est l'histoire d'un facteur qui veut optimiser la distribution du courrier en parcourant, au moins une fois, chacune des rues d'une ville et en cherchant à minimiser la distance totale parcourue. D'un point de vue mathématique, il doit trouver un chemin de poids minimum entre deux sommets du graphe en utilisant chaque arête au moins une fois. Ce problème a été posé, pour la première fois, par le Chinois Kwan Mei-Ko, en 1962. Bien que très semblables dans leur formulation, le problème du postier chinois et celui du voyageur de commerce sont très différents dans leur résolution. Le problème du postier chinois, apparenté aux graphes eulériens, est de nature plus facile que le problème du voyageur de commerce. Téméraires ou peut-être un peu naïfs, nous n'avons cependant pas reculé devant la difficulté : nous partons à la conquête du problème du voyageur de commerce.

Chapitre 2

Il était une fois un vendeur itinérant...

Lorsqu'il pose le problème pour la première fois en 1857, sous la forme de défi, William Rowan Hamilton énonce : « *Un voyageur de commerce doit visiter une et une seule fois un nombre fini de villes et revenir à son point d'origine. Trouvez l'ordre de visite des villes qui minimise la distance totale parcourue par le voyageur* ». Le problème du voyageur de commerce, que nous noterons PVC dans ce travail, est un problème connu depuis longtemps. Son nom, *Problème du voyageur de commerce*, connu également dans sa version anglo-saxonne *Traveling Salesman Problem*, évoque ce que chaque représentant de commerce ou chaque voyageur doit résoudre afin de minimiser le coût de ses tournées et de ses voyages.

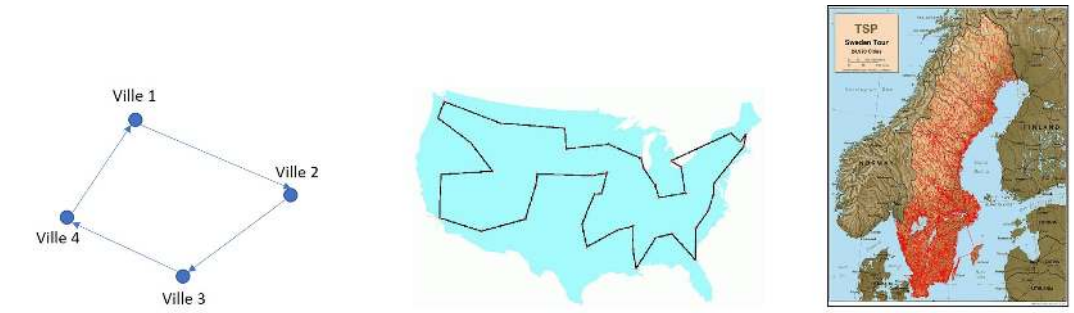


FIGURE 2.1 – Quelques exemples de PVC [10].

2.1 Historique

Les origines du PVC ne sont pas bien connues. Sa première mention remonterait à un livre allemand édité en 1832, avec un exemple sur 45 villes. Mais ce n'est que dans les années 20 que Karl Menger, mathématicien et économiste autrichien, donne au problème sa formulation mathémathique. Vers 1930, le problème réapparaît dans les cercles mathématiques de Princeton.

En 1954, une équipe de chercheurs américains, George Dantzig, Ray Fulkerson et Selmer Johnson publient la description d'une méthode pour résoudre le PVC et illustrent la puissance de celle-ci en résolvant un PVC de 49 villes, chose très impressionnante à cette époque. Le circuit qu'ils choisissent compte une ville de chacun des 48 états des États-Unis (l'Alaska et Hawaii n'en font pas partie à l'époque) et ajoutent Washington, D.C . ; les coûts de déplacement entre ces villes sont définis par les distances routières.

Mais c'est le concours organisé en 1962 par la multinationale américaine *Proctor and Gamble* qui contribue à rendre le problème populaire. Le concours prévoit la résolution d'un PVC de 33 villes et une belle prime est offerte au vainqueur.

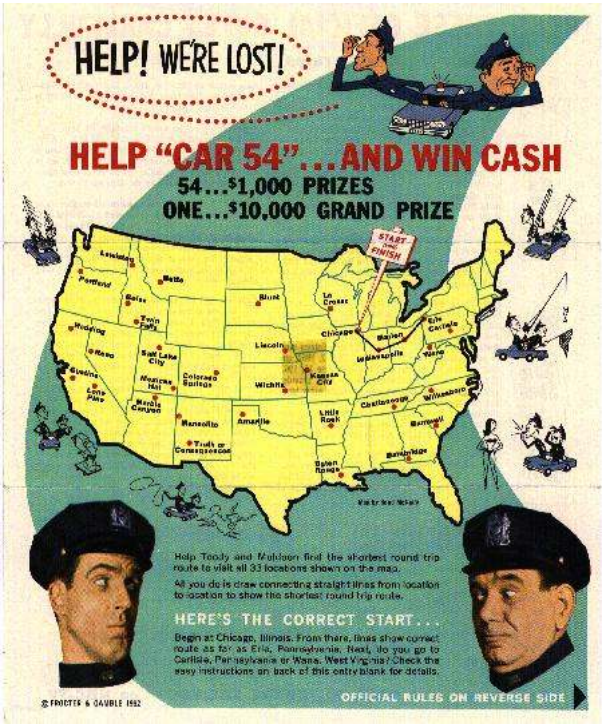


FIGURE 2.2 – Affiche du concours lancé par Proctor & Gamble en 1962 [10].

Richard Manning Karp (mathématicien et informaticien américain) démontre en 1972 que le PVC est *NP-complet* : il ne peut donc pas être résolu dans des temps polynomiaux (raisonnables) [1]. Il faut donc préférer une technique d'approximation à une technique de résolution exacte.
Pourtant, depuis son apparition, les techniques de résolution du PVC ne cessent de s'améliorer grâce entre autres aux avancées importantes réalisées dans le domaine de l'algorithmique. Aujourd'hui, il est possible de résoudre le problème avec plusieurs dizaines de milliers de villes et la course folle aux records n'est pas prête de s'arrêter.

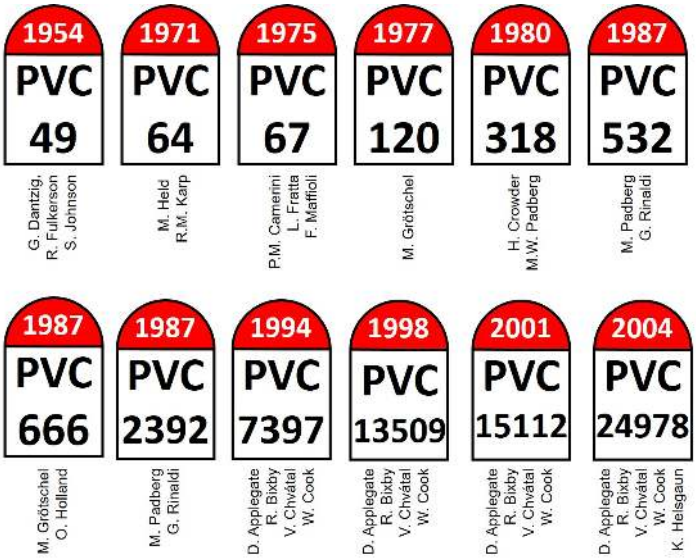


FIGURE 2.3 – Grands jalons de l'histoire des records du PVC. En noir, le nombre de villes traitées. Sous chaque borne kilométrique, l'équipe de recherche.

1. Cfr. Qu'est-ce que la complexité d'un problème ? en page 22

2.2 La grande complexité du PVC

Il ne faut pas se fier à l'apparente simplicité de l'énoncé du PVC. Sa difficulté réside principalement dans le fait qu'une exploration exhaustive de tous les circuits peut très vite s'avérer impossible. Pour estimer le nombre de circuits fermés différents visitant une et une seule fois n villes que nous devrions tester, fixons la première. Comme le circuit est fermé, ce choix n'a aucune conséquence pour la suite. Etant donné cette ville de départ, nous avons *n-1* possibilités pour la deuxième ville, *n-2* possibilités pour la troisième ville, etc. La multiplication de ces possibilités donne $(n-1)!$. Cependant, puisque nos frais de voyage ne dépendent pas de la direction que nous prenons pour effectuer le circuit, nous devons diviser ce nombre par 2 pour obtenir $\frac{(n-1)!}{2}$.

Nombre de villes n	**Nombre de circuits**	**Temps de calcul**
10	181 440	0,18 ns
20	$6{,}1.10^{16}$	61 s
25	$3{,}1.10^{23}$	9,8 années
49	$6{,}2.10^{60}$	2.10^{38} années

TABLE 2.1 – Nombre de circuits à tester en fonction de n et temps de calcul par un supercalculateur réalisant 10^{15} opérations par seconde.

La croissance explosive de $\frac{(n-1)!}{2}$ exclut la possibilité de vérifier tous les circuits un par un. Cette méthode de résolution peut être utilisée mais devient vite impraticable une fois les 20 villes dépassées. Un supercalculateur ne peut effectuer "que" 10^{15} opérations par seconde. Ce nombre peut sembler énorme au premier abord mais ce n'est pas suffisant pour des problèmes dont le nombre de villes se compte en dizaines. Si l'exploration d'un PVC à 10 villes se réalise de façon presqu'instantanée, un PVC de 25 villes demande pratiquement 10 années de recherche et le PVC de 49 villes résolu par G.Dantzig, R.Fulkerson et S.Johnson en 1954 plusieurs milliards de milliards de milliards de milliards d'années de traitement pour trouver le meilleur circuit. Résoudre le problème du voyageur de commerce signifie rechercher un algorithme qui rendra le nombre de calculs acceptable.

2.3 L'utilité du PVC

Une grande partie de la recherche sur le PVC est motivée par son utilisation comme prototype pour l'étude des méthodes générales qui peuvent être appliquées à un large éventail de problèmes d'optimisation. Cela ne veut toutefois pas dire que le PVC ne trouve pas d'application directe dans de nombreux domaines.

Spontanément, le PVC apparaît comme un sous-problème dans de nombreuses applications de transport et de logistique : organisation d'itinéraires de bus, livraison des repas par le service social, ramassage et distribution des colis postaux et une foule d'autres tournées.

Bien que les applications de transport soient le cadre le plus naturel pour le PVC, la simplicité du modèle a conduit à de nombreuses applications intéressantes dans d'autres domaines. Un exemple classique est la programmation d'une machine pour percer des trous dans une carte de circuit imprimé. Dans ce cas, les trous à forer sont les villes et le coût du voyage est le temps qu'il faut pour déplacer la tête de forage d'un trou à l'autre. Le temps de déplacement du dispositif de forage représentant une part importante du processus de fabrication global, le PVC peut jouer un rôle dans la réduction des coûts de fabrication.

En bref, la méthodologie mise en place pour la résolution du PVC rencontre une utilité pour le traitement des problèmes de transport, d'ordonnancement des tâches et l'optimisation des trajectoires dans l'industrie.

2.4 Vers la résolution du problème du voyageur de commerce

Plusieurs méthodes ont été envisagées pour résoudre le problème du voyageur de commerce, elles ont chacune leurs avantages et leurs inconvénients. On peut tout d'abord naïvement essayer de résoudre le problème soi-même, sans avoir recours à aucun outil si ce n'est "le talent". La méthode peut s'avérer très efficace lorsque le nombre de villes se compte sur les doigts de la main, mais dès que le nombre de villes dépasse la dizaine, le simple "talent" ne suffit plus et la chance prend le relais. On comprend vite que ce n'est pas avec ce genre de méthode que l'on pourra résoudre efficacement le problème. Nous devons avoir recours aux algorithmes, il en existe beaucoup qui permettent de résoudre le problème.
Avant d'aborder de façon approfondie la méthode qui fait l'objet de ce travail, l'algorithme génétique, nous présentons dans cette partie quatre autres méthodes choisies pour leur simplicité d'exécution : l'algorithme du *Cheapest Link* , le *Plus proche voisin*, la *colonie de fourmis* et enfin, la méthode du *Branch and Bound*.

2.4.1 L'algorithme du *Cheapest link*

Le *Cheapest Link* est un algorithme glouton [2]. Un algorithme glouton construit une solution pas à pas par décisions successives sans possibilité de retour en arrière. Pour former son circuit optimal, le *Cheapest Link* sélectionne les plus petites arêtes du graphe et rejette celles qui referment prématurément le circuit tant que celui-ci ne passe pas par tous les sommets.

Un exemple simple étant plus parlant qu'un long discours, les étapes de l'algorithme sont présentées ci-dessous en images. Nous considérons un PVC représenté par un graphe complet à 5 sommets, dont le poids des arêtes est donné.

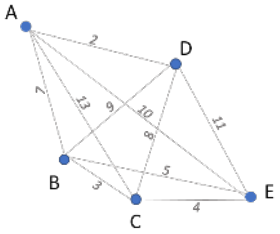


FIGURE 2.4 – Présentation du problème à résoudre.

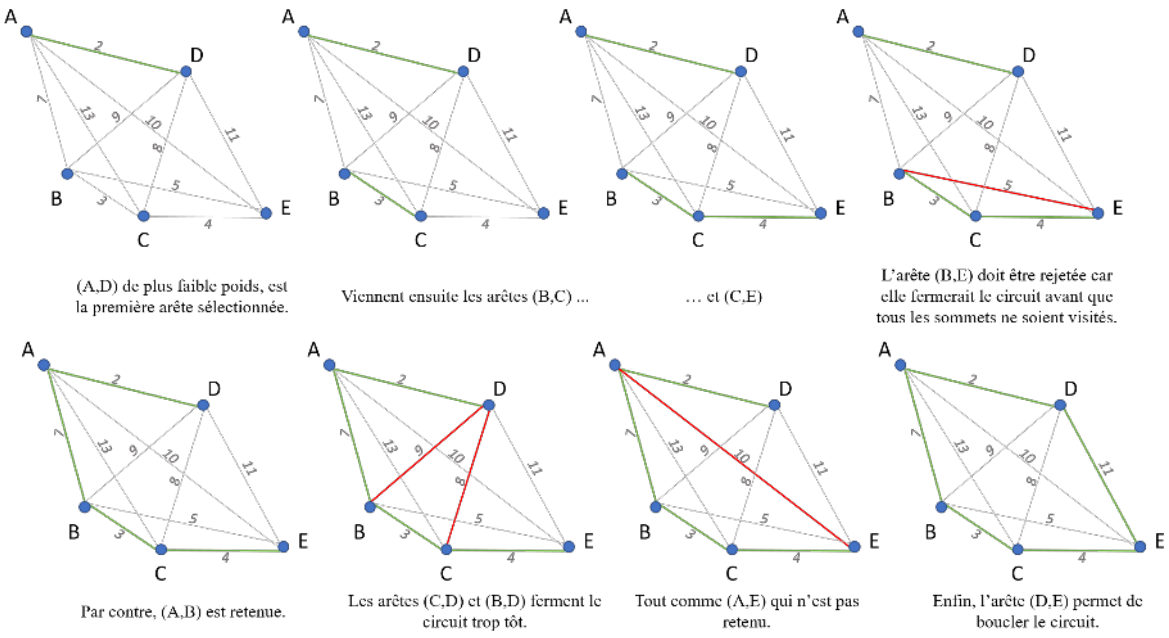


FIGURE 2.5 – Résolution du problème par le *Cheapest Link*.

2. Cfr. page 21

2.4.2 L'algorithme du *Plus proche voisin*

Il s'agit d'un autre algorithme glouton. Comme son nom l'indique, l'algorithme du plus proche voisin part d'une ville , visite ensuite la plus proche et répète le processus jusqu'à ce que toutes les villes soient visitées et qu'on puisse revenir à la ville de départ. Une décision localement optimale est prise à chaque étape de l'algorithme. Une chose à noter à propos de cet algorithme est que nous devons choisir un sommet de départ. Ce choix affecte le circuit retenu : un autre sommet fournit un autre circuit qui pourrait être plus court. On peut le constater en reprenant l'exemple précédent à cinq sommets. Si nous fixons notre ville de départ à A, l'algorithme propose la solution ADCBEA qui est un circuit de poids total 28. C'est moins bien que ce que propose le *Cheapest Link*.

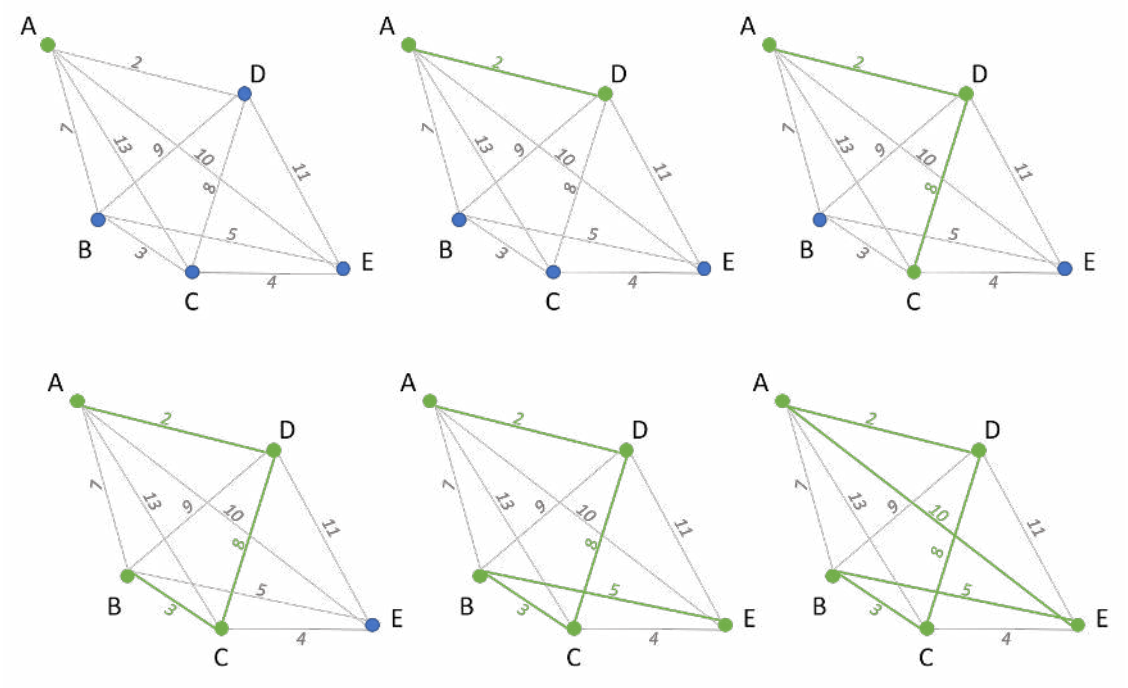


FIGURE 2.6 – Mise en oeuvre du *Plus proche voisin* en partant du sommet A.

Mais si nous fixons notre ville sur E, le circuit retenu est ECBADE pour un poids de 27. La deuxième tentative est donc meilleure que la première.

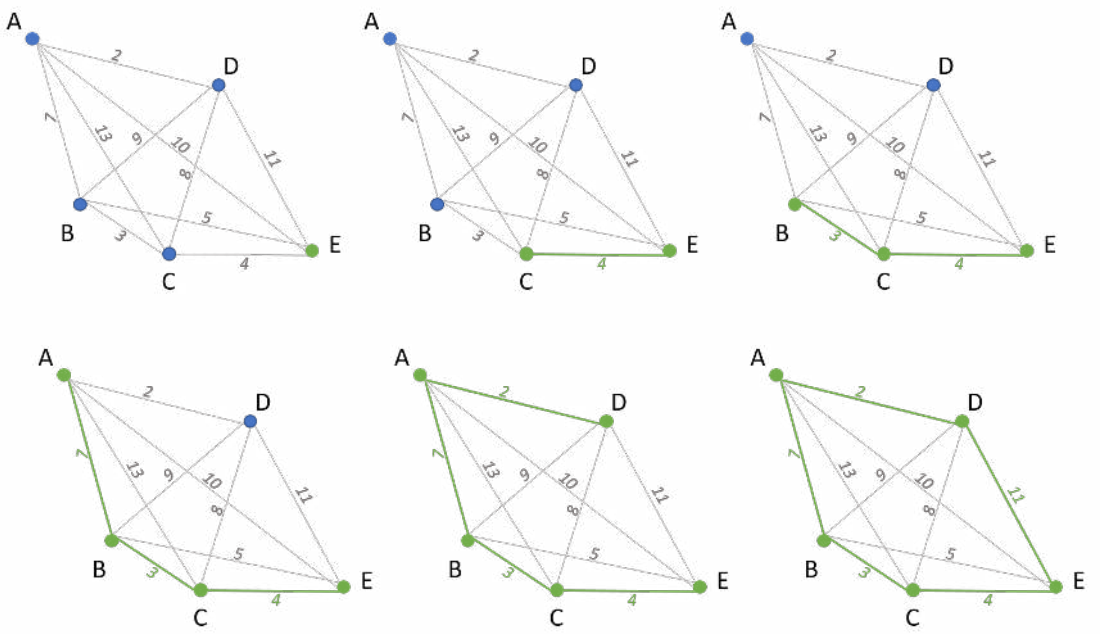


FIGURE 2.7 – Mise en oeuvre du *Plus proche voisin* en partant du sommet E.

Ceci illustre bien le caractère approximatif des algorithmes employés pour résoudre le PVC : on accepte de ne pas accéder à la solution exacte au profit d'un temps de calcul meilleur.

2.4.3 La colonie de fourmis

L'algorithme des colonies de fourmis fait partie des méta-heuristiques[3], ces algorithmes non-déterministes qui utilisent la nature comme modèle. Cette méthode est basée sur le comportement des colonies de fourmis dans la nature. Des biologistes ont observé que les fourmis sont capables collectivement de trouver le chemin le plus court entre une source de nourriture et leur nid. Cette faculté trouve son origine dans les phéromones, substance chimique sécrétée par les insectes, qui permet aux fourmis de s'échanger des informations.

Une des villes est désignée comme fourmilière et une première fourmi en sort pour prendre un chemin aléatoire et revient à la fourmilière. Sur le chemin, elle a laissé une trace de phéromone sur chaque arête. Une deuxième fourmi sort à son tour de la fourmilière et effectue un autre tour, en déposant elle aussi, des phéromones en chemin. Les fourmis privilégient le chemin ayant une forte concentration en phéromones. Plus l'arête est empruntée, plus les autres fourmis ont tendance à la parcourir.

Or, les phéromones se dissipent avec le temps. Plus une arête est longue, plus l'évaporation de phéromones est élevée et moins les fourmis circulent sur l'arête. Cette tendance permet d'éliminer du circuit les arêtes les plus longues. Le programme fixe donc un indice d'évaporation des phéromones afin que le circuit puisse évoluer en mieux. Il fait en sorte que seuls les meilleurs chemins soient conservés.

Encore une fois, un exemple est plus parlant. Reprenant notre PVC à cinq sommets, nous fixons arbitrairement la fourmilière en A et nous envoyons 4 fourmis éclaireuses qui repèrent 4 circuits différents : vert (AEDBCA), rouge (ADEBCA), bleu (ADCBEA) et jaune (ABCDEA), de longueur respective 46, 34, 28 et 39.

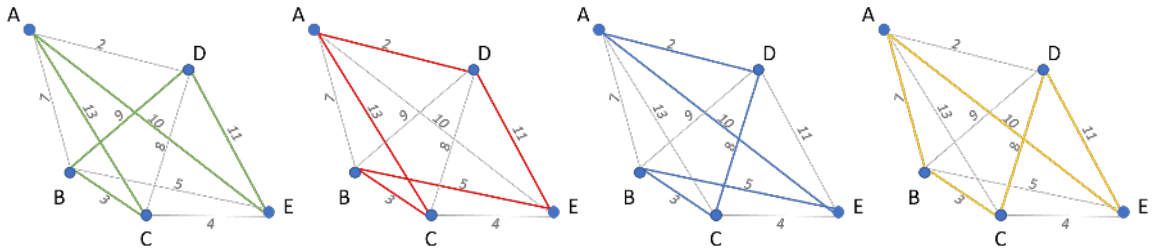


FIGURE 2.8 – Les 4 premiers circuits de la colonie de fourmis.

Nous choisissons, pour cet exemple, un indice d'évaporation de 3 : les circuits disparaissent quand trois autres sont meilleurs. Vert s'efface donc car c'est le plus long de nos quatre chemins. Le processus recommence alors : une cinquième fourmi quitte la fourmilière pour tracer un nouveau circuit et le moins bon de ces quatre nouveaux circuits est éliminé. Après un nombre arbitraire d'itérations, toutes les fourmis parcourent le même circuit. Ce signe de convergence nous indique que le processus est terminé et que l'algorithme est prêt à nous fournir sa solution optimale. Nous retrouvons d'ailleurs la solution proposée par les deux algorithmes gloutons présentés plus tôt.

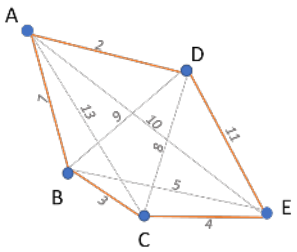


FIGURE 2.9 – La solution retenue par la colonie de fourmis.

3. Cfr. page 21

2.4.4 La méthode *Branch and Bound*

Bien que le PVC possède un ensemble fini de solutions possibles, le nombre prohibitif de ces solutions rend difficile l'énumération exhaustive et invite à la résolution par des méthodes heuristiques telles que la *Colonie de fourmis* ou le *Plus proche voisin*.

Il existe cependant des méthodes de résolution exactes. L'une d'elles est la méthode de *Branch and Bound*, ou, en français, *Procédure par Evaluation et Séparation progressive*. *Séparation* car le problème est divisé afin de constituer un "arbre de décisions" – c'est le *Branch* – et *Evaluation*, car la poursuite des investigations sur une branche relève d'une décision à prendre - c'est le *Bound*.

Cette méthode *Branch and Bound* consiste donc à énumérer les solutions d'une manière intelligente de sorte à éliminer en cours de processus des solutions partielles qui ne mènent pas à la solution que l'on recherche. De ce fait, on arrive souvent à obtenir la solution recherchée en des temps raisonnables.

Pour ce faire, cette méthode se dote d'une fonction qui permet de mettre une borne sur certaines solutions pour, soit les exclure, soit les maintenir comme des solutions potentielles.

Par convenance, on représente l'exécution de la méthode de *Branch and Bound* à travers une arborescence. La racine de cette arborescence représente l'ensemble de toutes les solutions du problème considéré.

On se déplace ensuite dans l'arborescence de telle sorte que toute solution partielle dont le poids total est plus grand que celui de la meilleure solution trouvée jusqu'à présent est exclue de la recherche. Ce processus est continué jusqu'à avoir une solution complète.

Si le poids de cette nouvelle solution complète est inférieur à celui de la meilleure solution déjà trouvée, le nouveau poids de référence est considéré comme étant celui de la meilleure solution courante.

Appliquons cette méthode à notre exemple en cours. On réalise d'abord un tableau à double entrée présentant les poids des arêtes reliant deux sommets entre eux. Comme le graphe représentant notre exemple est complet et non-orienté, nous obtenons un tableau symétrique par rapport à la diagonale principale. Nous prenons au hasard une ville de départ - ici, le sommet A -, ce qui n'affecte en rien la solution optimale puisque nous recherchons des cycles : il n'y a ni début ni fin, la ville choisie devra être visitée, quel que soit le cycle choisi.

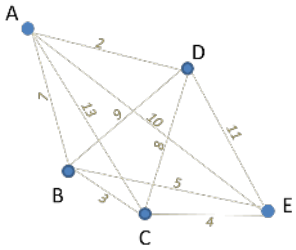
Nous déterminons ensuite une borne minorant le poids total de tout cycle hamiltonien pour le graphe. Le calcul de cette borne se fait grâce au constat suivant.

