République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Yahia Farès de Médéa
Département de Mathématiques et Informatique

MÉMOIRE

présenté en vue de l'obtention du

DIPLÔME DE MASTER EN MATHÉMATIQUES APPLIQUÉES

Spécialité : Recherche Opérationnelle

par

BABAALI Soumia

L'optimisation des problèmes mathématiques par l'utilisation des techniques d'intelligence artificielle

Soutenu publiquement le 24 juin 2025 devant le jury composé de

Président :	TOUMACHE Kamel	*M.C.B*	Université de Médéa
Examinateur :	BIBI Hamza	*M.C.B*	Université de Médéa
Encadreur :	HAYI Mohamed Yassine	*M.C.B*	Université de Médéa
Co-Encadreur :	BENTERKI Abdessalem	*M.C.A*	Université de Médéa

Année universitaire : 2024-2025

Table des matières

Introduction **1**

1 État de l'art **3**
1.1 Définitions et notations de base 3
1.2 Types de graphes 5
1.3 La coloration dans les graphes 14
1.4 Propriétés et algorithmes fondamentaux 17
1.5 Représentation des graphes en mémoire 19
1.6 Représentation des graphes par l'utilisation de python 23
1.7 Complexité algorithmique dans les graphes 23

2 Travaux connexes **25**
2.1 Introduction aux techniques d'intelligence artificielle en optimisation mathématique 25
2.2 Algorithmes classiques d'optimisation 26
2.3 Intelligence artificielle et optimisation : approches et algorithmes 27
2.4 Applications de l'IA dans la résolution de problèmes de graphes 33
2.5 État de l'art sur l' optimisation du chemin le plus court 33
2.6 Différents types d'optimisation 35

3 Méthodologie **39**
3.1 Motivation et objectifs 39
3.2 Choix des algorithmes d'IA à étudier 40
3.3 Description détaillée des algorithmes sélectionnés 41
3.4 Méthode de validation et de mesure des performances 63

4 Résultats et Discussions **65**
4.1 Environnement de développement 65
4.2 Description des données 67
4.3 Evaluation des algoritmes 68
4.4 Comparaison entre les trois approches 79
4.5 Discussion générale des résultats obtenus 81

4.6 Présentation de l'interface 82

Conclusion et perspective **86**

Bibliographie **88**

Table des figures

1.1 Graphe orienté 5
1.2 Graphe non orienté 6
1.3 Graphe pondéré 6
1.4 G_A est sous-graphe de G et G_B est sous-graphe induit de G. 7
1.5 Exemple d'un graphe G avec son graphe partiel. 7
1.6 Un graphe simple. 7
1.7 Le graphe complet (K_4) et le graphe discret ($\overline{K_4}$). 8
1.8 Un graphe 3-régulier d'ordre 6. 8
1.9 Le cycle C_8 et la chaine P_5. 9
1.10 Graphe connexe. 9
1.11 Un exemple d'un arbre T. 10
1.12 Le graphe biparti complet $k_{3,3}$. 10
1.13 Un graphe biparti 3-semi-régulier. 10
1.14 L'étoile $K_{1,q}$ et L'étoile subdivisée $K^*_{1,q}$. 11
1.15 Double étoile $S_{r,s}$. 11
1.16 La couronne d'un C_6. 11
1.17 Le graphe joint de P_2 et P_3. 12
1.18 Le cycle C_5 et son complémentaire $\overline{G}$. 12
1.19 Le graphe $C_4 + e$. 13
1.20 Le graphe K_3. 13
1.21 Un multigraphe avec 4 sommets 13
1.22 Coloration propre du graphe $G = K_4 - e$ avec $\chi(G) = 3$ 14
1.23 Coloration propre du graphe complet K_6 avec $\chi(K_6) = 6$. 15
1.24 Coloration propre du graphe discret $\overline{K_5}$ avec $(\chi(\overline{K_5}) = 1)$. 15
1.25 Coloration propre du cycle pair C_4 avec $\chi(C_4) = 2$. 15
1.26 Coloration propre du cycle impair C_5 avec $\chi(C_5) = 3$. 16
1.27 Coloration propre du graphe P_6 avec $\chi(P_6) = 2$. 16