Soit le cycle hamiltonien $(v_1, v_2, \ldots, v_n, v_{n+1} = v_1)$, il doit y avoir au moins deux arêtes incidentes à chaque sommet v_i du graphe : une arête sortante et une arête entrante. Le poids de l'arête sortante du sommet v_i est donné par $D[i, j]$: cela correspond à l'élément de la i^{me} ligne et j^{me} colonne de notre tableau), pour un certain $j \neq i$. Le poids de l'arête entrante est aussi donné par $D[k, i]$, pour un certain $k \neq j$, les deux arêtes devant être distinctes. Par conséquent, quel que soit un cycle hamiltonien, la somme des poids de deux arêtes incidentes à un sommet v_i doit être supérieure à celle des plus petits poids des arêtes incidentes à v_i.

En sommant sur les i, on compte par deux fois chacune des arêtes : elles sont à la fois sortantes pour un sommet et rentrantes pour un autre.

Autrement dit, quelle que soit la solution optimale, son coût est supérieur à $\frac{1}{2}\sum_1^n(e_{ini} + e_{outi})$ où e_{ini} et e_{outi} désignent deux arêtes adjacentes de poids minimum au sommet v_i.

Mettons ceci en musique. Le tableau à double entrée est présenté à la figure 2.10. Le minorant trouvé est $\frac{1}{2}(7 + 2 + 3 + 5 + 3 + 4 + 2 + 8 + 5 + 4) = 21,5$.



	A	B	C	D	E
A		7	13	2	10
B	7		3	9	5
C	13	3		8	4
D	2	9	8		11
E	10	5	4	11	

FIGURE 2.10 – Tableau de pondération des arêtes. En rouge, les arêtes entrantes et sortantes de poids minimum.

Les sommets suivants dans l'arborescence peuvent être B, C, D ou E. Pour chacune des solutions partielles, on calcule une nouvelle borne : c'est-à-dire le minorant du poids d'un cycle intégrant une arête désignée par la branche. Ainsi, pour le sommet B, la nouvelle borne est $\frac{1}{2}(7+2+3+\mathbf{7}+3+4+2+8+5+4) = 22,5$. La nouvelle borne doit prendre en compte le passage obligatoire par l'arête reliant A à B. La valeur 22,5 représente le plus petit poids d'un cycle hamiltonien incluant les arêtes (A,B) et (B,A).

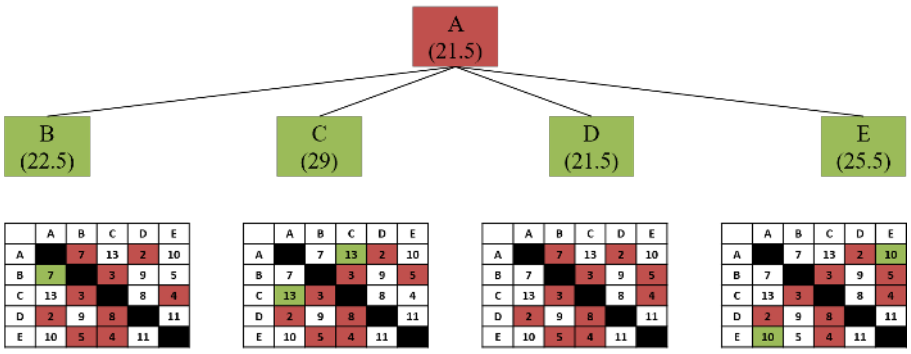


FIGURE 2.11 – Arborescence de premier niveau ayant comme racine le sommet A.

Une nouvelle borne est calculée pour chaque sommet. On obtient 29 pour le sommet C, 21,5 pour le sommet D et 25,5 pour le sommet E. Le sommet D possédant la plus petite valeur, on continue à explorer cette solution en descendant plus bas dans l'arborescence et en recalculant de nouvelles bornes.

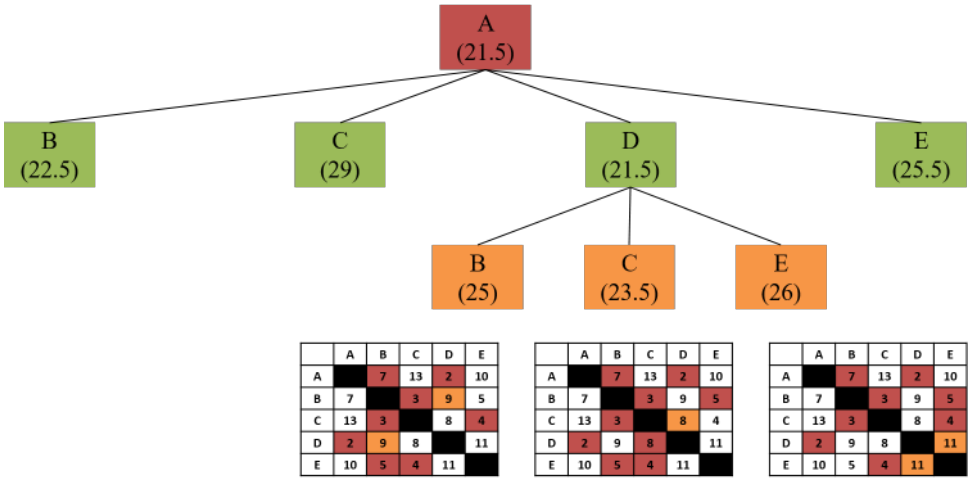


FIGURE 2.12 – Arborescence de deuxième niveau.

Les nouvelles solutions partielles issues de D montrent des poids supérieurs à la solution partielle de premier niveau B. On change de branche et on mène l'investigation sur le sommet B.

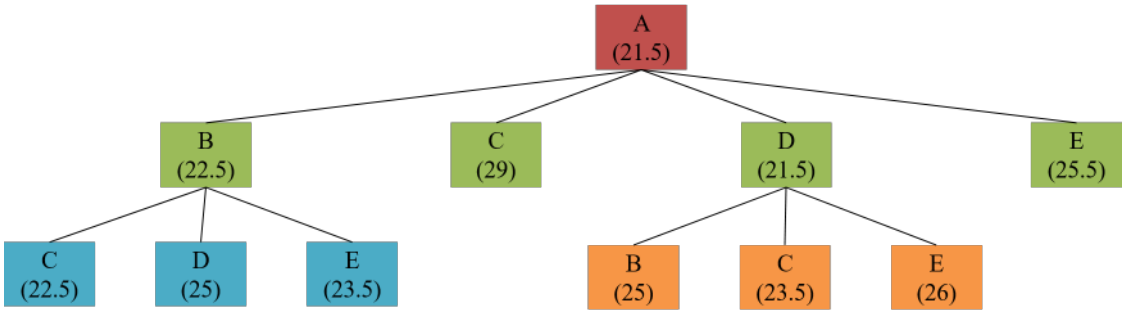


FIGURE 2.13 – Poursuite de l'investigation sur le sommet B.

Le processus se répète en évaluation- calcul de borne- et séparation - abandon provisoire de l'exploration - jusqu'à ce qu'une solution complète de poids inférieur à la borne courante soit trouvée. L'arborescence se termine pour notre exemple avec le cycle ABECDA de borne inférieure 25,5. Remarquez qu'il est identique à la solution ADCEBA, puisque le parcourir revient à faire le cycle gagnant en sens inverse.

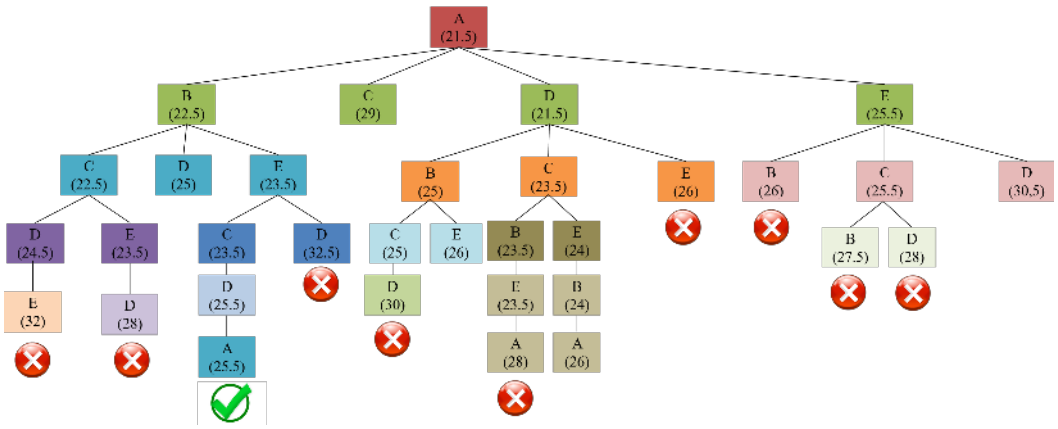


FIGURE 2.14 – Fin de l'arborescence du Branch and Bound.

Il reste à calculer le poids total du cycle. Ici, il est de $(7 + 5 + 4 + 8 + 2) = 26$. La solution est meilleure que celle trouvée par les méthodes heuristiques. De plus, on a la certitude qu'il s'agit de la solution optimale absolue. Mais ce qu'on gagne en exactitude, on le perd en temps de calcul car le *Branch and Bound*, quand il est mené à terme, est significativement plus lent que les méthodes approchées.

Dès lors, comment savoir si un algorithme est plus efficace qu'un autre dans la résolution du PVC ? Avec une heuristique, nous n'avons jamais la certitude, lorsqu'une réponse est fournie, qu'il s'agit de la réponse absolument optimale. Peut-être en existe-t-il une autre, encore meilleure. Pour départager les méthodes existantes, on peut simplement décréter que la méthode A est supérieure à la méthode B si A nécessite moins de temps ou moins de ressources pour résoudre chaque instance du problème. La difficulté est que ce temps de calcul dépend non seulement du nombre n de villes, mais également de leur configuration. Un autre critère est donc décidé : la méthode A est en avance sur la méthode B si, pour chaque grande valeur de n, le pire exemple de n-villes pour A prend moins de temps à résoudre que le pire exemple de n-villes pour B.

On pourrait dire que les algorithmes de résolution du PVC sont tous satisfaisants à leur façon. Dans le cadre de ce projet, nous avons choisi de nous attarder sur un de leurs semblables : l'algorithme génétique.

Deuxième partie

II

Des algorithmes bioinspirés

Chapitre 3

Optimisation, machines et problèmes difficiles

Précédemment, nous nous sommes intéressés au problème du voyageur de commerce, qui est un problème d'optimisation. Dès lors, voyons en quoi consiste l'optimisation et quel rôle jouent les algorithmes dans la résolution de ces problèmes.

3.1 Notion d'optimisation

Notre quotidien fourmille de questions d'optimisation : *quel est le trajet le plus rapide ? quel est le moyen de transport le plus économique ? quel ordonnancement des tâches fournit la meilleure productivité ?*

Optimiser, c'est chercher à améliorer les systèmes en visant la meilleure solution.

L'optimisation a pour objectif de trouver une solution optimale tout en prenant en compte un ensemble de contraintes et de variables inhérentes au problème. En pratique, l'optimisation mathématique consiste à trouver la meilleure solution à un problème qu'on a préalablement représenté par un modèle qui fait intervenir un ou plusieurs objectifs. Ces objectifs sont exprimés sous la forme d'une fonction mathématique, appelée souvent fonction objectif, qui peut être soumise à des contraintes. La solution optimale correspond à une valeur extrême, appelée aussi extremum – maximum ou minimum – de cette fonction objectif. Le but d'un problème d'optimisation est d'obtenir une solution maximisant ou minimisant une fonction objectif donnée. En économie par exemple, on s'en sert pour augmenter son bénéfice et diminuer son coût.

En mathématique, nous pratiquons l'optimisation lorsque nous recherchons les extrema d'une fonction. La réalité est bien plus complexe que ce problème mathématique abstrait. En économie, l'optimisation est souvent utilisée pour augmenter la rentabilité ou diminuer les coûts. Un problème d'optimisation peut donc vite s'avérer compliqué. En effet, non seulement, la représentation mathématique d'un problème n'est jamais totalement conforme à la réalité, mais en plus, les variables et les contraintes peuvent devenir très nombreuses selon la situation. Identifier le plus petit individu d'une famille de quatre personnes ne nécessite pas de grands moyens. Par contre, un problème de plus court chemin comme celui du PVC requiert l'outil informatique.

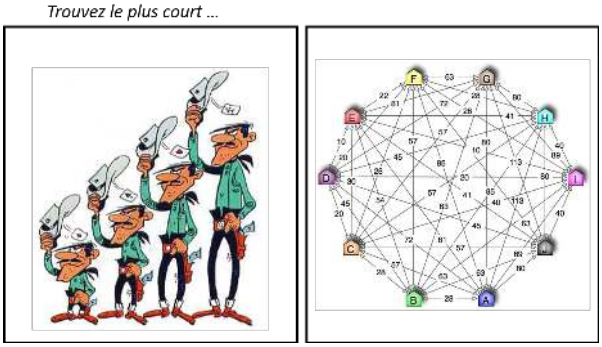


FIGURE 3.1 – Un problème d'optimisation très simple et un autre qui l'est beaucoup moins.

Dans certains cas, il n'est pas possible de trouver une solution analytique exacte : il faut résoudre « numériquement » le problème et recourir à un algorithme itératif pour s'approcher pas à pas de la solution. Et parfois, la réponse fournie par l'algorithme ne garantit pas le caractère absolument optimal de la solution.

3.2 Qu'est-ce qu'un algorithme ?

Sans qu'on s'en doute, les algorithmes ont toujours fait partie de notre quotidien, et ce, bien avant que le premier ordinateur ne voit le jour. Le mot « *algorithme* » vient du mathématicien musulman, Al Khwarizmi, qui a écrit au IX^e siècle le premier ouvrage systématique sur la résolution des équations linéaires et quadratiques. Les algorithmes existaient donc bien avant l'arrivée de l'ordinateur et étaient liés aux manipulations numériques. On peut définir un algorithme comme l'énoncé d'une suite d'opérations permettant de donner la réponse à un problème. Nous utilisons les algorithmes inconsciemment dans la vie de tous les jours. Lorsque nous ramenons de la célèbre enseigne suédoise jaune et bleue un meuble en kit, la procédure de montage s'apparente à la mise en oeuvre d'un algorithme.

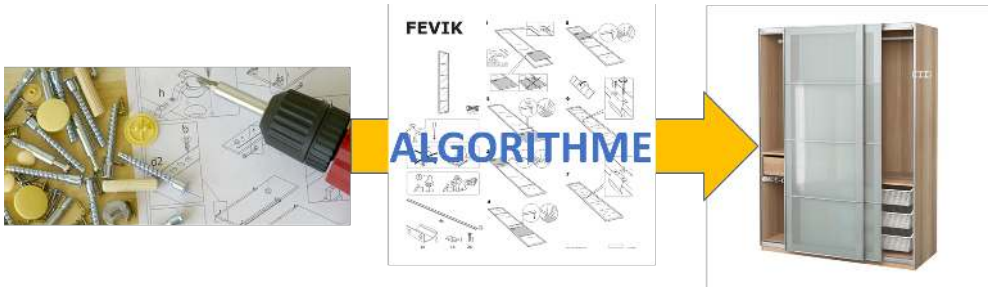


FIGURE 3.2 – Exemple d'algorithme de tous les jours.

3.2.1 Comment résoudre un problème par algorithme ?

Fournir un algorithme de résolution revient à donner une méthode générale permettant d'obtenir la réponse correcte pour chaque instance du problème.

Exemple : Résolution d'une équation du second degré ($ax^2 + bx + c = 0$) :

1. Entrer les valeurs des coefficients a, b, c.
2. Vérifier que l'équation est bien du second degré ($a \neq 0$?).
3. Calculer le réalisant ($\rho = b^2 - 4ac$).
4. Discuter :
   - Si $\rho < 0$, alors pas de solution réelle pour cette équation : $S = \emptyset$.
   - Si $\rho = 0$, alors on a une seule solution réelle double : $S = \{\frac{-b}{2a}\}$.
   - Si $\rho > 0$, alors on a deux solutions réelles : $S = \{\frac{-b+\sqrt{\rho}}{2a}, \frac{-b-\sqrt{\rho}}{2a}\}$.

Lorsqu'on résout un problème avec une machine, c'est analogue :

- Fournir une instance.
- Lancer l'exécution.
- Une fois la machine stoppée, on obtient la réponse correcte.

Attention : Il faut être capable d'interpréter le résultat lorsque la machine s'arrête : la solution est peut-être exactement la solution cherchée ou simplement une bonne solution.

Face à un problème à résoudre, un algorithme peut adopter diverses stratégies selon l'objectif à atteindre. Il peut être **déterministe** s'il accomplit un processus prédéfini pour chercher la solution. A l'opposé, un algorithme **non-déterministe** doit deviner à chaque étape la meilleure solution à l'aide d'heuristiques. Une **heuristique** est une méthode employée pour explorer intelligemment l'espace de solutions possibles, sans forcément examiner chacune des solutions potentielles. Donc, avec les mêmes données, un algorithme déterministe exécutera toujours la même suite d'opérations alors qu'un algorithme non-déterministe admettra de faire des choix durant son exécution.

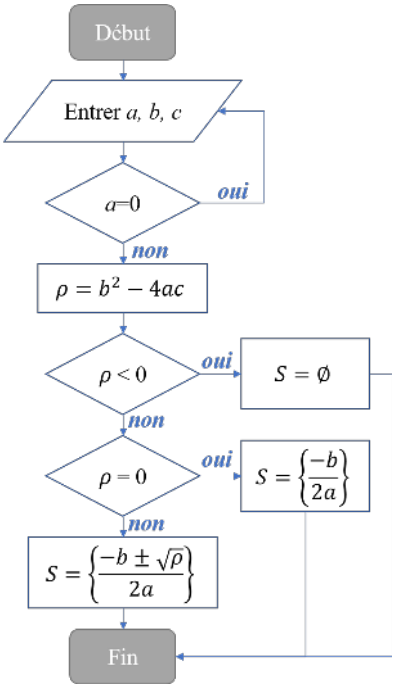


FIGURE 3.3 – Algorithme de résolution de l'équation du second degré.

Deux exécutions différentes d'un tel algorithme pourraient réaliser des choix différents et fournir éventuellement des solutions différentes. Une façon de réaliser ces choix est de les rendre aléatoires : on obtient ainsi les algorithmes **stochastiques**, ou probabilistes.

Dans ce travail, nous avons rencontré des algorithmes qu'on pouvait qualifier de gloutons ou d'heuristiques. Voici quelques mots d'explication sur cette terminologie :

- **L'algorithme glouton** : *greedy algorithm* en anglais, parfois appelé aussi algorithme *gourmand*, il fait le choix d'obtenir un optimum local, étape par étape, en espérant obtenir un résultat optimum global. Ce type d'algorithme présente un inconvénient de taille : comme son nom l'indique, il est très gourmand en temps de calcul.
- **L'algorithme heuristique** : algorithme qui fournit rapidement une solution réalisable, mais pas nécessairement optimale, pour un problème d'optimisation difficile. On privilégie donc la rapidité à l'exactitude de la réponse.

  Un algorithme heuristique est souvent spécifique au problème qu'il traite : il exploite certaines propriétés du problème pour orienter la recherche vers des zones susceptibles de contenir la solution. Certaines heuristiques reposent cependant sur des méthodes généralistes qui peuvent être déclinées dans beaucoup de problèmes. Dans ce cas, on parle de **méta-heuristiques**. Les méta-heuristiques, dont font partie les algorithmes génétiques, s'inspirent souvent de la nature.

FIGURE 3.4 – Exemples de méta-heuristiques.

3.3 Qu'est-ce que la complexité d'un problème ?

Pour effectuer un choix pertinent de l'algorithme de résolution, il est donc important de bien identifier à quelle catégorie le problème d'optimisation appartient : l'optimisation se fait-elle sous contrainte ? si oui, lesquelles ? s'agit-il d'un problème mono ou multi-objectif ? les données du problème sont-elles totalement déterminées ou y a-t-il une dimension stochastique ? ou plus fondamentalement encore, le problème est-il difficile ou non ? On peut classer les problèmes résolubles par algorithmes en trois familles :

- Problèmes faciles : résolubles par un algorithme efficace.
- Problèmes difficiles : résolubles par un algorithme mais par aucun algorithme efficace.
- Problèmes que l'on ne sait pas classer comme faciles ou difficiles : on ne connait ni algorithmes efficaces ni preuve de non-existence d'algorithmes efficaces.

On quantifie la complexité d'une question en mesurant sa réponse. Autrement dit, pour voir si un problème est compliqué, on s'intéresse aux algorithmes qui peuvent le résoudre.
La question de la complexité algorithmique nécessite une certaine étude. Il faut d'abord s'intéresser à l'efficacité des algorithmes ainsi qu'à la difficulté des problèmes traités par ces algorithmes. Lorsqu'un algorithme résout un problème, on peut se demander s'il existe un autre algorithme qui résout également le problème mais de manière plus efficace. Dans un premier temps, il est donc important de comparer, optimiser les performances des algorithmes. On cherche donc à objectiver une mesure de la performance des algorithmes, en temps et en espace, qui soit donc totalement indépendante de la machine physique qui l'exécute : on évite de se référer à un ordinateur en particulier qui pourrait être vite dépassé.

3.3.1 Complexité en temps

Un algorithme A étant une suite d'actions élémentaires à effectuer, on compte donc, indépendamment du processus, pour un problème à n entrées, les « pas » , c'est-à-dire les opérations élémentaires nécessaires pour terminer le calcul. On obtient d'abord $t\ (A,i)$, le temps d'exécution de l'algorithme A pour l'instance i. Ensuite, on regroupe les instances selon leur taille afin d'obtenir une mesure qui ne dépend pas d'une instance particulière. C'est ce qu'on nomme la **complexité en temps**, $c(A,n)$, dans le pire des cas pour les entrées de taille inférieure à n. C'est évidemment une fonction croissante avec n. Cette complexité en temps reste une notion très abstraite. Cependant, si on sait que l'ordinateur dont on dispose est capable d'effectuer une opération en un temps maximum K, il suffit de multiplier la fonction $c(A,n)$ par K pour obtenir le temps maximal requis.

3.3.2 Efficacité

Grâce à notre fonction de complexité en temps, on peut maintenant se questionner sur l'efficacité d'un algorithme. Un algorithme est dit efficace lorsque sa fonction de complexité en temps dans le pire des cas est bornée par un polynôme. Dans le cas contraire, il est considéré comme inefficace. Pour illustrer cette notion d'efficacité, prenons l'exemple suivant.

On fixe une durée de référence T (ici 1 heure). Aujourd'hui, compte tenu de la technologie dont on dispose, soit un ordinateur exécutant une opération élémentaire en K secondes, on peut traiter durant 1 heure, soit 3600 secondes, des entrées de taille maximum N. Ce qui se traduit par l'équation suivante :

$$K.c(A,N) = 3600$$

Si demain, notre machine va 10^6 fois plus vite, K devient $K' = \frac{K}{10^6}$. Alors, on pourra traiter en une heure un nombre d'entrées N' qui vérifiera :

$$K'.c(A,N') = 3600$$

On aura donc :

$$K'.c(A,N') = K.c(A,N)$$

Et de là, on en déduit que :

$$c(A,N') = 10^6.c(A,N)$$

Voyons ce que devient le nombre d'entrées traitées lorsque la fonction de complexité est polynomiale. Par exemple, prenons $c(A,N) = N^{12}$. L'équation précédente devient :

$$N'^{12} = 10^6.N^{12}$$

On peut donc exprimer N' en fonction de N :

$$N' = \sqrt{10}.N$$

L'amélioration technologique permettrait de traiter approximativement trois fois plus de données qu'à l'heure actuelle.

Prenons à présent une fonction de complexité exponentielle : $c(A,N) = 2^N$. Nous avons alors :

$$2^{N'} = 10^6.2^N$$

Isolons N' :

$$N' = 6\log_2(10) + N$$

On peut observer que l'amélioration technique n'est plus aussi significative que pour l'algorithme polynomial : ici, quel que soit N, l'amélioration ajoute de façon constante environ 20 entrées, ce qui, si le nombre N est élevé, peut s'avérer assez négligeable.
La distinction entre problème polynomial et problème exponentiel devient donc essentielle lorsqu'on considère les solutions pour des instances de grande taille. Le tableau suivant compare le temps nécessaire pour résoudre des problèmes caractérisés par des fonctions de complexité en temps types, polynomiales et exponentielles, en fonction du nombre d'entrées N . Pour se fixer les idées, nous imaginons utiliser un processeur capable de traiter 100 millions d'opérations par seconde. Chaque opération coûte donc 10^{-8}s. Si on prend par exemple la fonction de complexité N^5 et une entrée de taille 50, le temps t d'exécution est obtenu grâce au calcul suivant :

$$t = 50^5.10^{-8} = 3,125s$$

c(A,N)	Nombre d'entrées N					
	10	20	30	40	50	100
N	0,1 μs	0,2 μs	0,3 μs	0,4 μs	0,5 μs	1 μs
N^2	1 μs	4 μs	9 μs	16 μs	0,25 ms	0,1 ms
N^5	1 ms	0,032 s	0,243 s	1 s	3 s	100 s
2^N	10 μs	0,01 s	11 s	3 h	130 jours	4.10^{12} siècles
3^N	0,59 ms	35s	24 jours	3855 ans	$2,3.\ 10^6$ siècles	$1,6.10^{30}$ siècles

FIGURE 3.5 – Comparaison du temps de calcul de plusieurs fonctions de complexité en temps, polynomiales et exponentielles.

On remarque que les fonctions de complexité exponentielles possèdent une croissance très explosive par rapport à la croissance plus régulière des fonctions de complexité polynomiales.
A titre de comparaison, l'âge de la terre est de $4,54.10^7$ siècles. On se doute donc bien de l'impossibilité de traiter un problème de fonction de complexité exponentielle telle que la fonction 3^N lorsque l'instance est de grande taille.

Examinons maintenant avec le deuxième tableau les effets d'une amélioration technique sur les algorithmes ayant ces fonctions de complexité en temps. Le procédé de calcul est le suivant. Si on prend la fonction polynomiale N^5, on traite N_3 individus en 1 heure avec un ordinateur actuel. Si maintenant notre ordinateur est 100 fois plus rapide, le temps K' pour traiter une opération élémentaire est cent fois plus petit . Donc, on peut déduire le nombre maximum de données traitées de la façon suivante :

$$(N'_3)^5 = 10^2.(N_3)^5$$

ou encore :

$$N'_3 = \sqrt[5]{10^2}.N_3 \approx 2,5N_3$$

Calculons à présent, avec cette même amélioration technologique, le nombre de données traitées par un algorithme à fonction exponentielle telle que la fonction 2^N qui permet de traiter N_4 individus en une heure.

$$2^{N'_4} = 10^2.2^{N_4}$$

Ce qui revient à :

$$N'_4 = 2.\log_2 10 + N_4 \approx 9,97 + N_4$$

c(A,N)	Nombre de données max. traitables en 1 heure		
	Ordi. actuel	Ordi. 100x plus rapide	Ordi. 1000x plus rapide
N	N_1	$100N_1$	$1000N_1$
N^2	N_2	$10N_2$	$31,6N_2$
N^5	N_3	$2,5N_3$	$3,98N_3$
2^N	N_4	$6,64+N_4$	$9,97+N_4$
3^N	N_5	$4,19+N_5$	$6,29+N_5$

FIGURE 3.6 – Effet des améliorations technologiques sur le nombre de données traitées par plusieurs algorithmes en temps polynomial et en temps exponentiel.

Une amélioration technique, même 1000 fois plus rapide, reste négligeable pour un problème exponentiel car, pour la fonction de complexité 2^N par exemple, la taille de la plus grande instance traitable en une heure n'augmente que de 10 unités. Alors que cette amélioration technique permet presque de multiplier par 4 la taille de l'instance traitée pour la fonction N^5.
Ces tableaux montrent pourquoi les algorithmes en temps polynomial sont généralement vus comme bien plus souhaitables que les algorithmes en temps exponentiel.
Un problème est dit « intraitable » s'il est tellement difficile qu'aucun algorithme en temps polynomial ne peut le résoudre. Cette appellation doit être considérée comme une approximation grossière de son sens courant. La distinction entre algorithmes « efficaces » en temps polynomial et algorithmes « inefficaces » en temps exponentiel admet beaucoup d'exceptions quand les instances ont une taille limitée. Une exception est déjà observée dans le premier tableau : l'algorithme avec la fonction de complexité 2^N est plus rapide que l'algorithme avec la fonction de complexité N^5 lorsque N est inférieur à 20.

La plupart des algorithmes en temps exponentiel reposent sur des explorations exhaustives : les algorithmes gloutons peuvent en faire partie. Tandis que les algorithmes polynomiaux sont généralement obtenus grâce à une compréhension profonde de la structure d'un problème ou grâce au sacrifice de la perfection de la solution trouvée, au bénéficie du temps gagné.

Chapitre 4

Un peu de théorie de l'évolution

Nous avons vu que certains algorithmes étaient basés sur des méta-heuristiques inspirées de la nature. C'est le cas des algorithmes génétiques qui reposent sur le concept de sélection naturelle élaboré par Charles Darwin et sur les principes de la génétique de Gregor Mendel, tant au niveau du vocabulaire employé (population, individu, sélection, mutation,...) qu'au niveau de leur fonctionnement.

4.1 Charles Darwin (1809-1882)

Charles Darwin est un naturaliste anglais né en 1809 ayant révolutionné la biologie par ses travaux sur l'évolution des espèces. D'après lui, les vivants se modifient constamment et de nouvelles espèces apparaissent continuellement. L'évolution se fait de manière progressive. En 1859, il publie l'ouvrage « *De l'origine des espèces* » dans lequel il explique le principe d'évolution à l'aide de sa théorie de sélection naturelle.

4.1.1 Le Darwinisme

Cette théorie explique l'évolution des organismes au cours du temps. Chaque espèce est une collection d'individus présentant de petites différences plus ou moins favorables : les variations individuelles, qui sont héréditaires. Les vivants sont en concurrence et luttent pour survivre en raison de la limite des ressources mises à disposition par leur milieu. Ce sont donc les mieux adaptés à celui-ci qui vivent le plus longtemps - c'est la sélection naturelle - et se reproduisent le plus aisément - c'est la sélection sexuelle. Leurs gènes, forts, sont transmis aux générations suivantes.

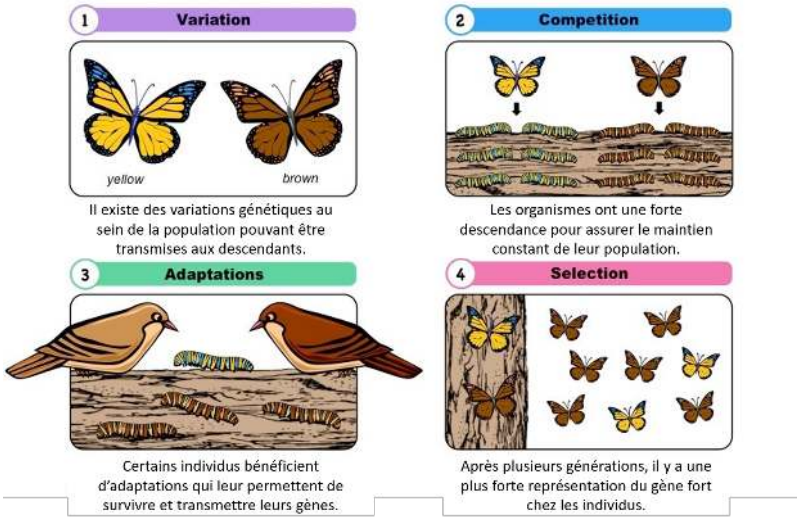


FIGURE 4.1 – Théorie de Darwin illustrée [19].

4.2 Gregor Mendel (1822-1884)

L'apparition d'espèces adaptées au milieu est aussi la conséquence de variations non dirigées du matériel génétique des espèces. Cette idée, on la doit à un moine botaniste autrichien : Johann Gregor Mendel. Mendel mena pendant 9 ans des expériences sur le pois (*Pisum sativum*) pour confirmer ses théories de l'hérédité. Mendel publia le résultat de ses études en 1866. À cette époque, on ignorait tout de la méiose et des chromosomes mais, en proposant l'existence d'unités héritables (qui seront appelées *gènes* en 1906 par le biologiste danois Wilhem Johannsen), Mendel fonda la génétique.

4.2.1 Lois de la génétique

De ses travaux, Mendel a tiré un certain nombre de principes connus aujourd'hui sous le nom de *lois de Mendel* et applicables à tout eucaryote ayant une méiose normale.

1. **Loi d'uniformité des hybrides de première génération** :

   La première loi de Mendel dit que « *si l'on croise deux races pures distinctes par un seul caractère, tous les descendants de la première génération, qui seront appelés des hybrides F1, sont identiques* ». Lorsque Mendel croise des pois à cosse jaune et des pois à cosse verte, en première génération, il n'obtient que des pois jaunes. Mendel qualifie ce trait de *dominant*, et celui qui ne se manifeste pas de *récessif*. Pour chacun des croisements de deux variétés ne différant que par un caractère, l'une des deux formes parentales se retrouve chez tous les hybrides en cas de dominance. En cas de codominance, c'est une forme intermédiaire qui apparaît.

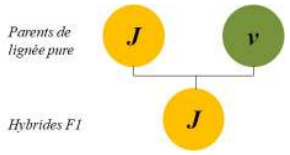


FIGURE 4.2 – Mendel utilise une lettre majuscule pour désigner le caractère dominant, une minuscule pour le caractère récessif correspondant.

2. **Loi de disjonction** :

   La deuxième loi dit que « *les gamètes ne possèdent qu'un seul allèle pour chacun des caractères* ». Lorsque Mendel croise entre eux les hybrides de première génération (F1), il apparaît dans les descendants (F2) des pois jaunes et des pois verts. Il comprend que pour faire apparaître un caractère récessif, il faut qu'au moins un des parents (F1) ait possédé le gène récessif pour pouvoir le transmettre. Par ailleurs, comme les parents manifestent tous le caractère dominant, il faut que ces parents aient aussi le gène dominant. Il en déduit que chaque parent (F1) possède, pour coder la couleur, deux gènes, l'un récessif et l'autre dominant. Bien que l'individu possède les deux gènes, seul le dominant s'exprime. Les individus F1 possèdent un *génotype* comportant à la fois le gène dominant et le gène récessif. Tous les pois F1 ont un génotype J/v : le gène « couleur jaune » J, dominant qui s'exprime, et le gène « couleur verte » v, récessif qui est dormant. La manifestation du génotype est appelée le *phénotype* et est dans le cas des pois F1, la couleur jaune J.

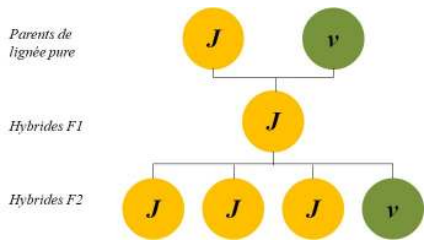


FIGURE 4.3 – Le caractère récessif v peut réapparaître à la deuxième génération.

3. **Loi de disjonction indépendante** :

   La troisième loi dit que « *la distribution des couples de gènes dans les gamètes se fait de façon indépendante.* » Mendel étudie conjointement deux caractères différents. Partant de deux lignées pures de phénotype pois jaune et lisse, pour la première et pois vert et ridé, pour la deuxième, les F1 obtenus sont tous de phénotype *JL* : l'allèle *Jaune* et l'allèle *Lisse* sont dominants vis-à-vis respectivement de l'allèle *vert* et de l'allèle *ridé*. En croisant les F1 entre eux, il obtient différents phénotypes dans des proportions caractéristiques : 9/16 de la F2 sont de phénotype jaune et lisse, 3/16 de type jaune et ridé, 3/16 vert et lisse et 1/16 vert et ridé. Cela ne peut s'expliquer que par une distribution indépendante des allèles caractérisant la couleur ou la forme lors de la formation des gamètes.

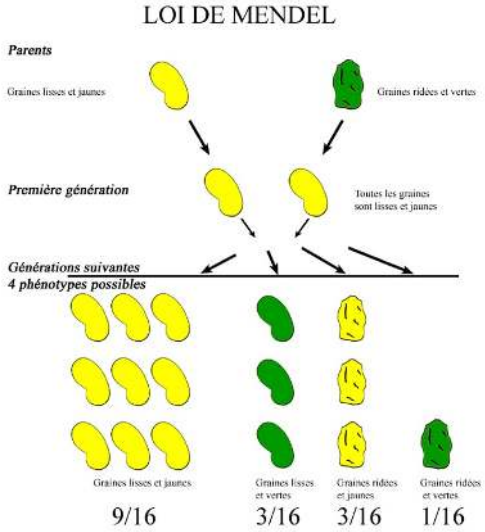


FIGURE 4.4 – Troisième loi de Mendel illustrée [6].

Les lois de Mendel constituent le fondement de la génétique du vingtième siècle. Plus tard, Thomas H. Morgan (1866-1945) complètera ces lois par sa théorie chromosomique de l'hérédité.

4.3 Théorie synthétique de l'évolution

Avec la génétique, Gregor Johan Mendel explique, le premier, la transmission des caractères innés, ce que n'avait pas pu démontrer Darwin. L'évolution des espèces est permise par des mutations au niveau des gènes. Un gène présentant un avantage sélectif pour un individu dans son environnement lui permettra de mieux survivre et mieux se reproduire. Il lèguera cet avantage à ses enfants. C'est donc un facteur héréditaire de l'évolution.

Le *néodarwinisme* concilie les idées mutationistes et la sélection naturelle en intégrant les mutations comme source de variation aléatoire, les mécanismes chromosomiques décrits par Morgan et la sélection naturelle.

En 1950, la théorie synthétique de l'évolution ajoute à ces idées les avancées dans le domaine de la paléontologie, de la biogéographie et de la génétique des populations. Bien que communément acceptée, cette théorie présente des aspects qui suscitent encore aujourd'hui la polémique dans la communauté scientifique.

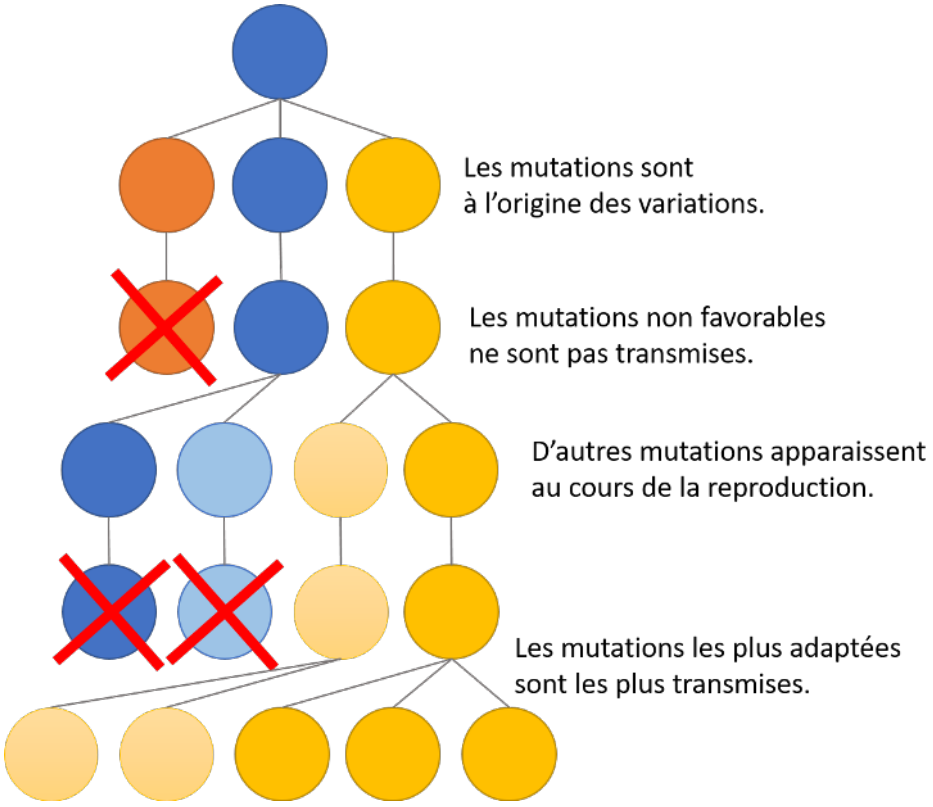


FIGURE 4.5 – Le néodarwinisme illustré.

Chapitre 5

Principes généraux des algorithmes génétiques

5.1 Naissance

Au début des années soixante, un chercheur de l'Université du Michigan, John Holland, s'intéresse aux processus d'adaptation qu'on trouve dans la nature et s'en inspire pour créer un algorithme de recherche basé sur les mécanismes de sélection naturelle et de la génétique. Il est en effet convaincu que la meilleure solution à un problème donné est contenue dans le pool génétique de la population au sein de l'espace de recherche, mais qu'elle ne peut s'exprimer car la combinaison génétique qui la définit est dispersée parmi les individus de la population. C'est en assurant le brassage génétique, par mutation spontanée et croisement sélectif, qu'on peut s'approcher de la solution optimale. Ainsi, Holland donne naissance aux premiers algorithmes génétiques et en 1975, sa publication « *Adaptation in Natural and Artificial System* » les fait connaître à la communauté scientifique. Dans les années quatre-vingt, les travaux de David Goldberg, autre chercheur américain en informatique, vont largement contribuer à les enrichir et à susciter une émulation parmi les informaticiens, qui y trouveront de vastes champs d'application.

5.2 Mécanisme

Les algorithmes génétiques sont des algorithmes itératifs – basés sur la répétition – stochastiques – en référence au hasard. Ils reposent sur les deux principes du néodarwinisme : la survie des individus les mieux adaptés - selon Charles Darwin - et la recombinaison génétique - selon Johann Gregor Mendel. Ils empruntent d'ailleurs leur terminologie au modèle évolutionniste : on parle de population, d'individu, de génération, d'adaptation, de sélection, de reproduction et de mutation.

5.2.1 Petit lexique comparé

Les objets qu'un algorithme génétique traite sont les suivants :

- **Individu** : en termes d'optimisation, un indvidu représente une solution possible au problème. En termes biologiques, un individu est représenté par un *chromosome*.

  La forme de l'individu doit respecter les contraintes du problème ; ces contraintes étant contenues dans le *génôme*. Un chromosome est une chaîne de *gènes*, qui peuvent posséder plusieurs *allèles*. L'algorithme repose sur un code qui attribue un symbole à chaque allèle possible. Cette attribution a lieu durant la phase de codage. Comme en biologie, on désigne par *génotype*, l'ensemble des gènes représentés par un chromosome (en langage binaire, cela pourrait correspondre à une séquence de bits) et par *phénotype*, l'ensemble des valeurs observables prises par chaque gène (en langage binaire, cela pourrait correspondre à la valeur décimale à laquelle a été attribuée la séquence de bits).

- **Population** : ensemble d'individus ou de solutions possibles traitées par l'algorithme.

  Cette population étant en constante évolution, on utilise également la notion de *génération*, désignant l'ensemble de la population à un moment donné du processus.

- **Fonction d'adaptation** : c'est une mesure de la qualité de l'individu par rapport au problème.

  La fonction d'adaptation est une manifestation du principe de *compétition* inhérent au néodarwinisme. Les individus possèdant la fonction d'adaptation la plus forte sont réutilisés pour participer à la reproduction ou à la génération suivante. C'est cette fonction, appelée aussi *fitness*, que l'algorithme cherche à optimiser.

- **Opérateurs de reproduction** : pour renouveler la population et assurer la diversité, l'algorithme emploie 2 opérateurs : le *croisement* et la *mutation*.

5.2.2 Forme canonique d'AG

Voici les grands principes de mise en œuvre d'un algorithme génétique standard :

- **Initialisation** :

— Création : création aléatoire des individus pour une population initiale.
— Codage : chaque individu est codé et les caractéristiques propres qui l'identifient sont traduites par la machine. Plusieurs formes de codage existent mais c'est souvent le codage binaire qui est utilisé.

- **Evaluation** : mesure de la faculté adaptative de l'individu, processus durant lequel nous pouvons directement apercevoir quels sont les meilleurs individus pour la reproduction ou l'individu susceptible d'être la solution du problème. L'opération attribue une valeur à chaque individu, nommée fonction d'adaptation ou *fitness*.

- **Exécution** :

— Sélection : les meilleurs individus sont sélectionnés pour la reproduction et forment un groupe de parents.
— Reproduction : l'opérateur de croisement ou *cross-over* permet de mélanger les codes génétiques des deux parents. L'opérateur de mutation permet de générer des modifications aléatoires dans le code génétique des enfants.
— Génération nouvelle : une nouvelle population se constitue, en remplaçant partiellement ou totalement les individus de la génération précédente par les nouveaux individus créés.

- **Retour à l'évaluation** : une réévaluation des individus permet de décider de poursuivre ou d'arrêter le processus.

5.2.3 Comparaison avec le néodarwinisme

Pour le néodarwinisme, le but est de trouver "*le meilleur individu*" qui sera le plus apte à survivre dans son environnement. Sa descendance héritera de ses gènes qui subiront éventuellement les mutations les plus bénéfiques grâce auxquelles l'espèce ne cessera de s'améliorer avec pour objectif d'atteindre l'espèce parfaite. Le meilleur individu représente pour les algorithmes génétiques la solution du problème, c-à-d l'individu qui, dans l'espace de recherche, est le plus apte à développer de nouvelles solutions en cas de difficulté et va donc devenir parent d'une nouvelle génération de solutions encore meilleures.

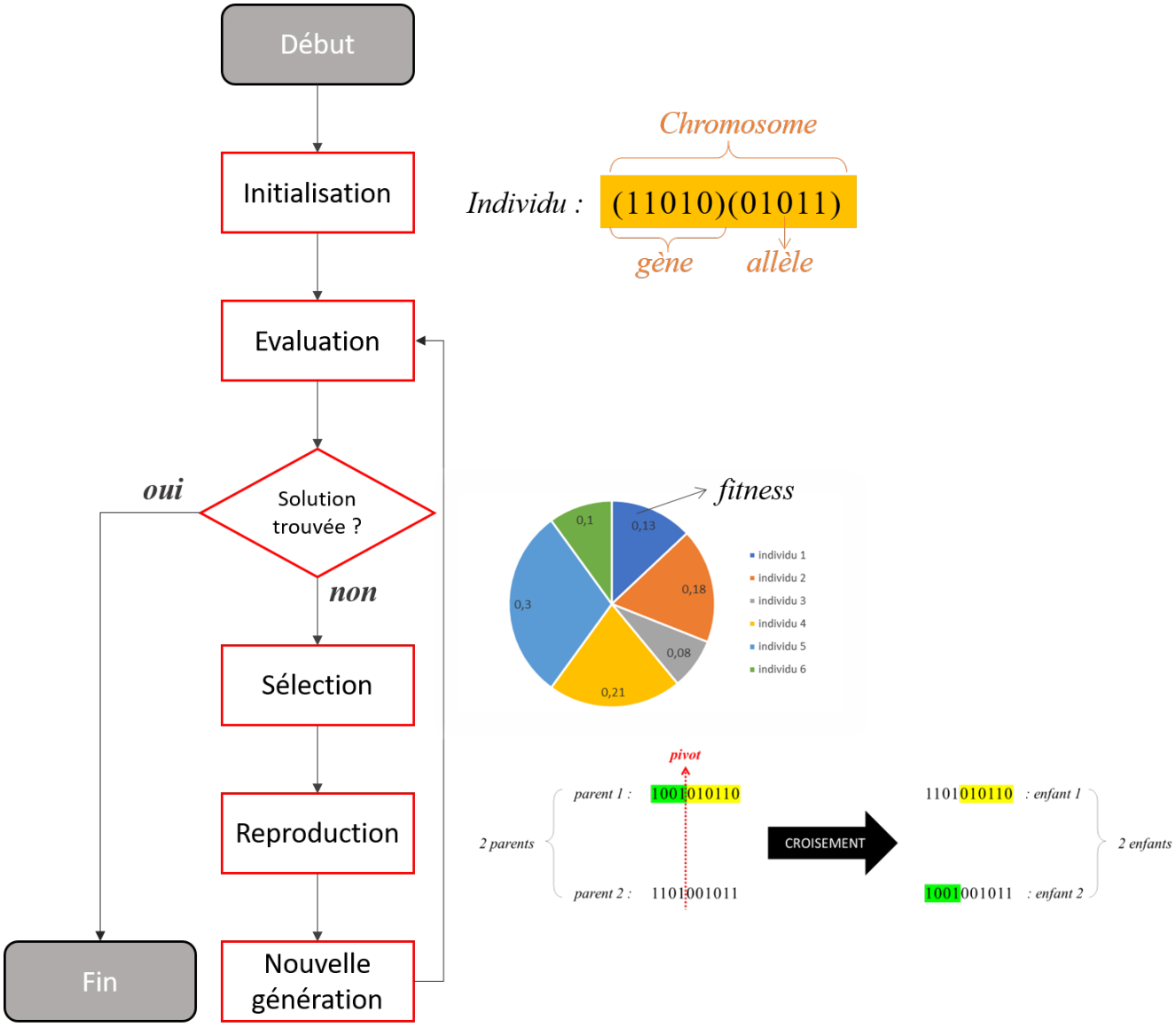


FIGURE 5.1 – Algorigramme génétique.

Chapitre 6

Fonctionnement détaillé

6.1 Codage des données

Le codage est une étape très importante. En effet, chaque paramètre définissant une solution potentielle de l'espace de recherche, soit un individu, doit être chiffré. Comme dans la nature, l'individu possède un ensemble de chromosomes, chaque chromosome est une suite de gènes et chaque gène peut prendre diverses valeurs possibles, chaque valeur étant un allèle. Il faut donc trouver une manière unique de coder chacun de ces allèles.

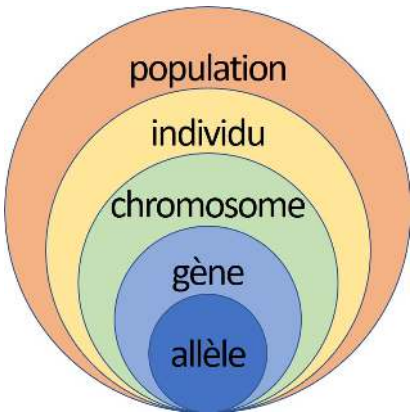


FIGURE 6.1 – Structure d'organisation à plusieurs niveaux.

Il existe trois méthodes pour coder les individus d'un algorithme génétique. Il est possible de passer de l'une à l'autre assez facilement.

- **Le codage binaire** :

  C'est le codage le plus fréquemment utilisé. L'alphabet binaire ne comporte que deux symboles - 0 et 1- . On les nomme couramment *bit* (de l'anglais *binary digit*, soit en français *chiffre binaire*). En faisant le parallèle avec la biologie, un bit correspond à un nucléotide et un allèle correspond à une séquence de bits. On parle même parfois de génotype quand on évoque la représentation binaire d'un individu et de phénotype pour désigner sa valeur réelle. Pour avoir une petite idée de ce qu'est le codage binaire, voici un exemple : on veut coder en binaire le nombre 26.

2^4	2^3	2^2	2^1	2^0
1	1	0	1	0

$$(2^4 \times 1) + (2^3 \times 1) + (2^2 \times 0) + (2^1 \times 1) + (2^0 \times 0) = 26$$

Dans le jargon biologiste, la séquence 11010 désigne le *génotype* et 26 est le *phénotype*. La formule générale pour décoder un chromosome x de longueur n présenté sous forme d'une séquence de bits a_i (a_i pouvant prendre soit la valeur 0 soit la valeur 1) est la suivante :

$$x = \sum_{i=0}^{n-1} a_i 2^i$$

Dans l'exemple précédent, nous avons codé un entier, si bien que chaque bit correspondait à une puissance naturelle de 2. Mais on pourrait imaginer devoir coder des nombres décimaux ou négatifs. Par exemple, si on voulait explorer l'espace de recherche caractérisé par l'intervalle $[-4, 4]$, en représentant les individus par une séquence de 8 bits, le décodage des chaînes de bits devrait se faire en plusieurs étapes. Tout d'abord, il faut envisager qu'une chaîne de 8 bits permet de coder 2^8, soit 256, valeurs différentes. L'échelle dans la réprésentation binaire n'est donc pas la même que dans la représentation réelle puisque notre intervalle initial $[-4, 4]$ a une longueur 8 et que le nouvel intervalle de codage est de longueur 255 et s'étend de 0 à 255. Le rapport d'échelle est le suivant :

$$\frac{8}{255} \approx 0,0314$$

Cela signifie qu'on se déplace dans l'intervalle $[-4, 4]$ par pas de $\frac{8}{255}$. Ensuite, l'origine de l'intervalle réel est -4 et pas 0. On en tient compte en effectuant une opération de translation. Voyons ceci avec un exemple. On souhaite décoder la séquence (10011010). On procède d'abord à la conversion dans le système décimal :

$$(2^7 \times 1) + (2^6 \times 0) + (2^5 \times 0) + (2^4 \times 1) + (2^3 \times 1) + (2^2 \times 0) + (2^1 \times 1) + (2^0 \times 0) = 154$$

On tient compte de l'échelle :

$$154 \times \frac{8}{255} \approx 4,831$$

On effectue la translation pour obtenir la valeur finale :

$$4,831 - 4 \approx 0,831$$

La séquence (10011010) désigne le chromosome approché à 10^{-3} par la valeur 0,831 dans notre espace de recherche.

La représentation génotypique binaire en chaînes de bits fut celle que John Holland employa à l'origine des AGs. Elle présente l'avantage d'une mise en place simple des opérateurs de croisement et de mutation. Cependant, ce type de codage atteint ses limites pour des problèmes d'optimisation dans des espaces de grande dimension.

- **Le codage réel** :

  Si les premiers algorithmes génétiques utilisaient un codage binaire, on s'aperçut très vite que le codage réel lui était préférable. Une des raisons est que le codage réel permet une plus grande marge de valeurs possibles. Dans l'exemple précédent, le codage sur une séquence de 8 bits ne permettait que 2^8, soit 256, valeurs différentes. Or, pour des problèmes où les solutions sont très rapprochées, une petite différence dans une valeur varie la performance de façon importante. Le codage réel est un système de codage par nombres représentés dans la base décimale (ex : 2 ;36 ;129,4) ou parfois même par lettres (ex : A ;F ;T).

  $$Ex : individu_1 = \{A, C, B, D\}$$

- **Le codage Gray** :

  Un autre problème du codage binaire est qu'il ne parvient pas à rendre compte de la proximité de certains nombres : deux nombres peuvent être réellement proches mais présenter des codes binaires assez éloignés. Pour mesurer cet éloignement entre deux séquences de bits, on utilise la *distance de Hamming* qui calcule le nombre de bits qui diffèrent. Ainsi, codés sur une séquence de quatre bits, les nombres 1, 2 et 3 sont respectivement codés par (0001), (0010) et (0011). La distance de Hamming entre 1 et 2 est de 2 puisque 2 bits les différencient alors que la distance de Hamming entre 1 et 3 n'est que de 1 ! Dans la représentation binaire, 3 semble plus proche de

Nombre	Codage binaire	Codage Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101

TABLE 6.1 – Comparaison de séquences binaires et Gray.