4.1 Carte de répartition du Dataset 68
4.2 Histogramme de répartition des cours par créneau de l'algorithme force brute 71
4.3 Graphe avant coloration généré à partir du Dataset 72

4.4 Exemple détaillé d'un noeud du graphe 72
4.5 Graphe après coloration 73
4.6 Histogramme de répartition des cours par créneau de l'algorithme de coloration avec heuristiques 75
4.7 Courbes d'évolution de l'algorithme génétique 77
4.8 Courbe de convergence de la fitness 78
4.9 Histogramme de répartition des cours par créneau avec l'algorithme génétique 79
4.10 Onglet "Acceuil" de l'interface 82
4.11 Onglet "Configuration des Modules" de l'interface 83
4.12 Onglet "Modification" de l'interface 84
4.13 Onglet "Emploi du temps" lors de la génération 85
4.14 Onglet "Emploi du temps" après la génération 85
4.15 Onglet "Visualisation PDF" 86

Liste des tableaux

4.1 Caractéristiques des environnements d'exécution 65
4.2 Bibliothèques Python utilisées et leurs descriptions 66
4.3 Paramètres utilisés dans l'algorithme génétique 76
4.4 tableau comparatif entre les trois approches 80

Remerciements

Tout d'abord, je tiens à remercier Dieu, le tout puissant, de m'avoir donné la volonté, la force, la patience et le courage afin de réaliser ce modeste travail.

Je remercie mes parents pour leur bienveillance, leur affection et leur soutien inconditionnel, ainsi, leur générosité et tendresse, étant donné qu'ils m'ont motivées tout au long de ces années pour obtenir mon diplôme.

J'exprime ma profonde gratitude à mon promoteur, Dr. HAYI Mohamed Yassine, et à mon co-promoteur, Dr. BENTERKI Abdessalem, pour leurs précieux conseils, leur disponibilité et leur aide. Je les remercie également pour leur accompagnement, leur patience et leurs orientations.

J'adresse mes remerciements aux membres de jury, Dr. TOUMACHE Kamel, et, Dr. BIBI Hamza, pour leur présence, leur lecture attentive de ce mémoire, et pour les remarques qu'ils m'adresseront lors de cette soutenance afin d'améliorer mon travail.

Je tiens à remercier également du fond du cœur mes chères sœurs et mon petit frère qui ont toujours été à mes côtés, pour leurs soutiens et leurs encouragements.

Finalement, j'ai envie de remercier ma famille et tous ceux et celles qui ont participé de près ou de loin à mon projet.

Dédicace

Ce modeste travail est dédié :

À mes chers parents :

"Aucun hommage ne pourrait être à la hauteur de l'amour dont ils ne cessent de me combler et de leurs sacrifices. Que Dieu leur procure bonne santé, bonheur et longue vie. Vous avez toujours été mes piliers, votre soutien inconditionnel a été ma force tout au long de ce parcours, et je suis infiniment reconnaissante de vous avoir à mes côtés, sans vous, ce projet n'aurait pas été le même. Vous avez fait partie intégrante de mon voyage."

À mes chères sœurs et mon cher frère :

"Je vous aime énormément. Vous avez toujours été mes confidents et ma source d'inspiration, vous m'avez apportée un soutien inestimable, des conseils précieux et un réconfort indéfectible. Vos encouragements et votre présence ont illuminé mes journées les plus difficiles."

À ma sœur bien-aimée Ihssane :

"Pour ta patience, ton soutien indéfectible, ton écoute et tes encouragements, Merci de m'avoir toujours crue capable, même dans les moments de doute. Ce mémoire porte aussi ta trace."

"À ceux et celles qui, de près ou de loin, m'ont apportée soutien, conseils et réconfort."

BABAALI Soumia

Résumé

L'intelligence artificielle s'impose comme un outil puissant pour résoudre des problèmes complexes dans des domaines aussi variés que la médecine, le commerce ou encore les mathématiques. Grâce à ses algorithmes avancés, elle permet d'automatiser des tâches, d'analyser des données massives et de trouver des solutions optimales en un temps réduit.

Dans le cadre de notre mémoire, nous avons choisi de nous intéresser à un problème concret et fréquent dans le milieu universitaire : la génération automatique d'un emploi du temps sans conflit. Ce problème, bien qu'en apparence simple, s'avère en réalité très complexe en raison des nombreuses contraintes à respecter (groupes, enseignants, salles, créneaux, etc.).

Pour y faire face, nous avons mobilisé la théorie des graphes et plusieurs algorithmes d'intelligence artificielle, notamment des approches heuristiques et méta-heuristiques telles que l'algorithme génétique et d'autres techniques d'optimisation. L'objectif était non seulement de produire des emplois du temps valides, mais aussi de comparer les performances de ces différentes méthodes afin d'en évaluer l'efficacité.

Mots clés : Intelligence artificielle, Problème mathématique, Optimisation, Graphe, Coloration, Algorithme génétique, Heuristique et Méta-heuristiques.

Abstract

Artificial intelligence has emerged as a powerful tool for solving complex problems in diverse fields such as medicine, commerce, and mathematics. Thanks to its advanced algorithms, it enables the automation of tasks, the analysis of large-scale data, and the identification of optimal solutions in a short amount of time.

In the context of our dissertation, we chose to focus on a concrete and recurring issue in the academic environment : the automatic generation of conflict-free timetables. Although seemingly simple, this problem is in fact highly complex due to the numerous constraints that must be taken into account (groups, instructors, rooms, time slots, etc.).

To address this challenge, we drew upon graph theory and various artificial intelligence algorithms, particularly heuristic and metaheuristic approaches such as the genetic algorithm and other optimization techniques. The objective was not only to produce valid timetables but also to compare the performance of these different methods in order to evaluate their effectiveness.

Keywords : Artificial Intelligence, Mathematical Problem, Optimization, Graph, Coloring, Genetic Algorithm, Heuristics and Metaheuristics.

ملخص

يبرز الذكاء الاصطناعي كأداة قوية لحل المشاكل المعقدة في مجالات متنوعة مثل الطب والتجارة والرياضيات. وبفضل خوارزمياته المتقدمة، يمكنه تنفيذ المهام آلياً وتحليل كميات هائلة من البيانات وإيجاد الحلول المثلى في فترة زمنية قصيرة.

لأغراض أطروحتنا، اخترنا التركيز على مشكلة ملموسة كثيراً ما نواجهها في البيئة الجامعية: إنشاء جدول زمني خالٍ من التعارضات بشكل أوتوماتيكي. هذه المشكلة، على الرغم من أنها تبدو بسيطة، إلا أنها في الواقع معقدة للغاية بسبب القيود العديدة التي يجب احترامها (المجموعات، والمدرسين، والقاعات، والفترات الزمنية، وما إلى ذلك).

للتعامل مع هذه المشكلة، قمنا بتعبئة نظرية الرسم البياني والعديد من خوارزميات الذكاء الاصطناعي، ولا سيما الأساليب الاستدلالية والتحويلية مثل الخوارزمية الجينية وغيرها من تقنيات التحسين. لم يكن الهدف هو إنتاج جداول زمنية صالحة فحسب، بل أيضًا مقارنة أداء هذه الأساليب المختلفة من أجل تقييم فعاليتها.

الكلمات المفتاحية : الذكاء الاصطناعي، المشكلة الرياضية، التحسين، الرسم البياني، التلوين، الخوارزمية الجينية، خوارزمية استكشافية و خوارزمية فوق استكشافية .

Liste des abréviations

IA :	Intelligence Artificielle.
AG :	Algorithme Génétique.
ACO :	Ant Colony Optimization.
PL :	Programmation Linéaire.
QP :	Programmation Quadratique.
TD :	Travaux Dirigés.
TP :	Travaux Pratiques.
Id :	Identifiant.
DEAP :	Distributed Evolutionary Algorithms in Python.

Introduction

L'élaboration des emplois du temps dans le cadre universitaire représente une mission à la fois délicate et fondamentale, en raison de la multitude de contraintes qu'il convient de prendre en compte. Ces contraintes peuvent être d'ordre pédagogique (comme les préférences horaires, ou encore le regroupement de certaines séances), logistique (comme la gestion des salles ou des équipements requis), ou organisationnel (éviter les chevauchements d'horaires, répartir équitablement les charges hebdomadaires, etc.). Avec l'augmentation continue du nombre d'étudiants, la diversification des formations et l'élargissement des programmes d'enseignement, cette tâche devient de plus en plus ardue, rendant les approches traditionnelles, qu'elles soient manuelles ou partiellement automatisées, peu efficaces, coûteuses en temps et sujettes à des erreurs.