1 que 2. Pour résoudre ce problème, on a employé le codage Gray. Le codage Gray a une forme similaire au codage binaire. La principale différence avec ce dernier est qu'entre deux nombres proches, on assure une petite distance de Hamming. Ainsi, dans le codage Gray, la différence entre deux éléments consécutifs x et $x+1$ n'est que de un bit. Le tableau 6.1 présente le codage des 7 premiers naturels en binaire et en Gray.

6.2 Génération de la population initiale

La résolution d'un problème par algorithme génétique débute avec la génération de la population initiale. Les individus composant cette population initiale doivent être de la même espèce que la solution potentielle : ils doivent respecter les contraintes du problème. Lorsqu'on n'a aucune idée de la position de la solution optimale, on peut choisir aléatoirement cette population initiale non homogène afin d'explorer une grande partie de l'espace des solutions. Si on ne connait pas le problème à résoudre, il est cependant conseillé de répartir celle-ci sur tout le domaine de recherche. Quand on dispose d'une certaine connaissance du problème, on peut aussi faire le choix de directement désigner de « bons » individus, plutôt que de laisser le hasard opérer.
En ce qui concerne la taille de la population, elle n'est pas prédéfinie : c'est un paramètre laissé au libre choix de l'utilisateur. Mais il est bon de savoir qu'une petite population réduit le temps de calcul et peut donc évoluer plus rapidement qu'une grande population : un caractère favorable présent dans une petite population pourra être transmis plus rapidement que dans une grande. Toutefois, une population plus importante augmente la diversité génétique et permet donc une plus grande possibilité d'adaptation à divers environnements. Il faut donc trouver un juste milieu.
En tout premier lieu, on génère aléatoirement ou non quelques individus afin de composer la population initiale. La population va ensuite évoluer de génération en génération.

6.3 Fonction d'adaptation

La fonction d'adaptation ou *fitness* est la fonction qui détermine la capacité de chaque individu à survivre et à se reproduire. Plus le fitness d'un individu est élevé, plus l'individu est «fort» et donc, plus il a de chances d'être sélectionné pour engendrer une nouvelle population. La fonction d'adaptation est en général une fonction à valeur réelle qui a 0 comme valeur plancher. Le processus de l'algorithme génétique cherche alors à maximiser cette fonction d'adaption afin d'obtenir le meilleur individu, c-à-d la solution au problème. Le choix de la fonction doit donc rendre compte de la qualité réelle de l'individu. Si le problème consiste en la recherche d'un maximum local d'une fonction, la fonction d'adaptation peut être la fonction elle-même ; a contrario, si on cherche un minimum local d'une fonction, la fonction d'adaptation est une modification de la fonction à minimiser de sorte que l'algorithme puisse malgré tout chercher à maximiser le fitness pour trouver le minimum.
On voit donc que le choix de la fonction d'adaptation est crucial et dépend à la fois du problème à résoudre et de l'espace de solutions. De plus, la fonction d'apadation est coûteuse en temps de calcul car elle est appelée plusieurs fois durant la procédure itérative : avant la sélection pour évaluer les individus parents potentiels, et après le croisement et la mutation, pour évaluer les enfants.

6.4 Opérateurs de sélection

L'opérateur de sélection est un opérateur qui va choisir les individus de la population initiale qui se reproduiront entre eux (individus parents) afin de former de nouveaux individus qui génèreront la nouvelle population. Cet opérateur utilise le fitness comme critère de sélection et décide si les individus doivent survivre, se reproduire ou mourir.
Il existe différentes méthodes de sélection. Nous en présentons ici quelques unes.

6.4.1 Sélection par la roulette *Wheel*

Il s'agit de la méthode la plus fréquemment utilisée. On utilise le principe d'une roue de loterie biaisée.

A chaque individu correspond un secteur d'une roue (la roue de loterie). La superficie de ce secteur est proportionnelle à la capacité (*fitness*) de l'individu à s'adapter : plus un individu a une fonction fitness élevée plus il a de chance d'être sélectionné. A chaque secteur correspond une probabilité cumulée. Lorsque chaque individu s'est vu attribuer un secteur, on « fait tourner » la roue jusqu'à l'arrêt : on choisit aléatoirement un nombre entre 0 et 1. L'individu sélectionné est celui dont le secteur est pointé, c-à-d dont la plage de probabilité cumulée contient le nombre tiré au sort. Dans l'illustration suivante, la population est consituée de 6 individus auxquels un secteur de roue a été attribué. On visualise bien que l'individu 3 a moins de chance d'être sélectionné que l'individu 1. Si le hasard tire la valeur 0,5, c'est l'individu 4 qui est désigné puisque la plage de probabilité cumulée lui correspondant se situe entre 0,39 et 0,60.

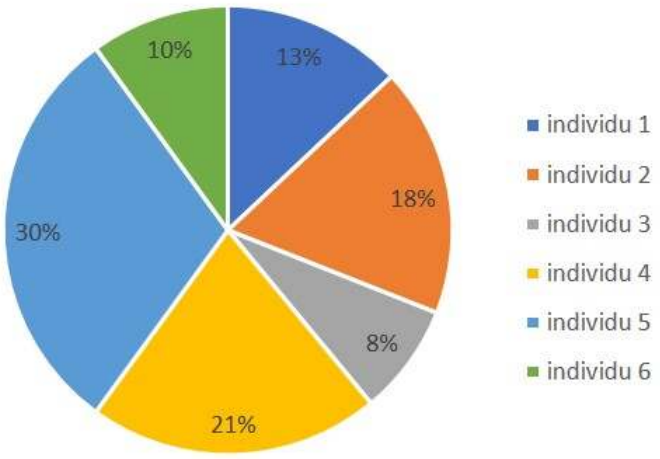


FIGURE 6.2 – Sélection par la roulette.

On reproduit alors cette opération autant de fois que nécessaire pour obtenir tous les individus qui font désormais partie des individus parents, un individu pouvant donc être sélectionné plusieurs fois.

Pour obtenir la probabilité p_i associée à chaque individu, on utilise la formule suivante :

$$p_i = \frac{f_i}{\Sigma_{j=1}^{Np} f_j}$$

avec f_i qui représente le fitness de l'individu i et N_p est la taille de la population.

Le nombre n_i de fois que l'individu i peut espérer être sélectionné, si la roulette tourne n fois, est donné par :

$$n_i = n \times p_i$$

Bien que ce soit la méthode de sélection la plus utilisée, elle comporte plusieurs inconvénients :

- La méthode présente une forte variance : les individus avec une fonction fitness moins élevée que d'autres peuvent tout de même être sélectionnés, ce qui va totalement à l'encontre de l'idée des AGs puisque le but de ces derniers est de sélectionner les meilleurs individus afin d'arriver au final à trouver un individu unique (le "*meilleur*").
- Il est possible aussi qu'un individu avec une fonction fitness très élevée (dont la probabilité d'être choisi est également élevée) soit sélectionné plusieurs fois, ce qui diminuerait fortement la diversité de la seconde population. Cette population ne saurait donc plus évoluer par la suite : on arriverait au phénomène de « convergence prématurée ». Le processus serait bloqué autour d'une solution finale, trouvée prématurément, qui n'est pas la solution optimale.

La méthode de la roulette existe également dans une forme appelée *sélection par le rang*. Elle consiste à faire une sélection en utilisant une roulette dont les secteurs sont proportionnels, non pas au fitness, mais au rang des individus (N pour le meilleur, 1 pour le moins bon, pour une population de taille N).

6.4.2 Sélection par tournoi

C'est la méthode donnant en général les meilleurs résultats. Elle possède un paramètre T, la taille du tournoi. Le principe est le suivant : on effectue un tirage de T individus dans la population et chaque tirage donne lieu à un combat. L'individu dont la fonction fitness est la plus élevée, donc celui qui a remporté «le combat» est désigné comme parent. On reproduit ce processus autant de fois que nécessite l'obtention des n nouveaux individus de la 2^e génération.

6.4.3 Sélection universelle stochastique

Cette méthode est très peu utilisée et possède une très faible variance. On partitionne un segment en tronçons. Chaque individu est associé à un tronçon dont la longueur est proportionnelle à son fitness. On détermine ensuite n points équidistants sur le segment, n étant le nombre d'individus devant composer la génération suivante. Les individus pointés par les points dans le segment seront ceux qui figureront dans la nouvelle génération. Il s'agit encore d'une méthode de sélection stochastique proportionnelle, mais cette fois, les individus sélectionnés simultanément.

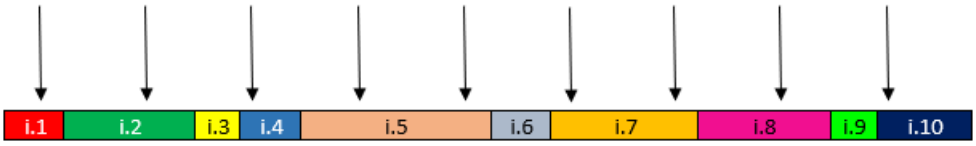


FIGURE 6.3 – Sélection universelle stochastique.

6.4.4 Sélection élitiste

C'est la seule méthode de sélection qui soit déterministe. Son principe est de classer par ordre croissant les individus en fonction de leur fitness. Ensuite, on sélectionne un ou plusieurs individus parmi les meilleurs de ce classement qui constitueront la population de parents. On génère ensuite par croisement les individus enfants nécessaires à la constitution de la génération suivante.
Les individus les moins performants sont totalement éliminés de la population, et le meilleur individu est toujours sélectionné – on dit que cette sélection est *élitiste*.

- Cette méthode présente une convergence fortement prématurée car on a une variance et une diversité presque nulles : les individus les moins bons n'ont aucune chance de survivre.
- Elle permet cependant de ne pas perdre des individus avec une fonction fitness élevée donc, solution possible du problème.

6.5 Opérateur de croisement

Commençons directement par une petite illustration... Considérons deux gènes A et B pouvant être améliorés par mutation. Il est peu probable que les deux gènes améliorés A' et B' apparaissent par mutation dans un même individu. Mais l'opérateur de croisement permettra de combiner rapidement A' et B' dans la descendance de deux parents portant chacun un des gènes mutants. L'opérateur de croisement va donc être capable de créer un individu encore mieux adapté. Il va permettre le brassage génétique de la population et l'application du principe d'hérédité du néodarwinisme. Le croisement est une propriété naturelle de l'ADN. C'est par analogie qu'ont été créés les opérateurs de croisement dans les AGs.

Il existe deux méthodes de croisement : simple ou double.

- Le croisement **simple** ou le croisement en un point consiste à fusionner les particularités de deux individus (les parents), à partir d'un pivot choisi au hasard, afin d'obtenir un ou deux enfants.

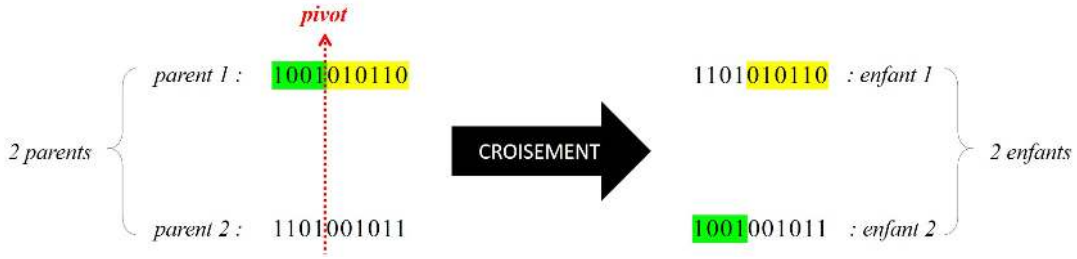


FIGURE 6.4 – Croisement simple.

- Le croisement double ou le croisement en deux points repose sur le même principe, sauf qu'il y a deux pivots. Celui-ci est généralement considéré comme plus efficace.

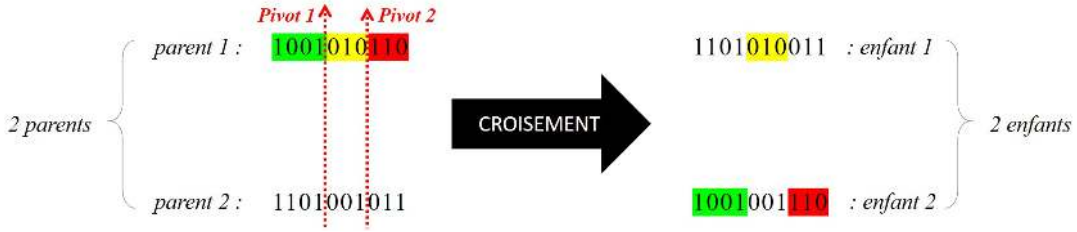


FIGURE 6.5 – Croisement double.

Reprenons...

— L'opérateur de croisement s'applique entre deux individus différents.

— Résultat : deux chromosomes sont formés à partir de la recombinaison des gènes des deux parents.

— Deux enfants sont « produits » pour la génération suivante.

6.6 Opérateur de mutation

Cet opérateur consiste à modifier un ou plusieurs allèles d'un gène avec une probabilité souvent assez faible (de l'ordre de 0,01 à 0,1). Il consiste donc à modifier aléatoirement les caractéristiques d'une solution. Il joue le rôle d'« élément perturbateur ».
Si l'on veut muter un individu qui est codé en binaire ou en Gray, il suffit de modifier la valeur d'un des bits comme le montre la figure suivante.

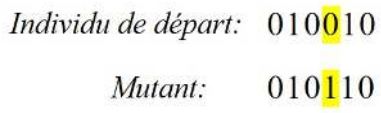


FIGURE 6.6 – Mutation aléatoire.

Si on veut muter en codage réel, il existe plusieurs manières de procéder :

* On peut intervertir deux gènes :

A,C,E,B,F,D → A,F,E,B,C,D

* On peut décaler un gène :

A,E,C,F,B,D → A,E,F,B, C,D

* On peut inverser l'ordre d'une partie des gènes :

B,F,C,E,A,D → C,F,B,E,A,D

L'opérateur de mutation possède plusieurs qualités :

- Il permet une plus grande diversité de solutions.
- Il permet d'éviter une dérive génétique, c-à-d que l'un ou plusieurs des gènes favorisés par le hasard ne se répandent tandis que les autres s'effacent de plus en plus. Dans ce cas défavorable, les allèles favorisés sont alors présents dans la plupart des solutions au même endroit.
- Il permet d'éviter les risques d'une convergence prématurée. Lorsqu'une convergence se fait, tous les individus deviennent pratiquement identiques mais ne sont pas de réelles solutions de l'algorithme génétique. Le croisement n'a donc plus aucun effet sur la génération suivante. C'est à ce moment-là que l'opérateur de mutation devient primordial car il va permettre de modifier un individu aléatoirement, ce qui va décoincer la situation.
- Il permet d'atteindre une propriété s'appelant l'*ergodicité*. Cette propriété garantit que chaque valeur de l'espace de recherche peut être atteinte. En effet, étant donné que l'opérateur de mutation intervient aléatoirement, on a la certitude que toutes les valeurs de l'espace de recherche peuvent apparaitre dans les diverses générations. On est donc certain de pouvoir trouver une réelle solution au problème.

6.7 Génération d'une nouvelle population

Une fois les nouveaux individus créés par croisements ou mutations, il faut sélectionner ceux qui vont participer à l'amélioration de la population et à la génération suivante. Le programmeur est libre de choisir les individus qu'il souhaite conserver : une nouvelle évaluation du fitness peut décider de remplacer la population initiale dans sa totalité, d'ajouter les nouveaux individus à la population initiale, ou encore, choisir de ne garder que les individus ayant la meilleure fonction d'adaptation. Donc, seuls les individus de la population initiale ou les enfants ayant les meilleures caractéristiques sont conservés pour la génération suivante et les individus qui ont de moins bonnes caractéristiques sont éliminés. Il n'est d'ailleurs pas recommandé de substituer en totalité les individus de la population initiale par les nouveaux individus créés, les enfants n'étant pas toujours meilleurs que leurs parents.

Une méthode relativement efficace consiste à insérer les nouveaux individus dans la population, à trier cette population agrandie selon l'évaluation de ses membres, et à ne conserver que les n meilleurs individus.

Pour connaître la proportion de nouveaux individus dans la population, on utilise le *Generation GAP* qui désigne le rapport entre le nombre de nouveaux individus et le nombre total d'individus de la population considérée.

6.8 Convergence et solution finale

De génération en génération, la fonction fitness des individus doit évoluer pour tendre vers un optimum : on cherche la convergence vers le meilleur individu. Tant que la fonction n'atteint pas l'optimum, le problème n'est pas résolu. La difficulté est que, bien entendu, la solution optimale n'est pas connue à l'avance. Il faut décider quand le processus itératif doit s'arrêter pour présenter la solution trouvée. Généralement, le critère d'arrêt choisi l'est en fonction du temps de calcul : les algorithmes génétiques n'ont pas pour vocation de trouver la solution optimale, mais un ensemble de "meilleures solutions possibles" en un temps limité. On peut également prendre en compte le critère de convergence. Un des signes de la convergence de l'algorithme est une augmentation progressive vers l'uniformité : de plus en plus, les individus composant la population se ressemblent et possèdent les mêmes caractéristiques. On dit qu'un gène a convergé quand 95% de la population possède la même valeur du gène. On dit aussi qu'une population a convergé lorsque les gènes de tous les individus ont eux-mêmes convergé.

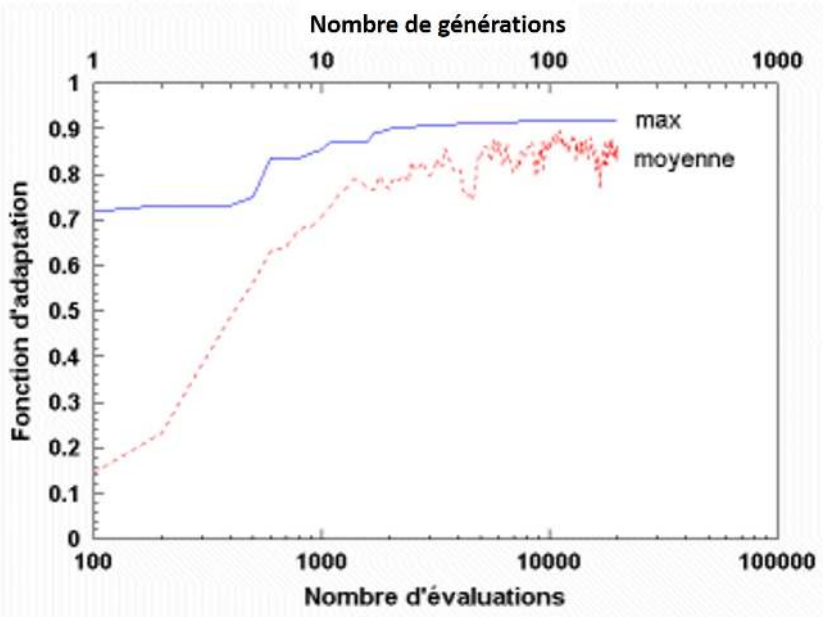


FIGURE 6.7 – Illustration de la convergence de la fonction d'adaptation moyenne [22].

On peut constater sur la figure 6.7 que l'amélioration de la population est très rapide au début et devient de plus en plus lente à mesure que le temps passe . A la fin, les légères fluctuations de la moyenne sont essentiellement dues aux mutations. On voit que cette valeur moyenne de la fonction d'adaptation a tendance à se rapprocher de celle de l'individu le plus adapté. Cela correspond à une uniformisation croissante de la population.

6.9 Deux exemples pour comprendre

Pour illustrer les propos précédents et mettre en oeuvre sur quelques générations les opérateurs étudiés, nous utiliserons des exemples inspirés de situations courantes en mathématique : la recherche d'extrema locaux. La simulation a eu lieu en utilisant la fonction générant du hasard dans un tableur.

6.9.1 Cas de la fonction à une variable

La première fonction est $f : [0, 16[\rightarrow R : x \xrightarrow{f} |x - 4| + |(x - 3)^2 - 1| - E\left(\frac{1}{x^2+1}\right)$.

Le graphe de la figure 6.8 montre la présence de 2 minima locaux, un en 2 et un en 4. Nul besoin de recourir à un algorithme génétique pour les trouver ! Cependant, nous allons mettre à l'épreuve l'outil, sur un espace de recherche découpant l'intervalle [0,16[en 32 segments. Nous choisissons de coder en séquence binaire sur 5 bits.

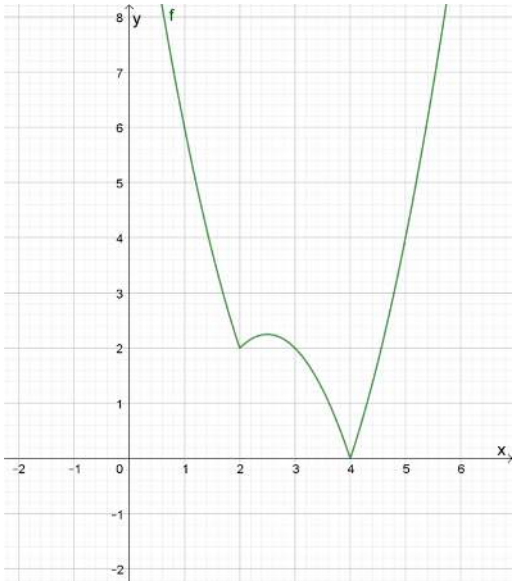


FIGURE 6.8 – Graphe cartésien de la fonction à une variable f.

Génération de la population initiale

Nous fixons la taille de la population à n=6. Nous générons de façon aléatoire 6 individus, représentés par des chromosones composés d'un gène unique codé par 5 bits. La fonction d'adaptation est choisie comme l'inverse de la fonction f, puisque nous sommes dans la recherche d'un minimum.

Numéro	**Séquence binaire**	**Valeur de x réelle**	**Valeur de $f(x)$**	**Fitness f_i**	**Probabilité p_i en %**	**Fonc. de répart. g_i en %**
Individu 1	00001	0,5	8,75	0,10	13	13
Individu 2	11111	15,5	166,75	0,01	1	14
Individu 3	00011	1,5	3,75	0,21	27	41
Individu 4	00101	2,5	2,25	0,31	39	80
Individu 5	00010	1	6	0,14	18	98
Individu 6	10100	10	54	0,02	2	100
			Total	**0,79**	**100**	

TABLE 6.2 – Population initiale.

Sélection de parents et croisement

La première génération d'individus présente un fitness moyen de l'ordre de 0,13. L'étape suivante consiste en la sélection des futurs parents. Nous avons choisis la méthode de la roulette qui va tourner six fois, afin de sélectionner 3 paires de parents, donnant par croisement 2 par 2 dans l'ordre de sélection 6 nouveaux individus. La roue de loterie est simulée ici par le tirage au sort d'un entier entre 0 et 100. Le point de croisement est également choisi aléatoirement dans la séquence.

N^o sorti	**Parents**	**Enfants**
85	00|010	00101
57	00|101	00010
92	0|0010	00011
21	0|0011	00010
41	001|01	00111
24	000|11	00001

TABLE 6.3 – Sélection des parents par la roulette et croisement.

Mutation aléatoire

En codage binaire, la mutation d'un gène correspond à l'inversion aléatoire de la valeur d'un bit. On simule de nouveau le hasard par un tirage au sort d'un nombre compris entre 0 et 100. Si ce nombre est inférieur à la probabilité de mutation convenue exprimée en pourcentage (nous avons ici fixé cette probabilité de mutation à 5%), la mutation s'opère : le bit 1 devient 0 et le bit 0 devient 1.

Avant mutation	Tirage au sort	Après mutation
00101	14 8 46 48 23	00101
00010	87 **4** 41 70 61	0**1**010
00011	27 56 65 93 20	00011
00010	48 87 70 44 11	00010
00111	34 16 95 **3** 52	001**0**1
00001	14 **4** 72 73 9	0**1**001

TABLE 6.4 – Mise en oeuvre de l'opérateur de mutation.

Réévaluation des nouveaux individus

Numéro	Séquence binaire	Valeur de x réelle	Valeur de $f(x)$	Fitness f_i	Probabilité p_i en %	Fonc. de répart. g_i en %
Individu 1	00101	2,5	2,25	0,31	20	20
Individu 2	01010	5	4	0,20	13	33
Individu 3	00011	1,5	3,75	0,21	14	47
Individu 4	00010	1	6	0,14	9	56
Individu 5	00101	2,5	2,25	0,31	20	76
Individu 6	01001	4,5	1,5	0,36	24	100
			Total	**1,53**	**100**	

TABLE 6.5 – Deuxième génération d'individus.

Le fitness moyen est à présent d'environ 0,26. Il s'est donc amélioré. Par ailleurs, on constate que les individus ont une qualité qui se stabilise. En effet, les probabilités associées ne font plus apparaître des valeurs extrêmes comme le 1% du tableau précédent.

Numéro	Séquence binaire	Valeur de x réelle	Valeur de $f(x)$	Fitness f_i	Probabilité p_i en %	Fonc. de répart. g_i en %
Individu 1	00010	1	6	0,14	7	7
Individu 2	01110	7	18	0,05	2	9
Individu 3	01000	4	0	1	50	59
Individu 4	00010	1	6	0,14	7	66
Individu 5	00110	3	2	0,33	17	83
Individu 6	00110	3	2	0,33	17	100
			Total	**2,01**	**100**	

TABLE 6.6 – Cinquième génération.

Après cinq générations, on constate que le fitness moyen ($\approx 0,34$) a encore augmenté. On voit aussi que la solution optimale *4* compte parmi les individus de cette génération. Cependant, le programme l'ignore. Ce n'est que le critère d'arrêt qui pourrait présenter cette valeur comme solution finale. On voit également que la convergence vers un état stable n'est pas significative. Cette permanence de la variabilité est due à la fois à l'opérateur de mutation, dont le rôle est d'assurer la diversité et à la fois à la nature du codage binaire employé [1].

1. Cfr. Codage des données en page 33

6.9.2 Cas de la fonction à deux variables

La seconde fonction étudiée est $f : R \times R \to R : (x, y) \xmapsto{f} 15x^2y^2 \exp(-x^2 - y^2)$. Cette fois, le graphe cartésien fait apparaître quatre maxima locaux. Nous nous attacherons à localiser ces maxima sur l'espace de recherche $[-2, 2] \times [-2, 2]$. Comme c'est un maximum qui est recherché, la fonction elle-même sera choisie comme fonction d'adaptation.

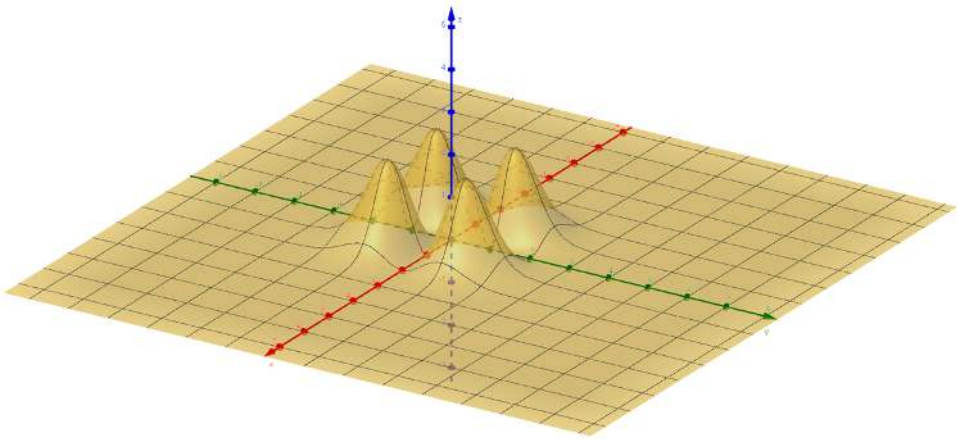


FIGURE 6.9 – Graphe cartésien de la fonction à deux variables *f*.