Dans ce contexte, le recours à des approches informatiques avancées s'impose comme une nécessité. L'objectif est de concevoir des systèmes capables de produire des plannings optimaux tout en respectant l'ensemble des exigences fixées. C'est dans cette perspective que s'inscrit notre projet, dont la finalité est de mettre en place une solution fiable et performante, reposant sur l'intelligence artificielle, apte à générer automatiquement des emplois du temps cohérents, équilibrés et adaptés aux contraintes du milieu universitaire.

Notre approche s'inscrit à la croisée de deux domaines complémentaires : la recherche opérationnelle, qui fournit les outils d'analyse et de modélisation des problèmes complexes, et l'intelligence artificielle, qui apporte des techniques puissantes pour explorer et optimiser efficacement l'espace des solutions. Cette synergie entre RO et IA constitue un cadre méthodologique solide, particulièrement adapté à la résolution des problèmes d'affectation et de planification.

Pour y parvenir, nous avons opté pour l'utilisation d'algorithmes heuristiques et métaheuristiques, reconnus pour leur aptitude à traiter efficacement les problèmes d'optimisation combinatoire. Les méthodes heuristiques permettent d'obtenir rapidement des solutions satisfaisantes dans de vastes espaces de recherche, tandis que les métaheuristiques offrent une exploration plus approfondie, permettant d'éviter les pièges des optima locaux. Ce choix se justifie par leur adaptabilité, leur robustesse et leur rendement élevé, qui constituent des atouts majeurs pour répondre aux exigences réelles de la gestion des

emplois du temps.

À travers ce travail, notre ambition est double : d'une part, fournir un outil technologique puissant, et d'autre part, participer à l'amélioration de la gestion du temps et des ressources humaines et matérielles au sein des institutions d'enseignement supérieur. L'objectif est également d'alléger la charge liée à la planification manuelle, tout en garantissant une meilleure coordination entre les différents intervenants (enseignants, étudiants, personnel administratif).

La structure de ce mémoire sera organisée comme suit :

Le **premier chapitre** est dédié à la théorie des graphes, en exposant ses fondements, ses notions clés, sa terminologie ainsi que les principaux algorithmes utilisés.

Le **deuxième chapitre** aborde les concepts de l'intelligence artificielle, avec une attention particulière portée aux heuristiques et métaheuristiques, ainsi qu'à leur rôle dans la résolution de problèmes d'optimisation liés aux graphes.

Le **troisième chapitre** traite de manière conceptuelle nos approches algorithmiques, en détaillant leur logique de fonctionnement ainsi que les indicateurs utilisés pour en mesurer la qualité.

Le **quatrième chapitre** présente la mise en œuvre pratique des algorithmes développés et propose une analyse de leurs performances sur différents scénarios.

CHAPITRE 1

État de l'art

Dans ce chapitre, nous allons introduire les concepts fondamentaux de la théorie des graphes, en définissant les principales notions et notations utilisées dans ce domaine. Nous présenterons ensuite les différents types de graphes et leurs caractéristiques. Une attention particulière sera portée aux propriétés essentielles et aux algorithmes fondamentaux, notamment ceux liés aux parcours dans les graphes et à la recherche de chemins optimaux.

Section 1.1

Définitions et notations de base

Graphes

Un graphe est défini par un couple $G = (V, E)$, où V représente un ensemble fini non vide de sommets et E un ensemble fini de relations structurées entre ces sommets. La nature de ces relations détermine deux catégories fondamentales : **les graphes orientés**, où E est composé de couples ordonnés (appelés **arcs** notées par $< u, v >$ ou $\overrightarrow{uv}$), et les **graphes non orientés**, où E est constitué de paires non ordonnées (appelées **arêtes** notées par (u, v) ou uv)[7].

Voici certaines notions et définitions qui seront employées tout au long de ce mémoire :

- **L'ordre d'un graphe** G est le nombre de ses sommets, que l'on note $| V |$.
- **La taille d'un graphe** G est le nombre de ses arêtes, que l'on note $m(G)$ ou $| E |$.
- Une arête (u, v) d'un graphe G est **une boucle** si les extrémités u et v coïncident.

- Deux sommets u et v sont dits **adjacents** si l'arête (u,v) ou (v,u) existe dans E, c'est-à-dire, si les deux sommets sont reliés par une arête.
- Deux arêtes sont **adjacentes** si elles ont une extrémité en commun.
- Une arête est **incidente** à un sommet si ce sommet est une extrémité de cette arête.

Voisinage et degré d'un sommet

Soit $G=(V,E)$ un graphe et v un sommet de G.

- Le **voisinage ouvert** de v est : $N_G(v) := \{u \in V : (u,v) \in E\}$.
- Le **voisinage fermé** de v est défini par : $N_G[v] := N_G(v) \cup \{v\}$.
- Pour un sous-ensemble $S \subseteq V(G)$, **le voisinage ouvert** de S est : $N_G(S) = \bigcup_{v \in S} N_G(v)$.
- Le **voisinage fermé** de S est :$N_G[S] := \bigcup_{v \in S} N_G[S]$. [10]

Pour G un graphe simple, on a :

- **Le degré d'un sommet** v est le nombre d'arêtes qui lui sont incidentes qui est la taille de son voisinage. Soit $v \in V$, le degré de v est noté par :

$$d_G(v) := |N_G(v)|.$$

Remarque 1 *On appelle **sommet isolé** un sommet de degré* 0. *Un sommet de degré* 1 *est appelé **sommet pendant** (ou **feuille**). L'unique sommet qui est adjacent à un sommet pendant est appelé **sommet support**.*

- **Le degré maximum** d'un graphe est le degré maximum de tous ses sommets, noté par :

$$\Delta(G) = \max\{d_G(v) : v \in V\}.$$

- **Le degré minimum** d'un graphe est le degré minimum de tous ses sommets, noté par :

$$\delta(G) = \min\{d_G(v) : v \in V\}.$$

Une propriété fondamentale du degré d'un graphe connue sous le nom de lemme des poignées de main (Handshaking Lemma) trouve ses origines dans les travaux de Leonhard Euler sur le problème des sept ponts de Königsberg, comme détaillé ci-après.

Corollaire 1.1 [7] *Soit $G=(V,E)$ un graphe simple, avec $V=\{v_1, v_2, ..., v_n\}$. Alors on a :*

$$\sum_{i=1}^{n} d_G(v_i) = 2m(G). \quad i = \{1...n\}$$

Preuve.

Une approche alternative, utilise une induction sur le nombre d'arêtes :

1. Pour un graphe sans arêtes ($|E| = 0$), tous les degrés sont nuls, donc

$$\sum_{i=1}^{n} d_G(v) = 0 = 2 \times 0 = 2m(G)$$

.

2. Supposons la propriété vraie pour tout graphe avec k arêtes. Ajoutons une arête $e = (u, v)$. Les degrés de u et v augmentent chacun de 1, donc la somme totale augmente de 2, préservant l'égalité

$$\sum_{i=1}^{n+1} d_G(v) = \sum_{i=1}^{n} d_G(v_i) + d_G(v_{n+1}) = 2k + 2 = 2(k+1).$$

Cette construction itérative confirme que la propriété se maintient pour tout nombre d'arêtes.[13] ■

Section 1.2

Types de graphes

Graphe orienté

Définition 1.1 *Un graphe orienté $G = (V, U)$ est défini par les deux ensembles [5] :*

- *$V = \{v_1, v_2, \ldots, v_n\}$ avec n entier fini et $n \geq 1$, où chaque $v_i \in V$ est **un sommet** du graphe.*

- *$U = \{u_1, u_2, \ldots, u_m\}$ avec m entier positif ou nul et fini, est appelé ensemble **des arcs**. Chaque $u_j \in U$ est une paire ordonnée de sommets. $u_j = (x, y)$: x est appelé extrémité initiale de u_j et y est appelé extrémité terminale de u_j, U peut être vide.*