Codage des premiers individus

De nouveau, le codage binaire sera employé. Mais à la différence de l'exemple précédent, chaque individu présente 2 gènes, un pour la valeur de l'abscisse x et un pour la valeur de l'ordonnée y. Nous décidons que chacun de ces gènes soit codé par une séquence de 6 bits. L'intervalle réel $[-2, 2]$ contenant chaque gène a une longueur 4. L'intervalle de codage est quant à lui de longueur $2^6 - 1$, c-à-d 63, et s'étend de 0 à 63. Le rapport d'échelle est le suivant :

$$\frac{4}{63} \approx 0,0635$$

Par ailleurs, l'origine de l'intervalle réel étant -2 et non 0, on n'oublie pas d'effectuer la translation nécessaire. Voyons avec le premier individu généré aléatoirement (111010)(010001). La première séquence binaire (111010) désigne l'absisse x et la seconde (010001) l'ordonnée y.
On convertit d'abord la séquence du gène x (111010) dans le système décimal :

$$(2^5 \times 1) + (2^4 \times 1) + (2^3 \times 1) + (2^2 \times 0) + (2^1 \times 1) + (2^0 \times 0) = 58$$

On tient compte de l'échelle :

$$58 \times \frac{4}{63} \approx 3,68$$

On effectue la translation pour obtenir la valeur finale :

$$3,68 - 2 \approx 1,68$$

On recommence pour le gène y (010001).

$$(2^5 \times 0) + (2^4 \times 1) + (2^3 \times 0) + (2^2 \times 0) + (2^1 \times 0) + (2^0 \times 1) = 17$$

$$17 \times \frac{4}{63} \approx 1,08$$

$$1,08 - 2 \approx -0,92$$

La double séquence (111010)(010001) correspond à l'individu (1,68 ;-0,92) dans notre espace de recherche.

Numéro	**Séquence binaire**	**Valeur de x réelle**	**Valeur de y réelle**	**Fitness** f_i
Individu 1	(111010)(010001)	1,68	-0,92	261,56
Individu 2	(111101)(100010)	1,87	0,16	43,17
Individu 3	(101110)(100001)	0,92	0,10	0,27
Individu 4	(111010)(0111110)	1,68	-0,10	6,47
Individu 5	(110111)(010111)	1,49	-0,54	67,34
Individu 6	(010101)(110011)	-0,67	1,24	3,44
		Fitness moyen		**63,71**

TABLE 6.7 – Evaluation de la population initiale.

Sélection et croisement

Notre choix se porte sur la méthode élitiste. Nous choisissons les trois individus les plus forts, c-à-d possédant le meilleur fitness : les individus 1, 2 et 5. Ensuite, nous opérons des croisements en locus aléatoire, sur chaque gène.

Numéro	**Parents**	**Enfants**
Individu 1	(1110|10)(010|001)	(111011)(010111)
Individu 5	(1101|11)(010|111)	(110110)(010001)
Individu 1	(11|1010)(0100|01)	(111101)(010010)
Individu 2	(11|1101)(1000|10)	(111010)(100001)
Individu 5	(11011|1)(0|10111)	(110111)(000010)
Individu 2	(11110|1)(1|00010)	(111101)(110111)

TABLE 6.8 – Sélection élitiste des parents et croisement.

Evaluation de la nouvelle génération

Numéro	**Séquence binaire**	**Valeur de x réelle**	**Valeur de y réelle**	**Fitness** f_i
Individu 1	(111011)(010111)	1,75	-0,54	209,89
Individu 2	(110110)(010001)	1,43	-0,92	85,56
Individu 3	(111101)(010010)	1,87	-0,86	619,15
Individu 4	(111010)(100001)	1,68	0,10	6,47
Individu 5	(110111)(000010)	1,49	-1,87	32,51
Individu 6	(111101)(110111)	1,87	1,49	422,17
		Fitness moyen		**229,29**

TABLE 6.9 – Evaluation de la deuxième génération.

Le tableau montre une amélioration significative du fitness et une réduction des écarts entre les fitness individuels. Cependant, le processus n'est pas terminé et nécéssitera encore plusieurs itérations avant de converger.
La fonction étudiée possède en réalité quatre maxima localisés dans l'espace de recherche aux points $(1,1); (-1,1); (1,-1)$ et $(-1,-1)$.

Chapitre 7

De nombreuses applications

Très vite, les algorithmes génétiques sont apparus comme un bon compromis pour résoudre un problème lorsqu'on préfère avoir une solution relativement bonne rapidemment plutôt qu'avoir la solution optimale en une durée extrêmement longue. C'est donc pour cela que les algorithmes génétiques se sont révélés utiles dans de très nombreux domaines. Les secteurs dans lesquels ils interviennent sont principalement ceux de l'économie et de la finance, où ils sont utilisés comme outils d'optimisation et de résolution numérique, ou dans la robotique où ils peuvent représenter l'apprentissage de la machine.

7.1 Utilisation ludique des AGs

Voici une utilisation très simple des AGs. Un jeu nommé *BoxCar2D*[1] illustre bien les possibilités des algorithmes génétiques. Le but est d'apprendre à l'ordinateur à construire un véhicule en utilisant des formes géométriques quelconques, des liaisons, des roues, ..., qui puisse parcourir la plus longue distance sur le circuit accidenté sans subir de dégât.

FIGURE 7.1 – Génération aléatoire d'une voiture [4].

Le programme crée une première voiture en assemblant au hasard les formes géométriques et génère ensuite de nouvelles populations de voitures selon les principes des AGs. Au fur et à mesure des générations, les voitures commencent à prendre forme et parcourent de plus longues distances.

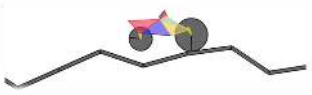


FIGURE 7.2 – Evolution de la voiture initiale [4].

Si on laisse le programme générer des voitures pendant assez longtemps, on obtient une voiture équilibrée qui "tient la route".

FIGURE 7.3 – Voiture finale au bout de plusieurs générations [4].

1. Jeu disponible en suivant le lien http ://boxcar2d.com/

7.2 Les algorithmes génétiques en robotique

Un algorithme génétique peut également enseigner un comportement intelligent à un robot. Le comportement du robot est déterminé par des circuits dont les sorties décident de son action (tourner, avancer, accélérer, ralentir) en fonction de ce qu'il perçoit (présence de murs à gauche,en face ou à droite). En robotique, l'algorithme génétique peut consister en un apprentissage non-supervisé, au sens où on n'indiquera pas au robot quelle décision prendre : l'algorithme génétique ne contrôle donc pas le robot, il permet de définir les meilleurs paramètres pour une situation donnée. Les individus sont composés des paramètres du robot qui peuvent être par exemple des coefficients dans une équation, des pentes, des accélérations de moteur, etc... La fonction d'adaptation, quant à elle, peut correspondre à la vitesse maximale atteinte, à l'autonomie du robot, à la précision d'un déplacement, etc... Par exemple, sur un robot slalomeur, l'AG cherchera les paramètres d'assemblage qui assurent un déplacement le plus précis et le plus rapide.

7.3 Un outil précieux pour les économistes

Pour faire face à une complexité croissante de l'économie, les économistes se sont naturellement tournés vers les AGs. Certains problèmes compliqués nécessitent en effet une résolution numérique. On peut donc retrouver les AGs dans la modélisation de croissance optimale ou comme outil de résolution d'équations difficiles telles que celles qui amènent à l'équilibre de Nash en théorie des jeux.

On retrouve également de plus en plus les AGs en économétrie. L'économétrie est un outil d'analyse qui permet de vérifier, sur la base d'observations statistiques, l'existence de relations entre des phénomènes économiques et de les mesurer. Par exemple, on s'intéressera au lien entre les dépenses publicitaires et les ventes réalisées. Il s'agit donc d'identifier des relations entre différentes variables et de rechercher la corrélation entre elles. Les AGs peuvent apporter une forme fonctionnelle à la relation, même si ces variables sont nombreuses, ou si la régression est non-linéaire.

Dans un autre domaine de l'économétrie, on utilise les AGs pour modéliser les séries temporelles. Les séries temporelles constituent une branche de l'économétrie dont l'objet est l'étude des variables au cours du temps. Parmi ses principaux objectifs figurent la détermination de tendances au sein de ces séries ainsi que la stabilité des valeurs (et de leur variation) au cours du temps. On s'intéresse aux variations d'une même variable au cours du temps, afin de pouvoir en comprendre la dynamique et éventuellement en prédire l'évolution. La variable étudiée peut être le PIB d'un pays, l'inflation, les exportations, les ventes d'une entreprise donnée, son nombre d'employés, le prix d'une option d'achat ou de vente, le cours d'une action, la pluviosité, le nombre de jours de soleil par an, le nombre de votants, la taille moyenne des habitants, leur âge... : tout ce qui est chiffrable et varie en fonction du temps. Associée à une banque de données solide, la modélisation produite par les AGs devient alors outil de prévision et de décision dans de multiples domaines : la finance, la bourse, la gestion des ressources humaines, la météorologie, la politique, ...

Dans le même ordre d'idée, les AGs sont utilisés dans la prévision d'événements rares : prévision d'utilisation frauduleuse de cartes de crédit à partir d'un historique d'achats, prévision de comportements inhabituels sur un marché financier, prévision de casses ou d'échecs de fonctionnement d'équipements téléphoniques ou électroniques à partir d'une série d'alarmes, etc. Le principe est le suivant : à partir d'une série historique de données, l'algorithme recherche et construit des règles de prévision permettant de savoir si un événement rare est susceptible d'apparaître dans un futur proche.

7.4 Applications industrielles des AGs

7.4.1 Application des AGs au sein de l'entreprise Motorola

L'entreprise Motorola, spécialisée dans l'électronique et les télécommunications, a utilisé les AGs dans le but de tester les applications informatiques. En effet, lorsqu'on modifie une application en changeant un de ses paramètres, il faut la soumettre à une batterie de tests afin de voir si elle fonctionne toujours et si les modifications apportées n'ont pas eu d'influences négatives sur le reste de l'application. Pour réaliser cela, la méthode classique est de définir manuellement des plans de test permettant un passage dans toutes les fonctions de l'application. Mais ce type de test nécessite un important travail humain et une importante perte de temps. Motorola a donc utilisé les AGs afin d'automatiser cette phase de définition de plans de tests et en réduire le temps d'exécution. Dans ces AGs, chaque individu correspond à un résultat d'exécution d'un programme et la fonction d'adaptation correspond à son aptitude à passer dans un maximum de parties du code de l'application. Les algorithmes génétiques ont permis l'amélioration de ces programmes de test.

7.4.2 Application des AGs dans la recherche de profils d'aile d'avion

Deux scientifiques spécialisés dans la mécanique des fluides, Navier et Stokes ont réussi à trouver des équations permettant de décrire le comportement de particules dans un gaz peu dense : c'est ce qui permet aux avions de voler. Ces équations, extrêmement complexes, peuvent être utilisées pour déterminer la portance et la traînée d'un profil d'aile d'avion donné mais ce qu'on souhaiterait typiquement obtenir, c'est l'inverse : quel profil d'aile d'avion donnera la portance maximale pour la traînée minimale ? Le seul moyen est de dessiner un profil d'aile et ensuite d'utiliser les équations de Navier-Stokes afin de déterminer la portance et la traînée du profil dessiné. Si le résultat n'est pas satisfaisant, il faut redessiner un nouveau profil d'aile et tester à nouveaux les équations. On peut donc comprendre que cela prenne beaucoup de temps. Là encore, c'est un algorithme génétique qui, à partir de paramètres définis, génère des profils d'aile et les évaluent selon le processus itératif décrit précédemment. Les profils qui offriront la meilleure portance pour la plus petite traînée, seront utilisés pour créer les générations suivantes. Ainsi, on peut arriver à trouver un profil d'aile convaincant en des temps restant raisonnables.

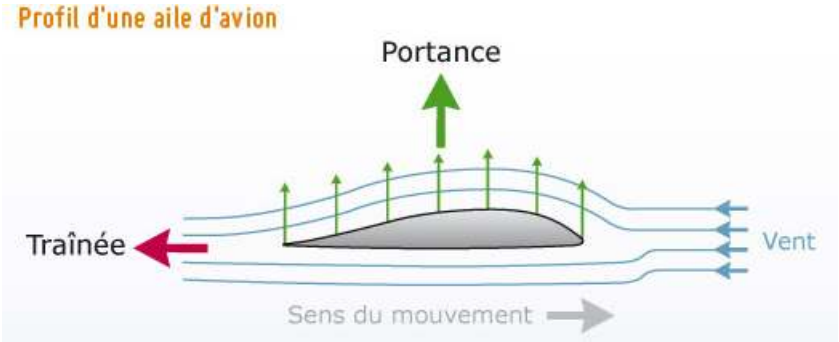


FIGURE 7.4 – Portance et traînée autour d'un profil d'aile d'avion [18].

7.4.3 Absorption optimisée des rayonnements solaires dans les cellules photovoltaïques

Des chercheurs du Laboratoire de Physique du Solide de l'Université de Namur ont développé un algorithme génétique [23] pour obtenir une absorption optimale du rayonnement solaire dans les couches minces de silicium cristallin pour des applications dans les cellules photovoltaïques. La majorité des cellules photovoltaïques utilise des technologies basées sur le silicium. Pour assurer un rendement optimal lors d'utilisation de couches minces (40 microns) de silicium cristallin, il faut développer des stratégies pour maintenir une forte absorption de radiation solaire dans ces couches. C'est possible en créant une structure du film mince capable de piéger la lumière, et ainsi compenser l'épaisseur réduite. La cellule photovoltaïque développée présente un réseau périodique de trous en forme de pyramides inversées. Le rôle de ces trous est de faciliter la pénétration de la lumière dans le silicium et de faire en sorte qu'elle y reste plus longtemps. Il y a également des couches « anti-reflet » à l'avant de la structure ainsi que des couches devant jouer un rôle de « miroir » à l'arrière. Cette fonction « miroir » permet à la lumière de repasser dans la couche de silicium, ce qui augmente encore la probabilité d'absorber la lumière dans cette couche. Les paramètres géométriques des pyramides inversées ainsi que l'épaisseur des différentes couches doivent être ajustés afin de maximiser l'absorption des radiations solaires dans le silicium cristallin. Une exploration systématique de toutes les combinaisons de paramètres serait bien trop chronophage. Cette situation justifie parfaitement l'emploi d'un algorithme génétique. Ici, son rôle est de déterminer les épaisseurs optimales pour ces différentes couches ainsi que les dimensions optimales pour le réseau périodique de trous. Dans cette recherche, la stratégie évolutive de l'AG a permis de trouver une structure capable d'absorber plus de 90% de la lumière incidente.

7.5 Et bien d'autres applications encore ...

Ce serait vain de vouloir énumérer de façon exhaustive toutes les applications des AGs tant elles sont nombreuses et variées : optimisation de fonctions numériques difficiles, traitement d'image (alignement de photos satellites, reconnaissance de suspects...), optimisation de réseaux (câbles, fibres optiques, mais aussi eau, gaz...), optimisation d'emplois du temps, optimisation de design, contrôle de systèmes industriels, ... Les AGs peuvent être utilisés pour contrôler un système évoluant dans le temps (chaîne de production, centrale nucléaire...) car la population peut s'adapter à des conditions changeantes. Ils peuvent aussi servir à déterminer la configuration d'énergie minimale d'une molécule ou à modéliser le comportement animal. Les AGs trouvent leur utilité dans des applications professionnelles comme dans la vie de tous les jours : ainsi, on peut utiliser un algorithme génétique pour résoudre un sudoku. Un problème dont la résolution est typiquement attachée aux algorithmes génétiques est le problème du voyageur de commerce. C'est pourquoi il est la ligne de conduite de ce travail. Dans la dernière partie de ce travail, nous présentons l'algorithme génétique que nous avons imaginé pour résoudre ce problème.

Troisième partie

III

De la théorie à la pratique : résolution du Problème du Voyageur de Commerce par Algorithme Génétique

Chapitre 8

Partie exécutive du programme

Pour résoudre notre PVC, nous avons décidé d'utiliser le langage Python, pour sa facilité d'acquisition, tout en optant pour la programmation orientée objet afin de mieux structurer notre programme. Par conséquent, la compréhension de cette partie du dossier nécessite quelques notions de langage de programmation Python.

Nous avons pris l'initiative de diviser notre programme en plusieurs parties, chacune représentée par un fichier bien défini. Dans chaque fichier, plusieurs classes ont été construites et plusieurs fonctions définies. Nous allons débuter nos explications en détaillant la composition de chacun de ces fichiers.

8.1 Fichier *Parcours*

Dans ce fichier, nous créons une classe *parcours* dans laquelle nous définissons une fonction *villes*. Cette fonction a pour but de retourner une liste de toutes ces villes accompagnées de leur coordonnées géographiques. Le PVC traité est donc associé à une base de données sous forme de liste. Chaque élément de la liste est lui-même une liste à 2 éléments : le nom de ville et le couple latitude et longitude. Nous avons choisi d'illustrer notre PVC en utilisant le parcours de l'équipe de Dantzig (cfr. point 9.1), mais cette liste pourrait très bien être la liste des capitales européennes, la liste des cercles d'étudiants d'une université ou encore la liste des musées à voir dans une ville.

```
class parcours:

    def villes(self):
        liste=[['Helena au Montana', (46.5883707, -112.02450540000001)], ['Des Mo
        """
        liste = [[ 'Clermont-Ferrand',(3.002556 , 45.846117)],['Bordeaux',(-0.644
        """
        return liste
```

FIGURE 8.1 – Introduction des lieux à visiter.

8.2 Fichier *Génération*

Dans ce fichier, on trouve une fonction *__init__* qui crée une liste nommée *circuit_base*. Le but est de nettoyer la liste obtenue précédemment grâce à la fonction *villes* du fichier *parcours* des noms de ville et de ne conserver que les couples de coordonnées. Grâce à cette partie de code, nous avons nos villes sous forme de coordonnées prêtes à être insérées dans des circuits.

```
class génération:
 def __init__(self):
   self.circuit_base = []
   self.parcours = parcours()
   self.circuit_int = self.parcours.villes()
# self.circuit_int circuit intermédiaire pour obtenir le circuit de base
# On veut le circuit de base sans les noms des villes inutiles dans le calcul de distance

   for i in range(0,len(self.circuit_int)):
       self.circuit_base.append(self.circuit_int[i][1])
# Etablit le circuit de base sans les noms de villes
```

FIGURE 8.2 – Création de la liste des coordonnées GPS à traiter.

8.3 Fichier *Population*

Nous créons une classe *population* dans laquelle nous définissons deux fonctions : *__init__* et *individu*. Dans la fonction *__init__*, il est à noter que deux paramètres sont pris en compte : *taille_population* et *génération*.
Cette fonction initialise une liste *circuit_all*, vide au départ. Les individus sont créés dans la seconde fonction *individu*. Un individu correspond à un circuit appelé *new_circuit*. Ce circuit est créé grâce à la fonction *sample* qui a pour but de disposer dans un ordre aléatoire les couples de coordonnées. Les individus créés, autant de fois que la taille de la population l'exige, complètent la liste *circuit_all*.
Cette partie de code correspond à la création de la population initiale de notre AG.

```
from génération import *
from math import *
from random import *

class population:
 def __init__(self,taille_population,génération):
  self.génération = génération
  self.taille_population = taille_population
  self.circuit_all = []

  for a in range(0,self.taille_population):
   self.circuit_all.append(self.individu())
   # Introduit les individus à la population

 def individu(self):
  self.new_circuit = sample(self.génération.circuit_base,len(self.génération.cir
  return self.new_circuit
```

FIGURE 8.3 – Génération de la population initiale.

8.4 Fichier *Algorithme*

Ce fichier est divisé en neuf parties que nous allons détailler une par une.

8.4.1 Fonction "*__init__*"

Ce bloc a pour fonction d'initialiser plusieurs variables comme les listes *distance_circuits*, *new_population*, *distance_moyenne*, *fit* ou encore *new* qui seront appelées en cours de programme et explicitées dans la suite de cette section. C'est aussi le lieu de définition de paramètres comme le taux de mutation (*muter*) qui est ici fixé à 0.02.

```
class algorithme:

 def __init__(self,population,génération):
  self.distance_circuits = []
  self.muter = 0.02    #  ----------TEST DEGRE DE MUTAT
  self.population = population # self.population égalar
  self.génération = génération # self.génération égalar
  self.new_population = []
  self.new_population = self.population.circuit_all[:]
  self.distance_moyenne = 0.0 # Définition de la distar
  self.make = 0
  self.fit = self.fittest(self.population.circuit_all)
  self.make = 1
  self.new_population[0] = self.fit
```

FIGURE 8.4 – Initialisation des variables et paramètres.

Ensuite, nous créons une première boucle. Celle-ci utilise deux fonctions : *evolution* et *crossover* (cfr. points 8.4.5 et 8.4.7). La fonction *evolution* est utilisée pour définir les deux parents et la fonction *crossover* est utilisée pour la création d'enfants qui sont ajoutés dans le circuit *new_population* : comme le nom de variable l'indique, une nouvelle génération d'individus se crée de la sorte.

```
for i in range(0,len(self.new_population)): #
 parent1 = self.evolution() # le parent1 est é
 parent2 = self.evolution() # le parent2 est é
 enfant = self.crossover(parent1,parent2) # l'
 self.new_population[i] = enfant # ajout de l'
```

FIGURE 8.5 – Génération d'une nouvelle population.

Une deuxième boucle permet le remplacement de certains circuits par de nouveaux générés aléatoirement et une troisième donne la possibilité à notre population *new_population* de subir une ou plusieurs mutations via la fonction *mutation*(point 8.4.6).

```
for a in range(1,int(len(self.new_population)/10)+1):
    self.new_population[-a] = (self.population.individu())
```

FIGURE 8.6 – Remplacement de circuits par d'autres générés aléatoirement.

```
for a in range(0,len(self.population.circuit_all)):
  self.mutation(self.new_population[a]) # exécution
```

FIGURE 8.7 – Appel à la mutation.

Le code désigne par *fit* le circuit optimal dans la population en cours et par *fit1* le meilleur circuit de *new_population*. Si la longueur de circuit de ce dernier est plus grande que celle de *fit*, ce qui signifierait que *fit1* est moins performant que *fit*, alors nous remplaçons le premier élément de *new_population* par *fit*. Sinon, c'est l'inverse qui se produit.

```
self.fit1 = self.fittest(self.new_population)
if self.distance(self.fit) < self.distance(self.fit1):
    self.new_population[0] = self.fit
else:
    self.new_population[0] = self.fit1
```

FIGURE 8.8 – Désignation du meilleur individu.

Pour terminer, l'ancienne population étant inutile, elle est remplacée par la nouvelle afin de réitérer le même processus.

```
self.population.circuit_all = self.new_population[:] #l'ancienne populat
```

FIGURE 8.9 – Remplacement de population.

8.4.2 Fonction *calcul de distance*

Pour le calcul de distance entre deux villes du circuit, nous faisons appel aux formules de trigonométrie sphérique afin de tenir compte de la courbure de la surface terrestre et d'utiliser les valeurs de latitude et longitudes renseignées.

```
def calcul_distance(self,ville1,ville2):
    #Trigonométrie Sphérique
    la_1,lo_1= ville1
    lor_1=pi*lo_1/180
    lar_1=pi*la_1/180
    la_2,lo_2= ville2
    lor_2=pi*lo_2/180
    lar_2=pi*la_2/180
    dl=lor_2-lor_1
    R=6378.137
    dist=R*acos(sin(lar_1)*sin(lar_2)+cos(lar_1)*cos(lar_2)*cos(dl))
    return dist
```

FIGURE 8.10 – Calcul de la distance entre deux villes au départ des coordonnées GPS.

8.4.3 Fonction *distance*

Le but de cette fonction est de calculer la longueur totale d'un circuit (nommée *distance_totale*). Au moyen d'une boucle, on parcourt le circuit et on ajoute la distance entre une ville et la suivante. Comme il s'agit d'un circuit fermé, on n'oublie pas d'ajouter la distance entre la dernière ville de la liste et la première. La longueur totale d'un circuit représente la fonction d'adaptation de nos individus.

```
def distance(self,circuit):
    self.distance_totale = 0.0
    for i in range(0,len(circuit)-1):
        self.distance_totale += self.calcul_distance(circuit[i],circuit[i+1])
    self.distance_totale += self.calcul_distance(circuit[0],circuit[-1]) # Cal
    return self.distance_totale
```

FIGURE 8.11 – Calcul de la longueur totale d'un circuit.

8.4.4 Fonction *fittest*

C'est cette fonction qui se charge d'évaluer la fonction d'adaptation et de retourner le meilleur individu de la population. Une boucle permet en effet de comparer chaque longueur de circuit à la longueur retenue comme optimale. Si l'individu a un meilleur fitness (la longueur de circuit est plus courte), c'est sa fonction d'adaptation qui est choisie comme référence.

```
def fittest(self,liste):

 fitness = 0 # Définition de la fitness à Zéro
 for i in range(0,len(liste)): #   De 0 au Nbre d'éléments contenus dans la lis
     a = self.distance(liste[i])
     if self.make == 0:
       self.distance_circuits.append(a)
     if self.distance(liste[fitness]) >= a: # Détermine le meilleur individu de
      fitness = i
 return liste[fitness] # Renvoie le meilleur individu de la liste (par sa dista
```

FIGURE 8.12 – Recherche du meilleur individu dans la population.

8.4.5 Fonction *evolution*

Nous avons la possibilité de choisir dans cette partie deux méthodes de sélection : la méthode dite de la roulette et la méthode dite du tournoi (cfr. Opérateurs de sélection en page 36).

- **La méthode *Roulette***

  Notre sélection par roulette repose sur le principe suivant : chaque fitness (*distance_ circuits*) est multiplié par un nombre entre 0 et 1 choisi au hasard. Cette liste est ensuite triée par ordre croissant. L'individu sélectionné est alors le premier élément de cette liste.

```
#Roulette
self.eval = [] # Définition de la liste eval
for i in range(0,len(self.population.circuit_all)): # De 0 au nbre de circuits c
    self.eval.append(int(self.distance_circuits[i]) * random())
return sorted(self.population.circuit_all,key=lambda Algorithme: self.eval)[0]
```

FIGURE 8.13 – Méthode de la roulette.

- **La méthode *Tournoi***

  Dans le tournoi, on sélectionne une poule formée d'un certain nombre d'individus choisis aléatoirement dans la population. Cette sélection prend la forme d'une liste *indice* créée avec la syntaxe *sample*. Ensuite, nous ajoutons dans une liste nommée *selection* les éléments issus de notre liste *distance_ circuits* situés aux rangs désignés par *indice*.La liste *selection* est recopiée sous le nom *ind* qui est trié par ordre croissant. Finalement, cette fonction retourne un élément de *population.circuit_ all* d'indice égal à *ind[0]*, celui dont le fitness est le meilleur, soit le gagnant du tournoi.

```
def evolution(self):                    # ----------TEST SUR LES METHODES DE SEL
      #Sélection
      self.indice = sample(range(0,50),5)
      self.selection = []
      for i in range(0,5):
        self.selection.append(self.distance_circuits[self.indice[i]])
      self.ind = self.selection[:]
      self.ind.sort()
      return self.population.circuit_all[self.selection.index(self.ind[0])]
```

FIGURE 8.14 – Méthode du tournoi.