La Figure 1.1 illustre un graphe orienté avec 4 sommets :

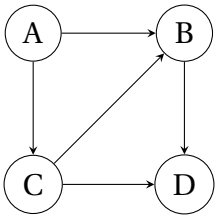


FIGURE 1.1 – Graphe orienté

Graphe non orienté

Définition 1.2 *Un graphe non orienté $G=(V,E)$ est défini par les deux ensembles [5] :*

- $V=\{v_1,v_2,\ldots,v_n\}$ *avec n entier fini et $n\geq 1$, où chaque $v_i\in V$ est* ***un sommet*** *du graphe.*

- $E=\{e_1,e_2,\ldots,e_m\}$ *avec m entier positif ou nul et fini, est appelé ensemble* ***des arêtes****. Chaque $e_j\in E$ est une paire non ordonnée de sommets. $e_j=(x,y)=(y,x)$, x et y sont appelés extrémités de e_j. E peut être vide.*

Le graphe de la Figure 1.2 possède cinq sommets et six arêtes qui sont $V(G)=\{x,y,z,u,v\}$, $E(G)=\{xy,yz,zx,zu,xu,uv\}$.

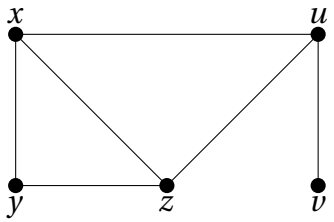


FIGURE 1.2 – Graphe non orienté

Graphe pondéré

Chaque arête est associée à un poids, représentant par exemple une distance ou un coût. (Voir Figure 1.3)

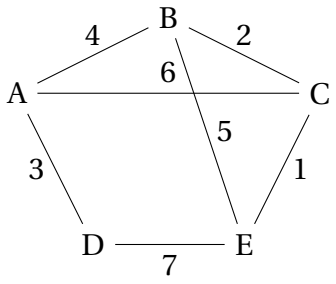


FIGURE 1.3 – Graphe pondéré

Sous-graphe et sous-graphe induit

Le graphe H est dit sous-graphe d'un graphe G lorsque $V(H)\subseteq V(G)$ et $E(H)\subseteq E(G)$. Un sous-graphe H de G est dit induit si $E(H)$ contient toutes les arêtes de G dont les deux extrémités sont dans $V(H)$. Pour définir un sous-graphe de G il suffit de considérer un sous-ensemble $A=\{x_1,x_2,x_4\}$ de V, le sous-graphe sera désigné par $G[A]$. (Voir Figure

1.4). Pour définir un sous-graphe induit de G il suffit de considérer un sous-ensemble $B = \{x_1, x_3, x_4\}$ de V. En outre, ce sous-graphe induit par B sera désigné par $G[B]$. On dira aussi que $B \subseteq V(G)$ induit le sous-graphe correspondant. (Voir Figure 1.4). Soit v un sommet de V. On notera $G\text{-}v$ le sous-graphe $G[V - \{v\}]$ [29].

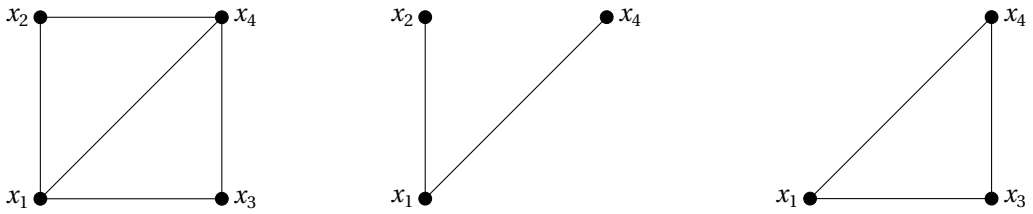


FIGURE 1.4 – G_A est sous-graphe de G et G_B est sous-graphe induit de G.

Graphe partiel

Soit $G = (V, E)$ un graphe, le graphe $G = (V, E')$ est un graphe partiel de G si $E' \subseteq E$ [29]. Autrement dit,on obtient G' en enlevant une ou plusieurs arêtes au graphe G. Le graphe de la Figure 1.5 représente un graphe G et son graphe partiel.