8.4.6 Fonction *mutation*

Le procédé utilisé dans cette fonction a pour seul but de créer une possible mutation dans le code génétique d'un enfant. Une mutation revient ici à intervertir deux villes dans le circuit. On parcourt d'abord le circuit au moyen d'une boucle. A chaque position dans la liste (*Pos1*), on choisit un nombre entre 0 et 1. Si celui-ci est inférieur au taux de mutation fixé plus tôt (cfr. point 8.4.1), la permutation a lieu. Pour cela, on désigne aléatoirement une position dans le circuit, appelée *Pos2* et on échange ensuite la ville du circuit à la position *Pos2* et la ville du circuit à la position *Pos1*.

```
def mutation(self,circuit):
 for Pos1 in range(0,len(circuit)):
   if random() < self.muter: # self.muter défini dans __init__
    Pos2 = randrange(0,len(circuit))

    Ville1 = circuit[Pos1]
    Ville2 = circuit[Pos2]

    circuit[Pos1] = Ville2
    circuit[Pos2] = Ville1
```

FIGURE 8.15 – Opérateur de mutation.

8.4.7 Fonction *crossover*

Il existe aussi deux possibilités de croisement entre individus. La première est appelée METHODE 1 et la seconde METHODE 2.

- **METHODE 1**

  Nous commencons par initialiser la liste *enfant* et par choisir aléatoirement le point de croisement contenu dans la valeur *cross*. La suite se passe en trois étapes (3 boucles). La première boucle sert à compléter la liste *enfant* par la première partie de liste de longueur *cross* du circuit *parent1*. La seconde boucle permet de terminer la création de *enfant* en ajoutant la deuxième partie (de longueur de "*parent1* - *cross*") de liste *parent2*.
  Pour résoudre le problème que certaines villes apparaissent deux fois dans la liste *enfant*, contrairement à ce que prévoit le cycle hamiltonien du PVC, la dernière boucle vérifie la présence de toutes les villes dans le circuit. S'il n'en manque aucune, la fonction se cloture en retournant la liste *enfant*. Si, au contraire, une ville est manquante, une boucle se lance pour trouver la (ou les) ville(s) double(s) et la(les) remplacer par la manquante.

```
#METHODE 1

def crossover(self,parent1,parent2):
 enfant = []
 cross = randrange(0,len(self.génération.circuit_base))

 for i in range(0,cross):
  enfant.append(parent1[i]) # Ajoute à enfant la première partie du parent1
 for i in range(1,len(parent1)-cross+1):
  enfant.append(parent2[-i]) # Ajoute à enfant la deuxième partie du parent2

 for i in range(0,len(parent1)):
    if enfant.count(parent1[i]) == 0:  # Trouve la ville manquante
     for ii in range(0,len(parent1)):
       if enfant.count(enfant[ii]) > 1:  # Trouve la ville double
          enfant[ii] = parent1[i] # Remplace la ville double par la manquante
          break
 return enfant
```

FIGURE 8.16 – Méthode de croisement 1.

- **METHODE 2**

  La seconde méthode commence par définir la valeur *cross* du point de croisement et deux listes vides nommées *part* et *add* qui, reconstituées, formeront notre *enfant*. On complète *add* par les éléments de *parent1* jusqu'au point de croisement *cross* et *part* par les éléments de *parent2* situés après *cross*. La jonction de *add* et *part* constitue la liste *enfant*. Comme dans la première méthode, une boucle traque les doublons et les remplace par les villes manquantes.

```
#METHODE 2

 def crossover(self,parent1,parent2):
   cross = randrange(3,len(self.génération.circuitbase())-3)
   add =[]
   part =[]

   for ad in range(0,len(parent1)):
       if ad < cross:
         add.append(parent1[ad])


   for par in range(0,len(parent2)):
       if par >= cross:
         part.append(parent2[par])


   enfant = add + part
```

./...

```
.../.
for a in range(0,len(add)):
    for i in range(0,len(part)):
        if add[a] == part[i]:
            part[i] = None


for m in range(0,len(parent1)):
  i = 0
  for y in range(0,len(enfant)):
     if parent1[m] == enfant[y]:
         i += 1
  if i == 0:
     for l in range(0,len(part)):
         if part[l] == None:
             part[l] = parent1[m]
             break

enfant = add + part
return enfant
```

FIGURE 8.17 – Méthode de croisement 2.

8.4.8 Fonction *distances moyennes*

Cette fonction permet de connaître l'évolution du fitness moyen en cours de processus en calculant la longueur de circuit moyen.

```
def distances_moyenne(self):
    for i in range(0,len(self.distance_circuits)):
        self.distance_moyenne += self.distance_circuits[i]
    return (self.distance_moyenne/len(self.population.circuit_all))
```

FIGURE 8.18 – Calcul de la longueur de circuit moyen.

8.4.9 Fonction *circuit final*

Le but étant de trouver le meilleur itinéraire à suivre, nous créons une liste *circuitfinal* qui renvoit un circuit composé des noms des villes uniquement (sans coordonnées). Suivent alors deux boucles dont le but est de faire correspondre les noms de villes à chaque terme du meilleur circuit trouvé par l'AG. La troisième boucle permet de tourner le circuit de telle façon que Washington D.C. soit la première ville du circuit final.

```
def circuit_final(self,circuit):
    self.circuitfinal = [] # Renvoie un circuit composé des noms des villes un
    for i in range(0,len(circuit)):
        for ii in range(0,len(circuit)):
            if circuit[i] == self.génération.circuit_int[ii][1]:  # Trouve les
                self.circuitfinal.append(self.génération.circuit_int[ii][0])
    for a in range(0,len(self.circuitfinal)):
        if self.circuitfinal[a] == 'Washington, D.C.':
            add = self.circuitfinal[a:]
            part = self.circuitfinal[:a]
            self.circuitfinal = add + part
    return self.circuitfinal
```

FIGURE 8.19 – Constitution du meilleur itinéraire.

8.5 Fichier *Programme*

8.5.1 Fonction *menu*

Cette fonction permet d'interagir avec l'utilisateur. On demande à celui-ci de fixer 3 paramètres : la taille de la population, le nombre d'itérations et le nombre de résultats intermédiaires. Ces trois paramètres seront stockés dans une liste *paramètres* et par la suite appelés dans le programme sous forme de variables.

```
def menu():
    print("----------------------Menu----------------------")
    tp = input("----->   Introduire la taille de la population : ")
    tp = int(tp)
    ni = input("----->   Introduire le nombre d'itérations : ")
    ni = int(ni)
    np = input("----->   Introduire le nombre de résultats intermédiaires : ")
    np = int(np)

    return tp , ni , np
```

FIGURE 8.20 – Interface d'utilisation.

8.5.2 Fonction *menu continuer*

Lorsque le programme se cloture, la possibilité de recommencer le processus est toujours présente. L'utilisateur peut introduire de nouveaux paramètres comme le nombre d'itérations et le nombre de résultats intermédiaires.

```
def menu_continuer():
    print("--------------------Menu---------------------")
    ni = input("----->   Introduire le nombre d'itérations : ")
    ni = int(ni)
    np = input("----->   Introduire le nombre de résultats intermédiaires :  ")
    np = int(np)

    return ni , np
```

FIGURE 8.21 – Interface d'utilisation : suite.

8.5.3 Fonctions *utilisation* et *continuer*

Les deux fonctions sont presque identiques. Nous allons donc les décrire ensemble. Ces fonctions commandent réellement le processus de l'algorithme génétique en appelant les fichiers *génération*, *population* et *algorithme*. Ces fonctions utilisent les variables de la liste *paramètres* introduites par l'utilisateur dans l'interface. On y vérifie aussi au moyen d'une boucle si l'étape du processus demande l'affichage de résultats intermédiaires, demandés par l'utilisateur (au point 8.5.1). Le cas échéant, le programme commande l'affichage de la longueur moyenne et de la meilleure distance à parcourir.

```
def utilisation():
 g = génération()
 pop = population(paramètres[0],g)
 for i in range(0,paramètres[1]):
  Algorithme = algorithme(pop,g)  # Opère l'evolution de la population
  if i % (paramètres[1]/paramètres[2]) == 0:
      print("Meilleure distance actuelle : ",Algorithme.distance(Algorithme.new_
      print("Distance moyenne actuelle : ",Algorithme.distances_moyenne())
      print("\n")
      continue
 return g,pop,Algorithme
def continuer():
    for i in range(0,paramètres[0]):
        Algorithme = algorithme(pop,g)
        if i % (paramètres[0]/paramètres[1]) == 0:
             print("Meilleure distance actuelle : ",Algorithme.distance(Algorithme.new_population[0]))
             print("Distance moyenne actuelle : ",Algorithme.distances_moyenne())
             print("\n")
             continue
```

FIGURE 8.22 – Commande de l'exécution de l'algorithme génétique.

8.5.4 Fin du programme

Après *ni* itérations, le programme se clôture en affichant à l'utilisateur la distance initiale, la distance optimale, le parcours à effectuer et le temps mis par le programme pour trouver la solution au problème.

```
fin = time.time()

print("Distance inititale = ",Algorithme.distance(g.circuit_base))# La Distance initiale
print('Distance optimale = ',(Algorithme.distance(Algorithme.fittest(Algorithme.new_population))))
p_e = Algorithme.circuit_final(Algorithme.fittest(Algorithme.new_population)) # Renvoi du parcours
p_e = ','.join(p_e)
print('Le parcours à effectuer est : ',p_e)
print("Le Temps mis par le programme est de : ",fin-debut,"secondes") # Calcule le temps qu'a mis l


os.system("pause")
```

FIGURE 8.23 – Commande d'affichage de clôture.

Chapitre 9

Modulation de l'algorithme

9.1 Tour choisi

Pour mettre à l'épreuve notre algorithme, nous avons choisi de résoudre le PVC historique de Dantzig, Fulkerson et Johnson, les premiers à trouver une solution pour une instance du PVC de taille respectable. Le nombre d'instances traitées -49- semble un nombre raisonnable pour éprouver notre algorithme. Par ailleurs, historiquement, la publication de leurs travaux en 1954 marque une percée dans les méthodes de résolution de ce problème. A l'époque, John Holland ne parle pas encore des algorithmes génétiques. C'est une autre méthode que l'équipe de chercheurs emploie – le *Cutting-Plane* – dont la postérité résout aujourd'hui des instances à plusieurs milliers de villes. Pour tester la puissance de leur méthode de résolution, ils choisissent une ville dans chacun des 48 états continentaux des États-Unis de l'époque et ajoutent à la liste Washington D.C. La raison de ce choix particulier est simple : à cette époque, la plupart des distances de route entre ces villes sont faciles à obtenir à partir d'un atlas. La liste de ces villes est présentées en figure 9.1. En réalité, ils n'introduisent pas 49 villes dans leur programme mais 42, car ils notent que la tournée optimale à travers les 42 villes utilise des routes qui traversent 7 villes qui peuvent, dès lors, être exclues du processus.

1. Manchester, N. H.	*18. Carson City, Nev.*	*35. Atlanta, Ga.*
2. Montpelier, Vt.	*19. Los Angeles, Calif.*	*36. Jacksonville, Fla.*
3. Detroit, Mich.	*20. Phoenix, Ariz.*	*37. Columbia, S. C.*
4. Cleveland, Ohio	*21. Santa Fe, N. M.*	*38. Raleigh, N. C.*
5 Charleston, W. Va.	*22. Denver, Colo.*	*39. Richmond, Va.*
6. Louisville, Ky.	*23. Cheyenne, Wyo.*	*40. Washington, D. C.*
7. Indianapolis, Ind.	*24. Omaha, Neb.*	*41. Boston, Mass.*
8. Chicago, Ill.	*25. Des Moines, Iowa*	*42. Portland, Me.*
9. Milwaukee, Wis.	*26. Kansas City, Mo.*	*A. Baltimore, Md.*
10 Minneapolis, Minn.	*27. Topeka, Kans.*	*B. Wilmington, Del.*
11. Pierre, S. D.	*28. Oklahoma City, Okla.*	*C. Philadelphia, Penn.*
12. Bismarck, N. D.	*29. Dallas, Tex.*	*D. Newark, N. J.*
13 Helena, Mont.	*30. Little Rock, Ark.*	*E. New York, N. Y.*
14 Seattle, Wash.	*31. Memphis, Tenn.*	*F. Hartford, Conn.*
15. Portland, Ore.	*32. Jackson, Miss.*	*G. Providence, R. I.*
16. Boise, Idaho	*33. New Orleans, La.*	
17. Salt Lake City, Utah	*34. Birmingham, Ala.*	

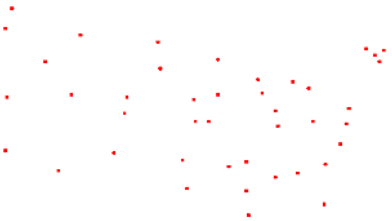


FIGURE 9.1 – Villes du PVC résolu par Dantzig, Fulkerson et Johnson en 1954. Les villes A à B sont les 7 villes exclues de l'algorithme de résolution[17]. Sous le tableau, une représentation plane des 42 villes à relier.

Il montrent alors que la tournée optimale à travers les villes 1,2, ..., 42,1 dans cet ordre est minimale pour ce sous-ensemble de 42 villes. De plus, puisque la route de la 40^e ville Washington, D. C. à la 41^e Boston, Massachusetts passe successivement par A, B, ..., G, le tour optimal est celui intégrant ces 7 villes dans le précédent tour entre les villes 40 et 41. Ainsi, visiter les villes 1, 2, ..., 40, A, B,..., G, 41, 42,1 dans cet ordre représente l'itinéraire le plus court, pour une distance de 12,345 miles, soit environ 19 867 km. Ce tour est illustré en figure 9.2.

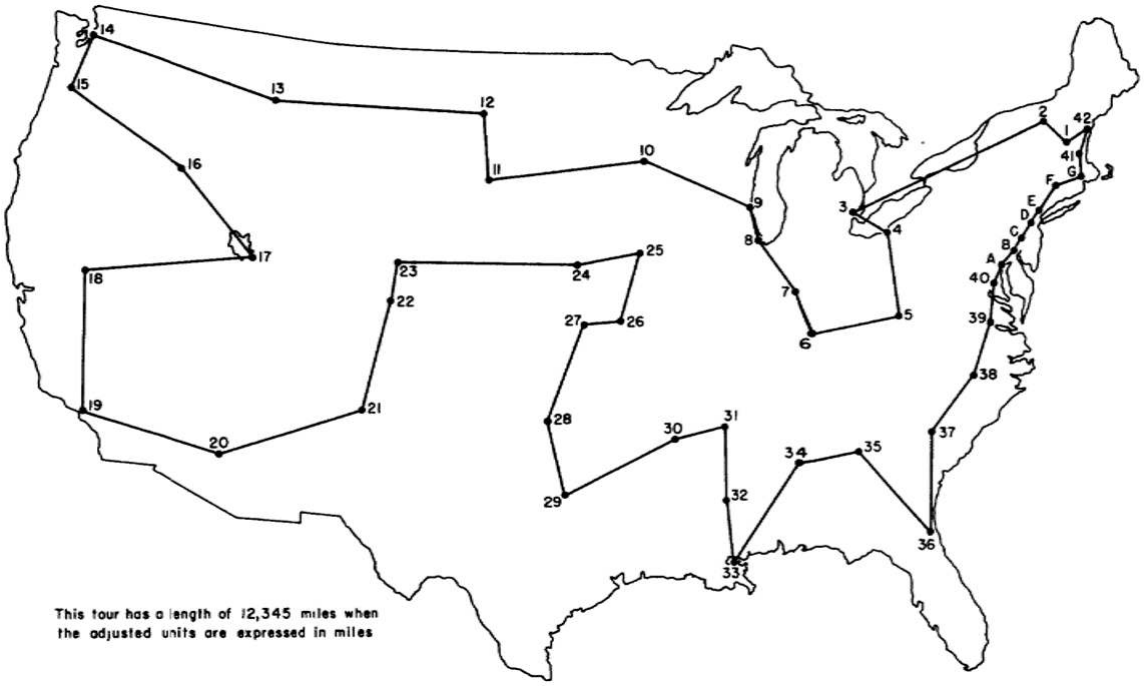


FIGURE 9.2 – Tour optimal retenu par Dantzig et son équipe [17].

9.2 Evolution de la population de génération en génération

La genèse d'une population initiale formée par des individus choisis aléatoirement, ici des itinéraires en boucle passant par les 49 villes à visiter, est la première étape de l'algorithme. Le travail de l'AG consiste alors à améliorer la population en cours en sélectionnant et croisant les individus les plus performants. A chaque itération de l'algorithme, on est censé trouver une population avec un meilleur fitness que l'ancienne. Le tableau et le graphique 9.2 illustrent ce fait. On remarque qu'entre la population initiale et la 10^{me} génération, l'amélioration est remarquable, tant pour le circuit optimal que pour le circuit moyen, avec des distances – minimales et moyennes - pratiquement divisées par un facteur 2 pour la population de 200 individus. Après cette dizaine de générations, la tendance à l'amélioration ralentit un peu jusqu'à l'obtention d'un comportement asymptotique. Cette convergence vers la stabilité indique la proximité de l'optimum recherché. Dans les trois situations, la longueur du circuit initial était de l'ordre de 84 milliers de kilomètres et nous finissons avec des longueurs de circuit optimales de 23 169 km, 21 209 km et 21 244 km respectivement pour les populations de 100, 200 et 1000 individus, soit pratiquement une division par un facteur 4.

Si la longueur minimale n'augmente pas d'une génération à la suivante – le meilleur individu d'une génération n'est jamais moins performant que celui de la génération précédente, on voit qu'une régression ponctuelle n'est pas à exclure en ce qui concerne le fitness moyen de la population : on observe que le graphe de la figure 9.4 n'est pas celui d'une fonction monotone.

Enfin, la longueur moyenne de circuit reste toujours supérieure à la longueur optimale trouvée. Cela montre que le processus pourrait se poursuivre pour affiner le résultat : on pourrait envisager un nombre plus élevé de générations pour voir si la décroissance, bien que faible, se poursuit.

Taille	100		200		1000	
Nbre de génér.	*Long. Opt. [km]*	*Long.Moy. [km]*	*Long. Opt. [km]*	*Long.Moy. [km]*	*Long. Opt. [km]*	*Long.Moy. [km]*
0	60066	81650	62160	81917	54270	81220
10	44696	51690	35740	43073	38815	46215
20	33499	39067	31504	37481	30137	37382
30	32224	38596	30976	38121	27425	35861
40	29487	35308	29949	34436	26140	33903
50	28829	36289	29798	35712	25461	32692
60	27915	34535	28320	34436	24648	32547
70	27713	35439	28056	33904	23747	30916
80	26031	32491	24817	30864	23028	31645
90	25799	32858	23607	29921	23028	30110
100	25799	32906	22927	29853	22855	31417
110	24898	34143	22927	33235	22855	32228
120	24436	31728	22927	30448	22855	30142
130	24436	31898	22442	29070	22831	31340
140	24436	31261	22442	29581	22831	31206
150	24168	33229	22385	32147	22585	31357
160	24133	30818	21635	28641	22184	31259
170	23484	30817	21635	30066	22184	29509
180	23484	29739	21635	30941	21736	28787
190	23484	31164	21275	29136	21736	29623

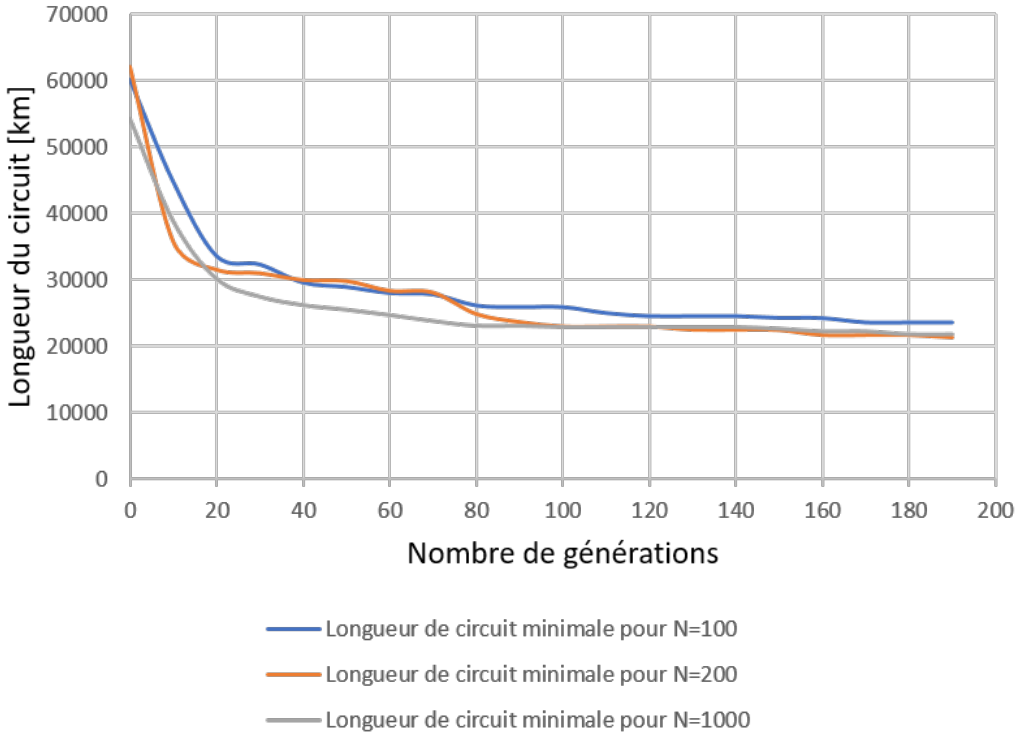


FIGURE 9.3 – Evolution de la longueur de circuit minimale en fonction du nombre de générations. Taille de la population N : 100, 200 et 1000 – Nbre d'itérations : 200.

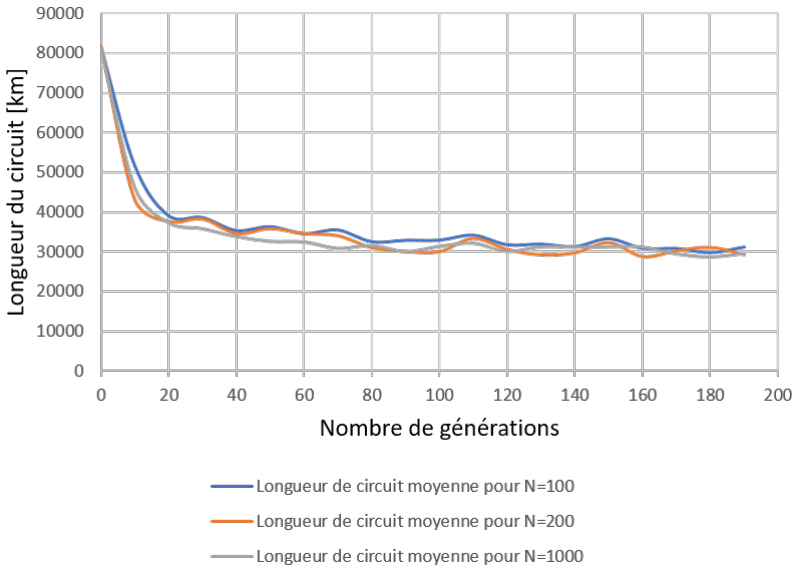


FIGURE 9.4 – Evolution de la longueur de circuit moyenne en fonction du nombre de générations. Taille de la population N : 100, 200 et 1000 – Nbre d'itérations : 200.

Le tableau 9.6 et la figure 9.5 montrent une résolution du problème par 1000 itérations. Le comportement décrit précédemment est confirmé pour les 100 premières générations : globalement, le fitness s'est amélioré de façon significative. Mais on voit également que cette amélioration se poursuit durant les 900 générations suivantes : ce n'est qu'à partir de la génération 600 que la fonction devient pratiquement constante. Pour obtenir une solution encore meilleure, il convient donc d'augmenter le nombre d'itérations à effectuer. Par contre, l'écart entre la distance moyenne et la distance minimale – de 8 milliers de kilomètres en moyenne- se maintient : bien que la population s'améliore globalement, celle-ci contient toujours des individus moins performants. Enfin, la tendance ponctuelle à la régression dans la population globale ne disparaît pas lorsque le nombre d'itérations augmente. Le tableau 9.6 semble indiquer également que l'amélioration des performances dépend de la taille de la population traitée : une population de 200 individus nous offre la plus petite longueur de circuit (17 257 km). La recherche du paramétrage optimal de l'algorithme traité dans la section suivante montre le rôle que joue la taille de la population dans l'efficacité de l'algorithme.

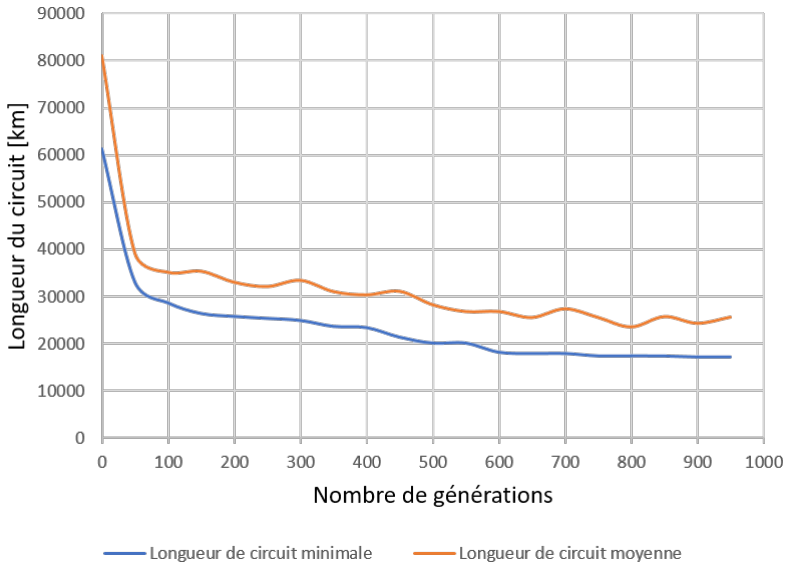


FIGURE 9.5 – Evolution de la longueur de circuit en fonction du nombre de générations. Taille de la population N : 200 – Nbre d'itérations : 1000.

Taille	100		200		1000	
Nbre de génér.	*Long. Opt. [km]*	*Long.Moy. [km]*	*Long. Opt. [km]*	*Long.Moy. [km]*	*Long. Opt. [km]*	*Long.Moy. [km]*
0	61261	81656	61218	81107	57789	81249
50	30385	35423	33064	39242	34390	42288
100	28107	35082	28680	35213	26364	35679
150	27668	34544	26436	35433	24672	31254
200	26394	32962	25821	33101	24439	33092
250	25531	32512	25404	32189	23694	32407
300	24889	33302	24965	33507	23614	32391
350	24418	31851	23716	31122	23614	32258
400	23942	35001	23458	30434	23070	31148
450	23293	30625	21437	31187	22870	31204
500	23286	31555	20189	28338	22869	31264
550	23286	31318	20189	26896	22602	30163
600	22711	30513	18257	26901	21911	31799
650	22508	30405	17997	25630	21524	32176
700	21722	29891	17997	27478	21066	30919
750	21722	30640	17481	25653	20894	31566
800	21621	28043	17481	23635	20894	30129
850	21621	28112	17465	25821	20320	28747
900	21621	29400	17257	24410	20320	28899
950	21303	28970	17257	25715	20092	27205

FIGURE 9.6 – Evolution de la longueur de circuit en fonction du nombre de générations. Taille de la population N : 100, 200 et 1000 – Nbre d'itérations : 1000.

9.3 Etude des performances des différents paramétrages

La difficulté majeure des algorithmes génétiques ne réside pas dans la mise en œuvre de l'algorithme lui-même, mais plutôt, dans le choix des valeurs adéquates des différents paramètres de ce dernier. En effet, le programmeur est libre de déterminer plusieurs paramètres qui règlent l'algorithme. Ce sont les choix posés, en fonction du problème, qui déterminent l'efficacité de celui-ci. Nous avons donc procédé par tâtonnements en évaluant la performance de l'algorithme selon les facteurs suivants :

- le nombre d'itérations,
- la taille de la population N,
- le mode de sélection,
- le mode de reproduction,
- le taux de mutation.

Pour mesurer la performance de l'algorithme, il faut bien sûr s'intéresser à la solution retenue. C'est pourquoi, nous avons conservé la valeur de la longueur optimale trouvée au terme du processus. En outre, nous avons retenu la longueur moyenne du circuit de la population au terme des itérations car la façon dont la population globale évolue en cours de processus présente un intérêt également. Enfin, l'efficacité se mesure également au temps écoulé avant de trouver la solution. Nos résultats présentent donc l'étude de cette troisième valeur. Le hasard étant omniprésent, les exécutions ne démarrent pas toutes avec la même population initiale. La zone de recherche initiale peut pourtant fortement influencer la rapidité de convergence : si le hasard voulait que la première zone de recherche contînt des individus très performants, le résultat final serait peut-être amélioré et la convergence rapide. C'est pourquoi, nous avons reproduit 5 fois toutes les séries de tests. Les tableaux qui suivent présentent les valeurs moyennes obtenues lors de ces tests.

9.3.1 Etude de la taille de la population N

Nous avons exécuté le programme en fixant le nombre d'itérations à 200 et le taux de mutation à 2%. La méthode de sélection retenue est la roulette. Le paramètre variable est ici la taille de la population. On observe que la performance des individus dépend bien de la taille de la population initiale. Malgré quelques anomalies locales, on peut dire que le fitness est une fonction croissante de la taille de la population : plus la taille de la population est grande, meilleure est la solution optimale trouvée. Cependant, le temps est lui aussi une fonction croissante de la taille de la population et cette croissance prend des allures exponentielles : si la taille de la population augmente, le nombre d'actions élémentaires à effectuer augmente également.
L'efficacité consiste à trouver un juste équilibre entre la performance du meilleur individu et le temps mis pour le trouver. Ainsi, on constate qu'une population de 200 individus est nettement préférable à une population de 25. Par contre, l'amélioration due au passage d'une population de 300 à celle de 1000 individus ne justifie peut-être pas l'augmentation importante du temps d'exécution. Il faut pouvoir déterminer si on a avantage à travailler avec une grande population, en limitant le nombre d'itérations, ou s'il vaut mieux travailler avec une petite population mais en permettant un nombre de générations élevé.

Taille N	*Long.Opt. [km]*	*Long.Moy [km]*	*Temps[s]*
25	31004	54882	13
50	27143	42416	12
75	25800	38713	16
100	24808	36013	20
125	24946	35385	24
150	24128	34086	28
175	23599	33161	35
200	23136	32764	41
300	22182	31003	78
400	22730	30807	131
500	22686	30409	204
1000	20860	28202	953

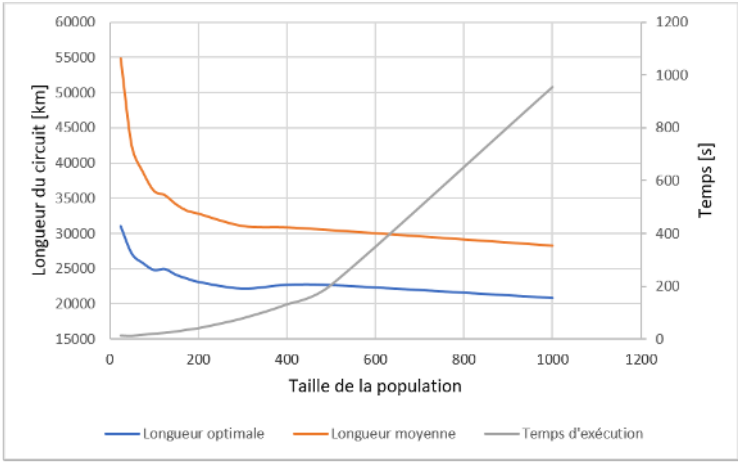


FIGURE 9.7 – Evolution de la longueur de circuit en fonction de la taille de la population. Nbre d'itérations : 200 - Taux de mutation : 2% - Méthode de sélection : Roulette.

9.3.2 Etude du nombre d'itérations

Cette fois, nous voulons observer quel est le nombre d'itérations idéal pour exécuter l'algorithme. Nous réalisons 3 études de ce paramètre pour 3 tailles de population (N=100, N= 200 et N= 1000) en fixant chaque fois le taux de mutation à 2% et en appliquant la sélection par roulette.
Le graphe obtenu dans le cas de la population de 100 confirme dans un premier temps qu'un certain nombre de répétions de l'algorithme est nécessaire pour obtenir un résultat satisfaisant. Il faut plus de 500 itérations pour obtenir un résultat inférieur à 20 000 km. Mais de façon très surprenante, il présente aussi un puits : il semblerait que l'on obtienne un optimum local des meilleures solutions (longueur de 17 138 km) pour 750 itérations. Ce n'est pas logique puisque la répétition du processus est censée améliorer la solution trouvée. N'oublions pas que notre algorithme n'est pas déterministe, il ne nous garantit pas d'obtenir le minimum absolu, seulement de s'en rapprocher en un temps raisonnable. L'apparente meilleure performance d'un nombre de 750 n'est qu'un leurre. Le hasard qui intervient à de nombreux niveaux de l'algorithme peut nous réserver bien des surprises.

Nbre d'itér.	Long.Opt. [km]	Temps[s]
25	38130	9
50	30871	14
100	28571	20
200	24818	37
300	23434	49
400	21191	63
500	21003	75
750	17138	107
1000	20196	142
2000	18874	284

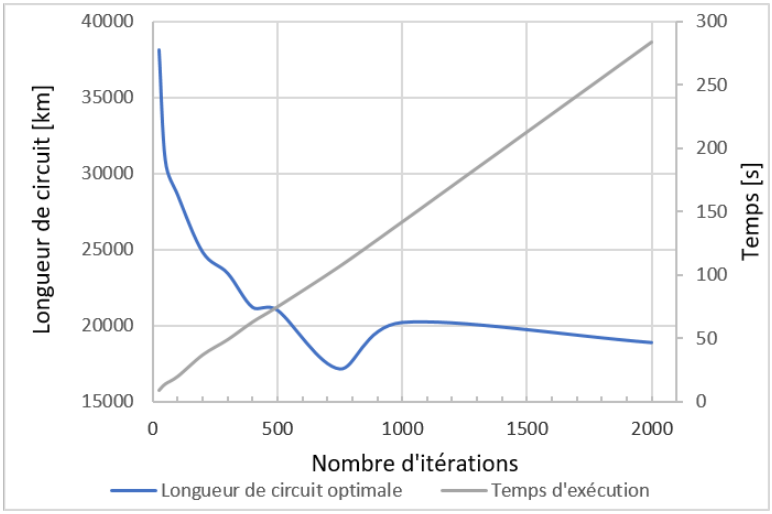


FIGURE 9.8 – Evolution de l'optimum trouvé en fonction du nombre d'itérations.
Taille de la population N :100 - Taux de mutation : 2% - Méthode de sélection : Roulette.

Nbre d'itér.	Long.Opt. [Km]	Temps [s]
25	33776	5,7
50	27982	11,26
100	24551	21,6
200	21636	42,49
300	20776	63,95
400	20183	84,57
500	19592	105
750	19228	158
1000	18850	210
2000	17801	419

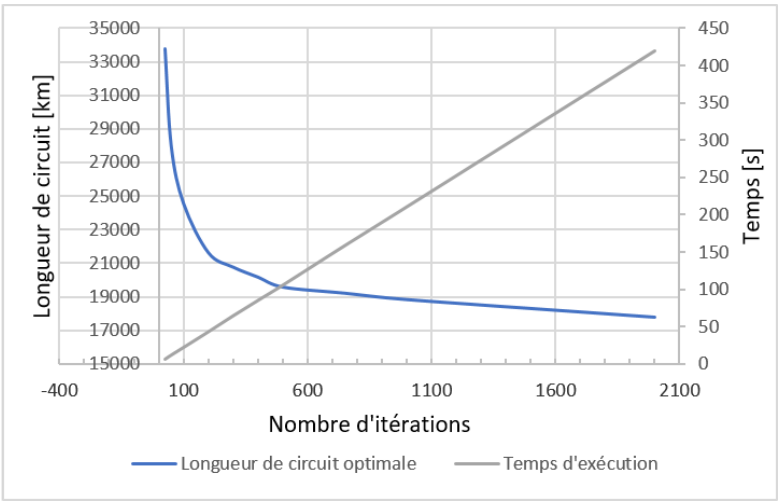


FIGURE 9.9 – Evolution de l'optimum trouvé en fonction du nombre d'itérations.
Taille de la population N :200 - Taux de mutation : 2% - Méthode de sélection : Roulette.

Nbre d'itér.	Long.Opt. [Km]	Temps [s]
25	32016	131
50	24423	247
100	22593	494
150	20535	741
200	19246	954
300	19796	1467
500	18517	2470

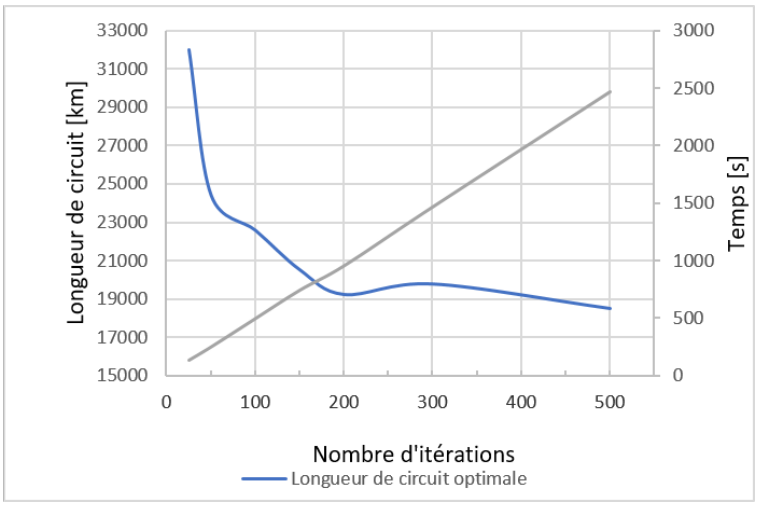


FIGURE 9.10 – Evolution de l'optimum trouvé en fonction du nombre d'itérations.
Taille de la population N :1000 - Taux de mutation : 2% - Méthode de sélection : Roulette.

L'évolution du temps d'exécution en fonction du nombre d'itérations montre bien entendu une relation de proportionnalité : les autres paramètres étant fixés, chaque itération demande le même temps de traitement. Ceci nous permet de répondre à la question précédente : vaut-il mieux limiter le nombre d'itérations avec une grande population ou, à l'inverse, travailler avec une petite population qu'on laisse évoluer sur plus de générations ? Le croissance linéaire nous renseigne sur le temps de vie d'une génération en fonction de la taille de la population : une itération demande un temps d'environ 0,14 s pour une population de 100 individus, 0,21 s pour 200 individus et devient 4,94 s pour 1000 individus !

Par ailleurs, on voit que le processus opéré 2000 fois sur une population de 200 individus donne une distance plus courte (17 801 km) que le processus opéré 500 fois sur une population de 1000 individus (18 517 km), tout en demandant un temps nettement moins long (7 minutes contre 41 minutes).

Lorsqu'on couple grande population et nombreuses itérations, on constate une ascension vertigineuse du temps d'exécution, sans pour autant améliorer significativement l'optimum. Il vaut donc mieux limiter la taille de la population à 100 ou 200 individus, mais lui permettre d'évoluer sur de nombreuses générations.

9.3.3 Etude du taux de mutation

Nous appelons *taux de mutation* la probabilité qu'un enfant engendré lors du crossover subisse une mutation génétique. Il s'agit donc d'une valeur comprise entre 0 et 1.
L'algorithme pourrait très bien fonctionner sans cet opérateur, tout en permettant l'évolution des individus. Nous verrons toutefois que cette fonction "de luxe" améliore significativement la rapidité de convergence vers l'optimum désiré.
Sur une petite population de 100 individus, on peut déjà dégager une tendance générale : un trop grand taux de mutation, supérieur à 0,1, a tendance à détériorer les résultats.
Pour des valeurs de probabilité de mutation comprises dans l'intervalle [0,0.07], il est difficile de tirer des conclusions. C'est pourquoi, nous recommençons l'expérience avec une population de 1000 individus.

Taux	*Long.Opt. [km]*	*Long.Moy [km]*
0	28707	32239
0,01	29046	48362
0,02	24127	31694
0,03	23941	33797
0,04	24089	34625
0,05	26940	38871
0,07	25629	37470
0,1	31092	50743
0,2	42176	65652
0,3	47417	72296

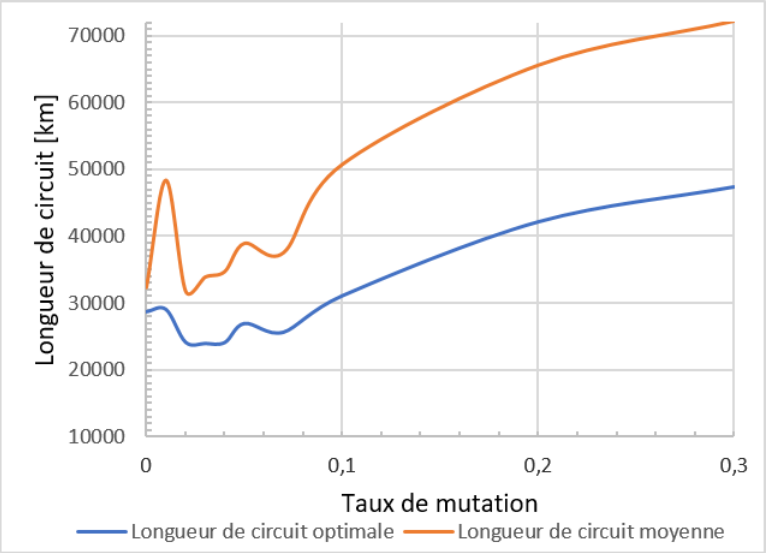


FIGURE 9.11 – Evolution de l'optimum trouvé en fonction du taux de mutation.
Taille de la population N :100 - Nombre d'itérations : 200 - Méthode de sélection : Roulette.

Taux	Long.Opt. [km]	Long.Moy [km]
0	30320	34282
0,01	20514	26689
0,02	20246	28182
0,03	20228	30429
0,04	19258	30944
0,05	19456	34319
0,07	22268	39307
0,1	25420	46926
0,2	37366	64607
0,3	42661	71324

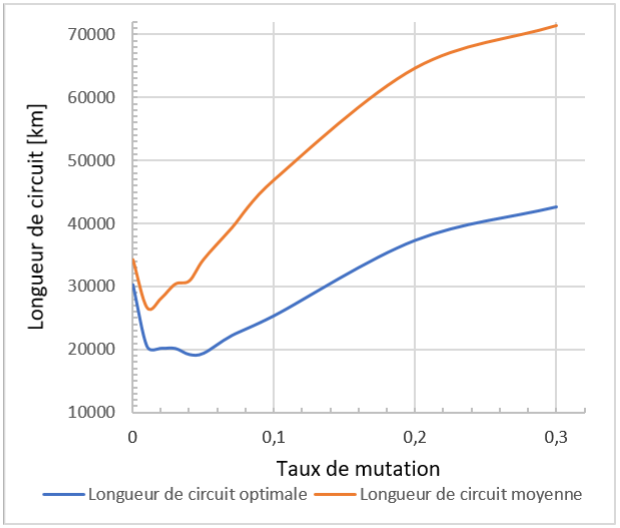


FIGURE 9.12 – Evolution de l'optimum trouvé en fonction du taux de mutation.
Taille de la population N :1000 - Nombre d'itérations : 200 - Méthode de sélection : Roulette.

Les résultats obtenus avec une population de 1000 individus confirment que, si le taux de mutation est grand (supérieur à 7%), alors la recherche devient purement aléatoire : le hasard génétique prime sur l'héritage parental. La population est diversifiée et l'AG perd de son efficacité.

Par ailleurs, on s'aperçoit que la mutation permet d'améliorer les résultats : un taux nul – qui correspond à supprimer la possibilité de mutation au sein de la population – offre une solution optimale moins performante qu'un taux de mutation compris entre 1% et 5%. La mutation permet la diversification de la population et l'empêche de stagner. Il semblerait que, dans le cas de notre AG, une probabilité de mutation dans l'intervalle [0.01, 0.05] donne les meilleurs résultats. Dans cette plage de valeurs, la performance évolue peu : les optima trouvés sont tous équivalents. En agrandissant la fenêtre (cfr.figure 9.13), on voit cependant que le fitness moyen, lui, se détériore quelque peu lorsque le taux de mutation augmente. Cela confirme bien que la mutation renforce les variations génétiques au sein d'une population.

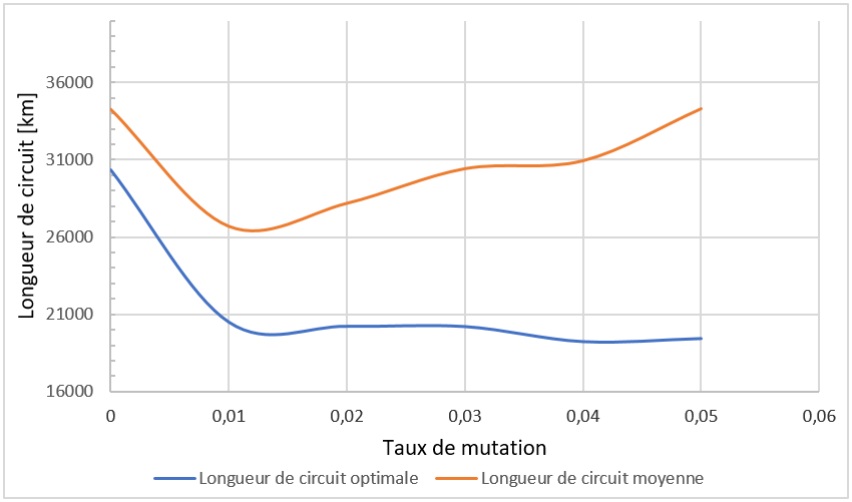


FIGURE 9.13 – Evolution de l'optimum trouvé en fonction du taux de mutation.
Taille de la population N :1000 - Nombre d'itérations : 200 - Méthode de sélection : Roulette.

Si on répète le processus itératif jusqu'à 1000 générations (sur une population de 200 individus), afin de laisser le temps aux mutations d'agir sur l'espèce, on obtient des résultats plus précis : le graphe 9.14 présente un minimum pour un taux de mutation de 2%.

Taux	Long. Opt. [km]	Long. Moy. [km]
0	20889	26096
0,01	17990	29922
0,02	16929	35902
0,03	17988	37106
0,04	21427	40261
0,05	22203	37634
0,07	23020	51133
0,1	31092	50743
0,2	42176	65652
0,3	47417	72296

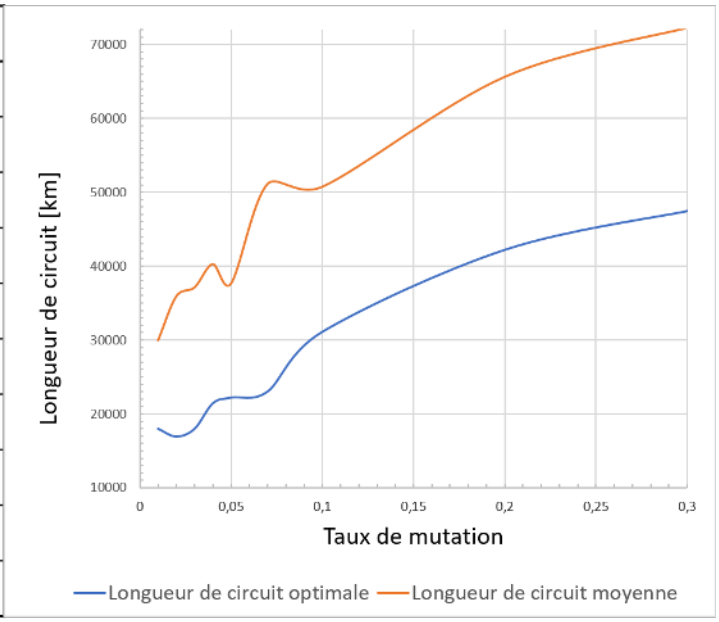


FIGURE 9.14 – Evolution de l'optimum trouvé en fonction du taux de mutation.
Taille de la population N :200 - Nombre d'itérations : 1000 - Méthode de sélection : Roulette.

L'étude de l'évolution de la longueur optimale de circuit de génération en génération illustrée par la figure 9.15 montre bien l'impact du taux de mutation sur l'évolution du meilleur individu. L'amélioration la plus rapide est obtenue pour le taux de 5% : après 20 itérations, la longueur optimale est déjà pratiquement divisée par 3, alors que le taux de 30% permet à peine de diviser par 2 cette distance au bout de plus de 60 itérations, résultat qui ne s'améliore pas au terme du processus.

L'absence de mutation donne de moins bons résultats également : il faut attendre 100 itérations pour diviser par un facteur 3 la longueur de circuit initiale, avec des résultats stagnants passé cette limite.

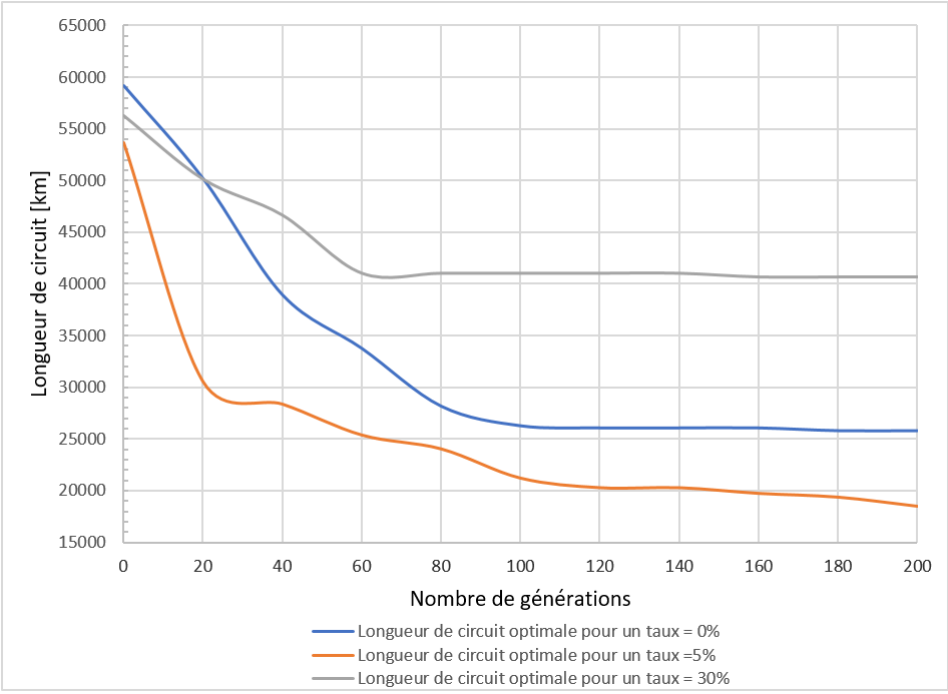


FIGURE 9.15 – Evolution de la longueur de circuit optimale en fonction du nombre de générations pour différents taux de mutation.
Taille de la population N :1000 - Méthode de sélection : Roulette.

L'expérience nous a montré que le taux de mutation était un paramètre plutôt complexe à quantifier pour en tirer le maximum de potentiel. Tantôt trop bas, il devient inutile ; tantôt trop élevé, il devient source de désordre au sein de la population. Une batterie de tests est donc indispensable pour déterminer la plage de taux de mutation la plus efficace pour l'algorithme génétique.
L'idée est que le meilleur individu dans la population en cours ait une chance de ne pas être impacté par des mutations qui pourraient s'avérer non favorables. Il existe donc une valeur critique à ne pas dépasser. Il est d'usage de respecter la règle suivante : le produit du nombre de gènes (bits si on code en Gray ou binaire), ici représenté par les positions de ville dans le circuit, par le taux de mutation par gène, doit être inférieur à 1. Dans notre cas, nous disposons de 49 gènes (pour les 49 positions dans le circuit). Il semblerait donc que le taux de 2% soit le taux critique à ne pas dépasser, sous peine de dégradation des résultats. Cela correspond au taux optimal que nous avons observé.

9.3.4 Etude de l'opérateur de sélection

C'est l'opérateur de sélection qui permet d'évaluer les solutions obtenues après application de la mutation et du croisement, en fonction de leur valeur pour le problème d'optimisation. Dans le problème qui nous occupe, chaque individu est un itinéraire et nous avons choisi comme fonction d'adaptation ou *fitness* la longueur totale de celui-ci. Nous cherchons à ce que l'opérateur de sélection désigne les individus de fitness minimum.

En matière d'opérateur de sélection, il y a différentes écoles : certains préfèrent la roulette, d'autres le tournoi, d'autres encore, utilisent la sélection élitiste (cfr. page 36.). Ces diverses méthodes ne conduisent pas aux mêmes résultats, selon le problème à optimiser. Pour notre part, nous avons employé deux méthodes stochastiques : la roulette et le tournoi.

Nous avons voulu tester la méthode du tournoi car nous pouvions ajuster la taille de celui-ci, c-à-d le nombre de participants au tournoi, et par là, ajuster la pression de sélection exercée sur la population. Que ce soit pour une grande population ou pour une petite population, le constat est le même : si la taille du tournoi augmente, les résultats se dégradent.

On peut le comprendre de façon intuitive : plus la pression est forte, moins les individus les moins adaptés sont sélectionnés. En effet, ce sont les meilleurs individus qui s'imposent lors de ces compétitions. Si le tournoi implique un nombre élevé d'individus, les individus les plus faibles ont moins de chance d'être sélectionnés car la probabilité qu'ils se trouvent en concurrence avec un des meilleurs individus de la population augmente.

Taille des tournois	*Long.Opt. [Km]*	*Temps [s]*
2	17929	93
5	20819	102
10	23887	112
15	23900	111
20	26206	109
50	34897	126

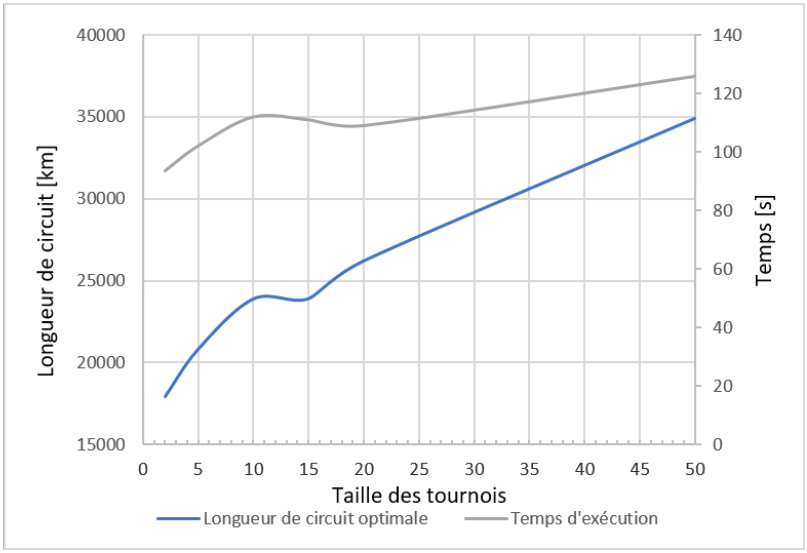


FIGURE 9.16 – Evolution de la longueur de circuit optimale en fonction du nombre de participants au tournoi.
Taille de la population N :100 - Nombre d'itérations : 200.