FIGURE 1.5 – Exemple d'un graphe G avec son graphe partiel.

Graphe simple

Un **graphe simple** est un graphe si l'on a les deux conditions :

1. i1 n'a pas de boucles,
2. entre deux sommets, il n'y a jamais plus d'un arc pour les relier[14].(Voir Figure 1.6)



FIGURE 1.6 – Un graphe simple.

Graphe complet

Un graphe complet est un graphe où chaque sommet est adjacent à tous les autres sommets. Le graphe complet d'ordre n est noté K_n. Dans ce graphe chaque sommet est de degré $n-1$ (Voir Figure 1.7).

Graphe discret

On dit qu'un graphe G est discret ou nul s'il ne contient aucune arête, c'est-à-dire que $E=\varnothing$ et noté $\overline{K_n}$ (Voir Figure 1.7).

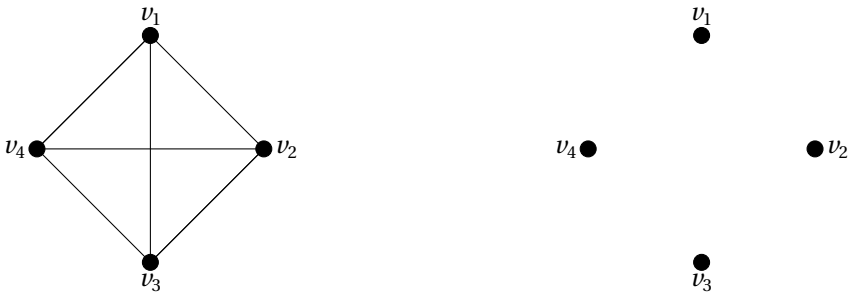


FIGURE 1.7 – Le graphe complet (K_4) et le graphe discret ($\overline{K_4}$).

Graphe k-régulier

Un graphe G est dit k-régulier si tous ses sommets sont de degré k, c'est-à-dire, $d_G(x)=k$ pour tout $x \in V$ (Voir Figure 1.8).

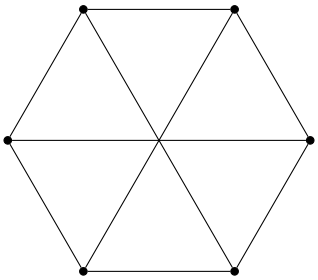


FIGURE 1.8 – Un graphe 3-régulier d'ordre 6.

Chaînes

Soit un graphe $G=(V,E)$.

- **une chaîne** P_k de G est une séquence finie de sommets $\{v_1, v_2, ..., v_k\}$, où pour tout $i=1,...,k,\{v_i, v_{i+1}\}$ est une arête de G. Les sommets v_1 et v_k sont les extrémités de la chaîne P_i et les sommets v_2 ,,..., v_{k-1} sont les sommets intérieursde P_k.
- **Une chaîne** P_k **est simple** si chaque arête de P_k est emprunté une seule fois.
- **Une chaîne** P_k **est élémentaire** si elle ne passe pas deux fois par le même sommet.

Cycles

- **Un cycles** C_k d'un graphe $G=(V,E)$ est une chaîne $\{v_1, v_2, \ldots, v_k\}$ telle que les extrémités sont confondues $v_1 = v_k$.
- **Un cycle élémentaire** est une chaîne élémentaire telle que ses deux extrémités sont confondues. Un cycle C_k est **simple** si chaque arête de C_k est emprunté une seule fois.

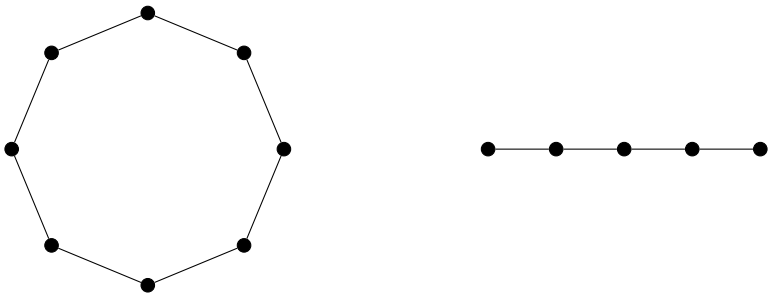


FIGURE 1.9 – Le cycle C_8 et la chaine P_5.