Taille des tournois	*Long.Opt. [Km]*	*Temps [s]*
2	21682	16
5	24725	18
10	29531	19
15	33211	20
20	35942	20
50	42144	23

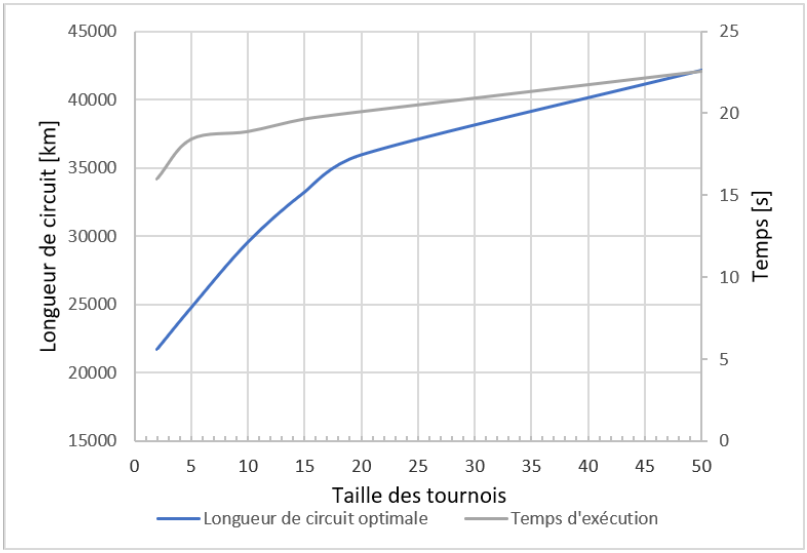


FIGURE 9.17 – Evolution de la longueur de circuit optimale en fonction du nombre de participants au tournoi.
Taille de la population N :200 - Nombre d'itérations : 200.

Taille des tournois	*Long.Opt. [Km]*	*Temps [s]*
2	24303	8
5	26933	9
10	32546	10
15	35710	10
20	39167	10
50	46763	11

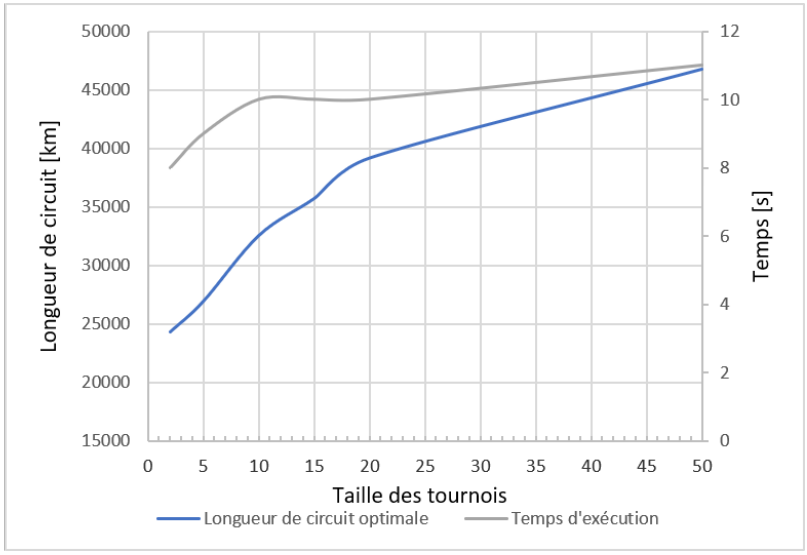


FIGURE 9.18 – Evolution de la longueur de circuit optimale en fonction du nombre de participants au tournoi.
Taille de la population N :1000 - Nombre d'itérations : 200.

Augmenter le nombre de concurrents dans un tournoi favorise donc la sélection des plus forts. On pourrait penser que c'est une excellente chose ; pourtant, cela peut avoir pour conséquence défavorable une convergence prématurée autour d'un optimum de mauvaise qualité. Il n'y a pas eu assez d'exploration dans la population : il ne faut pas oublier que les individus, moins performants, peuvent apporter de façon inattendue une amélioration à la situation en apportant une certaine variabilité dans la population.

Un aspect important des algorithmes génétiques est de conserver un équilibre entre exploitation des solutions déjà sélectionnées, car de bonne qualité, et exploration de nouvelles solutions. De façon très claire, la convergence est impactée par le nombre de participants au tournoi : c'est le duel qui l'emporte sur toutes les autres compétitions. Par ailleurs, nous avons observé une grande stabilité dans les résultats produits par le tournoi : on peut répéter plusieurs fois le processus, dans des conditions de paramétrage similaires, la solution finale fournie ne change pratiquement pas. On observe aussi que le temps de traitement augmente sensiblement avec la taille du tournoi.

Tous les résultats présentés dans les sections précédentes utilisaient la méthode de la roulette. Dans notre implémentation de cet opérateur, la probabilité d'être sélectionné est proportionnelle aux valeurs absolues du fitness : un individu 10 fois meilleur (présentant un parcours 10 fois plus court) a 10 fois plus de chance d'être sélectionné. Cela lui laisse l'occasion de dominer l'ensemble de la population et les individus les plus faibles. Il apparaît que la sélection par roulette soit mal adaptée à des cas de population à fort écart-type, car les individus les plus forts sont continuellement sélectionnés pour les générations suivantes. Ceci explique peut-être pourquoi nous avons constaté que cette méthode se montrait moins constante dans ses résultats.

Mais la roulette présente un avantage : nous avons observé qu'elle avait la capacité de construire une population de bonne composition. L'idée d'une méthode hybride, combinant la fabrication d'une bonne population globale grâce à la roulette, suivie d'un affinage de la solution par le tournoi pourrait peut-être représenter un bon compromis.

9.3.5 Etude de l'opérateur de croisement

Nous avons construit deux méthodes de croisement à locus simple. La première choisit le point de croisement sans restriction dans le chromosome : on sépare à un endroit quelconque l'itinéraire d'un parent en deux branches qui seront assemblées aux branches complémentaires de l'autre parent. La seconde méthode travaille également avec un locus simple mais elle ne permet pas un échange de chaîne de longueur inférieure à 3 allèles.

	Long. Opt. [km]	*Long. Moy. [km]*	*Temps [s]*
Crossover I 1 enfant	22870	27915	1107
Crossover I 2 enfants	22442	54190	602
Crossover II 1 enfant	24640	28916	1081
Crossover II 2 enfants	25623	55863	701

FIGURE 9.19 – Tableau présentant les résultats obtenus avec différents opérateurs de croisement. Taille de la population N :1000 - Nombre d'itérations : 200.

Le tableau 9.19 réalisé pour une population de 1000 individus évoluant sur 200 générations montre que les deux méthodes renvoient des résultats très similaires. La méthode 1 se montre peut-être légèrement plus efficace.

Ces deux méthodes ont été employées pour produire, tantôt un enfant unique, tantôt deux enfants. Les résultats obtenus pour ces deux versions, se valent en ce qui concerne la qualité de la solution fournie, avec cependant l'impression d'un léger avantage pour la méthode de l'enfant unique. On voit également que la population moyenne est de qualité moindre quand le croisement génère 2 enfants. La politique de l'enfant unique semble uniformiser la population.

Par contre, la production de 2 enfants présente l'avantage de temps gagné. En effet, la génération simultanée de 2 enfants divise pratiquement par 2 le temps nécessaire à l'exécution de l'algorithme.

Pour confirmer nos observations, il faudrait augmenter le nombre de tests réalisés et laisser la possibilité à la population d'évoluer sur plus de générations. On voit alors que les résultats sont beaucoup plus nuancés : la longueur optimale ne semble pas tellement affectée par le nombre d'enfants produits par croisement. Par ailleurs, on confirme bien que la première méthode de croisement, ne limitant pas la position du locus est à préférer à la deuxième.

Taille de la population N=100	*Long. Opt. [km]*	*Long. Moy. [km]*
Crossover I /1 enfant	23605	35005
Crossover I /2 enfants	22340	38742
Crossover II /1 enfant	25718	33720
Crossover II /2 enfants	24936	38735
Taille de la population N=200		
Crossover I /1 enfant	22198	33998
Crossover I /2 enfants	20522	37050
Crossover II /1 enfant	24687	30787
Crossover II /2 enfants	23794	39027
Taille de la population N=1000		
Crossover I /1 enfant	18664	34731
Crossover I /2 enfants	19076	37446
Crossover II /1 enfant	22612	31188
Crossover II /2 enfants	22869	37800

FIGURE 9.20 – Tableau présentant les résultats obtenus avec différents opérateurs de croisement. Nombre d'itérations : 1000.

9.4 Notre solution au problème

Après avoir tiré des enseignements de ces évaluations, nous avons opté pour le paramétrage suivant :

- **Taille de la population** : 200 individus.
- **Nombre d'itérations** : 2000.
- **Taux de mutation** : 2%.
- **Méthode de sélection** : tournoi sous forme de duel.
- **Méthode de croisement** : crossover I générant 1 enfant.

La figure 9.21 illustre une des solutions optimales au problème. Nous disons bien *une parmi d'autres* solutions optimales, puisque nous n'avons jamais la certitude que l'AG nous offre la solution absolue. En faisant tourner plusieurs fois l'AG, nous avons obtenu des distances du même ordre de grandeur, mais présentant de légères modifications dans le parcours.

Néanmoins, nous sommes satisfaits du résultat. La population initiale générée à l'aveugle, de façon aléatoire, par l'AG, présentait un itinéraire optimum de 84 254 km. Après 2000 générations, la longueur minimale du circuit est descendue à 16 371 km, avec une moyenne dans la population de 30 618 km, ce qui représente une amélioration significative des résultats. Le temps mis pour obtenir cette valeur a été de 144 s.

La comparaison de nos résultats à ceux obtenus par Dantzig et son équipe en 1954 nous encourage également. Pour rappel, à l'époque, ils obtiennent une distance optimale de 19 867 km. Leur tour vainqueur a la même allure que le nôtre, mais présente quelques petites différences, surtout dans la façon de rejoindre les villes situées plus au centre du territoire américain. N'oublions pas que leur problème comprend une dimension dont nous n'avons pas tenu compte : le circuit de 1954 utilise les voies terrestres. Les distances sont calculées en se référant aux routes existantes. De notre côté, nous nous sommes contentés de travailler avec les coordonnées GPS et donc, les distances renseignées sont celles mesurées à vol d'oiseau. Cela explique notamment la différence de presque 3 500 km à notre avantage.

1. *Washington, D.C.*
2. *Richmond en Virginie*
3. *Raleigh en Caroline du Nord*
4. *Columbia en Caroline du Sud*
5. *Jacksonville en Floride*
6. *Atlanta en Géorgie*
7. *Birmingham en Alabama*
8. *New Orleans en Louisiane*
9. *Jackson au Mississippi*
10. *Memphis au Tennessee*
11. *Little Rock en Arkansas*
12. *Dallas au Texas*
13. *Oklahoma City en Oklahoma*
14. *Cheyenne au Wyoming*
15. *Denver au Colorado*
16. *Santa Fe au Nouveau-Mexique*
17. *Phoenix en Arizona*
18. *Los Angeles en Californie*
19. *Carson City au Nevada*
20. *Portland dans l'Oregon*
21. *Seattle dans l'état de Washington*
22. *Boise dans l'Idaho*
23. *Salt Lake City dans l'Utah*
24. *Helena au Montana*
25. *Bismarck au Dakota du Nord*
26. *Pierre au Dakota du Sud*
27. *Omaha au Nebraska*
28. *Topeka au Kansas*
29. *Kansas City au Missouri*
30. *Des Moines en Iowa*
31. *Minneapolis au Minnesota*
32. *Milwaukee au Wisconsin*
33. *Chicago en Illinois*
34. *Indianapolis en Indiana*
35. *Louisville au Kentucky*
36. *Charleston en Virginie-Occidentale*
37. *Cleveland en Ohio*
38. *Detroit au Michigan*
39. *Montpelier dans le Vermont*
40. *Portland dans le Maine*
41. *Manchester dans le New Hampshire*
42. *Boston dans le Massachusetts*
43. *Providence dans l'état de Rhode Island*
44. *Hartford dans le Connecticut*
45. *New York dans l'Etat de New York*
46. *Newark au New Jersey*
47. *Philadelphia en Pennsylvanie*
48. *Wilmington au Delaware*
49. *Baltimore dans le Maryland*

FIGURE 9.21 – Tour optimal produit par notre AG pour visiter une unique fois les 49 villes du circuit choisi en revenant au point de départ.

Conclusion

A l'issue de ce travail, nous prenons conscience du fabuleux potentiel des algorithmes génétiques. Leurs applications sont multiples : optimisation de fonctions numériques difficiles, traitement d'image, optimisation d'itinéraires et d'emplois du temps, design industriel,...
Beaucoup d'aspects nous ont séduits dans ces algorithmes.

Nous avons apprécié qu'à l'instar d'un grand nombre de technologies qu'on retrouve par exemple en médecine ou dans l'aéronautique, ils puisent leur inspiration dans la nature. Recréer artificiellement la théorie de l'évolution de Darwin pour améliorer un système, n'est-ce pas une idée géniale ?

Par ailleurs, la conception d'un AG est relativement simple : l'approche se veut générique et indépendante du problème traité. Aucune propriété de continuité ni de dérivabilité n'est nécessaire au bon déroulement de la méthode, seule la connaissance du fitness des individus suffit. La mise en place d'un algorithme génétique est donc assez aisée, quel que soit le problème à résoudre, même si, pour nous, novices de la programmation, cette aventure ne fut pas un long fleuve tranquille.

Grâce à sa rapidité d'exécution, l'AG remplit parfaitement les conditions de résolution de problèmes complexes tels le *Problème du Voyageur de Commerce*, qui, malgré la simplicité de son énoncé, est impossible à résoudre en un temps raisonnable avec des algorithmes déterministes. Nous vous avons montré qu'il était inimaginable de devoir tester toutes les solutions possibles d'un tel problème, même en disposant d'un supercalculateur.

Néanmoins, l'AG n'a pas que des avantages. La séduisante simplicité de son implémentation ne doit pas faire oublier que son objectif est l'efficacité. Or, fabriquer un AG efficace est une tâche difficile. Ainsi, nous avons plusieurs fois observé une convergence prématurée : la solution fournie ne correspond pas à l'optimum global, mais à un optimum local autour duquel l'algorithme a stagné. L'analyse des résultats, parfois trompeurs, invite donc à la prudence.
Il faut garder en permanence à l'esprit que l'AG ne renvoie pas systématiquement la solution parfaite : il est choisi par l'analyste parce qu'il représente le meilleur compromis entre temps de traitement et qualité de résultats. Il faut donc accepter que le principal atout de l'algorithme génétique devient aussi son principal défaut : si l'algorithme peut se contenter de chercher une solution à l'aveugle, le programmeur, lui, doit le diriger pour augmenter la qualité des résultats.

Notre propre expérience de l'AG et notre long cheminement composé de recherches, conceptions, déceptions et victoires, nous ont montré combien il était difficile de concevoir un bon AG. La clé de la puissance d'un AG réside dans le choix pertinent de bons paramètres pour les divers opérateurs (mutation, croisement, sélection, remplacement). Une longue phase de réflexion et de tests s'avère donc indispensable avant d'obtenir des résultats satisfaisants.
Par exemple, l'étude du taux de mutation, paramètre autant utile que complexe, a été riche d'enseignements. Ajoutant encore plus de hasard pour permettre la variabilité, la mutation garantit l'évolution permanente dans les populations générées. Cependant, en la forçant avec un taux trop élevé, elle devient source de désordre chaotique au sein de l'algorithme. Le dosage des paramètres, tels que le taux de mutation, repose sur une bonne analyse des causes d'échecs et pas mal de bon sens.

La construction de notre algorithme génétique, si elle n'a pas été une tâche facile, a certainement représenté une source d'amusement. Après de longues heures à comprendre le langage Python, il a fallu se lancer dans le travail et essayer de construire un semblant d'algorithme, ne faisant qu'essuyer échecs après échecs au début de l'aventure. Maintenant que nous pouvons prétendre avoir progressé dans la compréhension d'un tel algorithme, nous n'avons qu'un seul souhait : en découvrir toute l'étendue. Evidemment, force est de constater que notre algorithme est perfectible ! Malheureusement, le temps nous prend au dépourvu. Paramétrer finement l'algorithme de manière à ce que celui-ci démontre toute sa puissance, utiliser les routes existantes avec un kilométrage réel et non pas des distances à vol d'oiseau dans un souci de précision, utiliser l'algorithme sur d'autres parcours, plus ambitieux en nombre d'instances, sont autant d'améliorations que nous aurions aimé apporter.

Nous sommes convaincus que les AG sont promis à un bel avenir. Avec l'expansion incessante de l'informatique, les entreprises, les banques, la recherche et même les particuliers exigent des algorithmes de plus en plus performants. Les AGs ont de nombreux atouts pour répondre à cette demande. Nous avons d'ailleurs montré qu'ils prenaient une place grandissante dans notre quotidien sans qu'on s'en rende compte. Selon nous, ils ne pourront qu'évoluer positivement et l'intérêt que les scientifiques leur porteront ne pourra que grandir.

Nous aimerions vous faire part de ce que ce projet nous a apporté. Malgré nos souhaits d'optimisation de nos propres performances, nous ne sommes pas déçus car ce travail de belle ampleur et instructif aura ravi chacun d'entre nous. Plus encore, il nous a permis de nous dépasser et d'acquérir une méthode rigoureuse de travail. Nous avons tous emmagasiné de l'expérience durant cette année : expérience, inédite pour la plupart d'entre nous, de la programmation, expérience de l'autodidactisme, reposant sur la curiosité motivée par l'envie d'apprendre et de comprendre, et surtout expérience essentielle du travail en équipe. Sur ce dernier point, nous avons appris beaucoup : le travail en équipe, bien qu'il ne soit pas toujours facile à mener, est important pour un tel projet. Le fait d'interagir et d'opposer ses idées avec celles de ses condisciples fut en effet indispensable au bon déroulement et à l'amélioration du projet.

Terminons sur une note philosophique. Les algorithmes génétiques sont, par essence, non déterministes. C'est peut-être dans ce mécanisme de l'aléatoire qu'ils puisent leur principale force : les populations initiales, les croisements, les mutations, les modes de sélection relèvent d'une part de chance. Cela nous a fait réfléchir à l'importance que prend le hasard dans l'évolution du monde. Sans vouloir interférer dans le débat quantique qui opposa jadis Einstein à Bohr – « *Dieu ne joue pas aux dés.* » vs « *Mais qui êtes-vous, Albert Einstein, pour dire à Dieu ce qu'il doit faire ?* » -, nous avons constaté que le hasard, quand on n'en abuse pas, faisait bien les choses puisqu'il a permis dans le cas de notre AG de révéler des solutions inattendues. Les petits mondes virtuels créés dans les algorithmes génétiques démontrent que le hasard a bien son mot à dire dans les phénomènes naturels. Mieux, il contribue largement à la création d'individus meilleurs. Cet aspect hasardeux peut nous faire frémir dans la mesure où nous ne détenons pas le contrôle de l'imprévisible. Accepter que nous ne puissions pas toujours tout comprendre et tout maîtriser, n'est-ce pas accueillir avec bon sens nos propres limites ?

Bibliographie

[1] *La sélection naturelle et la sélection sexuelle*. Deboeck, 2007.

[2] *Fundamentals of the New Artificial Intelligence : Neural, Evolutionary, Fuzzy and More*, chapter 4. Springer, 2008.

[3] Jean-Marc Alliot and Nicolas Durand. Algorithmes génétiques. *Institut de Recherche en Informatique de Toulouse*, 2005.

[4] David Bau. Computation intelligence car evolution using box2d physics, -.

[5] Youssef Bokhabrine. Etude et comparaison d'algorithmes d'optimisation pour la reconstruction 3d par supershapes et r-fonctions. Master's thesis, Université de Bourgogne, 2006.

[6] Cyril Boussion, Pierre Guengant, and François Grenier. Génétique des populations : une loi mathématique. *extrait de http ://tpe709.free.fr/*, 2003.

[7] Corine Brucato. The traveling salesman problem. *University of Pittsburgh*, 2013.

[8] Pierre Collet. Optimisation stochastique Évolutionnaire. *Université de Strasbourg*, 2015.

[9] Damon Cook. Evolved and timed ants : Optimizing the parameters of a time-based ant system approach to the traveling salesman problem using a genetic algorithm. Master's thesis, New Mexico State University, 2000.

[10] William Cook. Traveling salesman problem extrait de www.math.uwaterloo.ca/tsp/, 2016.

[11] Gabriel Cormier. Systèmes intelligents. *Université de Moncton*, 2004.

[12] James Dalgety. The puzzle museum, 2017.

[13] Albert Dipanda. Les algorithmes génétiques. *Faculté des sciences de l'Université de Bourgogne, Département IEM*, 2004.

[14] Nikos Drakos. Data structures and algorithms. *University of Leeds*, 1998.

[15] Nicolas Durand. *Algorithmes génétiques et autres méthodes d'optimisation appliquées à la gestion de trafic aérien*. PhD thesis, INPT, 2004.

[16] Eric P. et Julien H. Algorithmes génétiques extrait de www.pobot.org/algorithmes-genetiques.html. *Club de Robotique de Sophia-Antipolis*, 2011.

[17] G.Dantzig, R. Fulkerson, and S.Johnson. Solution of a large-scale travelling-salesman problem. *The Rand Corporation*, 1954.

[18] Inconnu. Pourquoi l'avion vole ? in ptitgenie.com/sciences/pourquoi-avion-vole, 2014.

[19] Inconnu. Natural selection, 2016.

[20] Gilbert Laporte. A short history of the traveling salesman problem. *HEC Montréal*, 2006.

[21] Vincent Magnin. *Contribution à l'étude et à l'optimisation de composants optoélectroniques*. PhD thesis, Université des sciences et technologies de Lille, 1998.

[22] Vincent Magnin. Algorithmes évolutionnaires et algorithmes génétiques, 2006.

[23] Alexandre Mayer, Jérôme Muller, Aline Herman, and Olivier Deparis. *Optimized absorption of solar radiations in nano-structured thin films of crystalline silicon via a genetic algorithm*. PhD thesis, Laboratoire de Physique du Solide, Université de Namur, 2015.

[24] Didier Muller. Algorithmique.

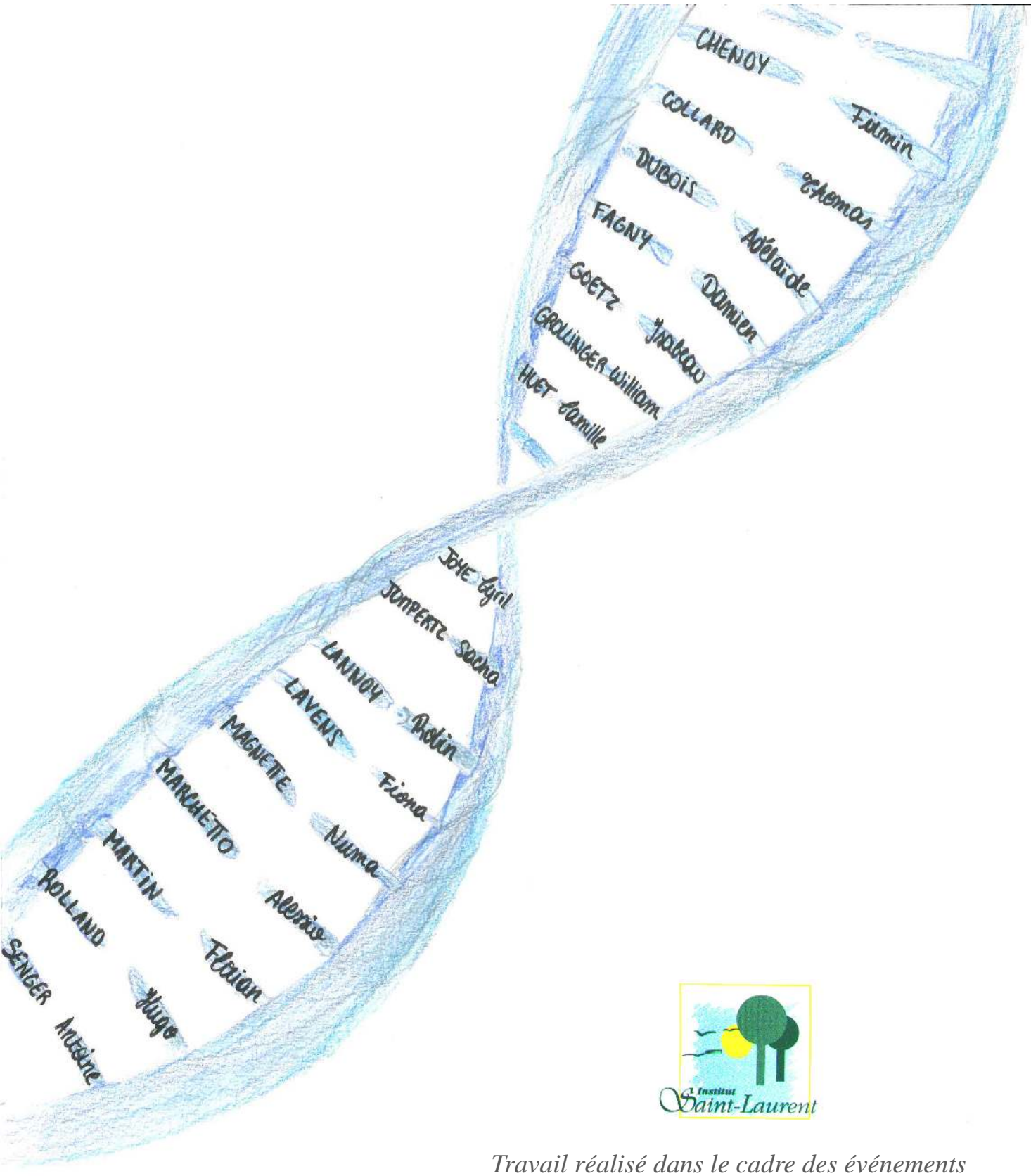
[25] Christelle Reynès. *Etude des Algorithmes génétiques et application aux données de protéomique*. PhD thesis, Université Montpellier I, 2007.

[26] Amanur Rahman Saiyed. The traveling salesman problem. *Indiana State University*, 2012.

[27] Christine Solnon. Résolution de problèmes combinatoires et optimisation par colonie de fourmis. *Institut National de Sciences Appliquées de Lyon*, 2015.

[28] Amédée Souquet and Francois-Gérard Radet. Algorithmes génétiques. *Travail de fin d'année*, 2004.

[29] Thomas Vallée and Murat Yildizoelu. Présentation des algorithmes génétiques et de leurs applications en économie. *Revue d'économie politique*, pages 711–745, 2004.



Institut Saint-Laurent

Travail réalisé dans le cadre des événements

DédraVMATHisons

WETENSCHAPS EXPO SCIENCES BRUXELLES