Connexité

- Un graphe $G=(V,E)$ est **un graphe connexe** s'il existe une chaîne entre toutes les paires de sommets du graphe.
- Un graphe connexe a n sommets posséde au moins $n-1$ arêtes.
- une composantes connexe d'un graphe G est un sous-graphe induit maximal connexe de sommets.
- Une arête e est appelée **un isthme** de G si le graphe $G-e$ possède plus de composantes connexes que G.
- Un sommet v du graphe G est appelé un sommet d'articulation du graphe G si le graphe $(G-v)$ a plus de composantes connexes que G.
- dans le graphe de la Figure 1.10 les sommets a, d, f sont des sommets d'articulation et l'arête a, d est un isthme.

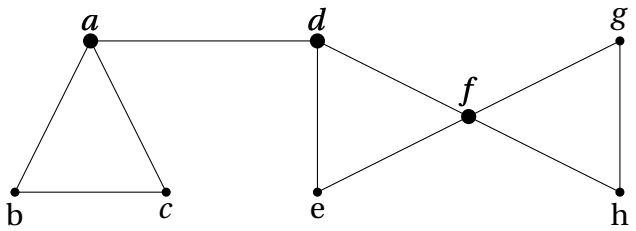


FIGURE 1.10 – Graphe connexe.

Arbre

Un arbre est un graphe connexe sans cycle, il contient exactement $(n-1)$ arête et noté T. Un graphe sans cycle mais non connexe est appelé forêt. (Voir Figure 1.11)

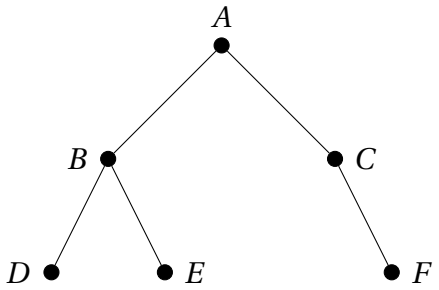


FIGURE 1.11 – Un exemple d'un arbre T.

Graphe biparti

Un graphe biparti est un graphe G d'ordre $n \geq 2$, dont l'ensemble des sommets est partitionné en deux stables X et Y . Un graphe biparti complet est un graphe biparti maximal en terme de nombre d'arêtes, alors $| E | = | X | . | Y |$, noté $K_{p,q}$ avec $p = | X |$ et $q = | Y |$ (Voir Figure 1.12).

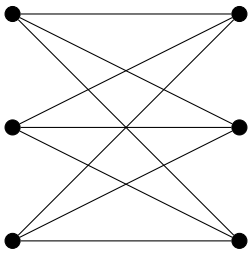


FIGURE 1.12 – Le graphe biparti complet $k_{3,3}$.

Graphe biparti k-semi-régulier

Un graphe biparti k-semi-régulier est un graphe G d'ordre $n \geq 2$, dont l'ensemble des sommets est partitionné en deux stables X et Y, avec chaque sommet de l'une du deux parties est de degré k (Voir Figure 1.13).

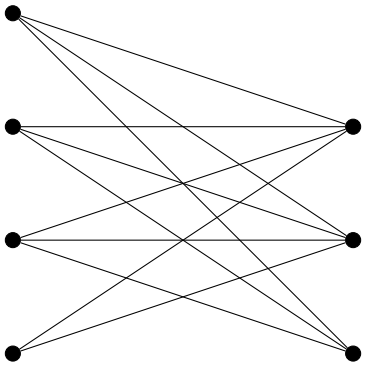


FIGURE 1.13 – Un graphe biparti 3-semi-régulier.

Étoile

Une étoile est un cas particulier d'un graphe biparti complet dans lequel $| V_1 | = 1$ et $| V_2 | = q$. Elle est notée par $K_{1,q}$. Le sommet de V_1 est appelé centre ou sommet central de l'étoile. Une étoile subdivisée est une étoile dont chaque arête est subdivisée une fois (Voir Figure 1.14). Nous noterons $K^*_{1,q}$ l'étoile subdivisée obtenue à partir de l'étoile $K_{1,q}$.

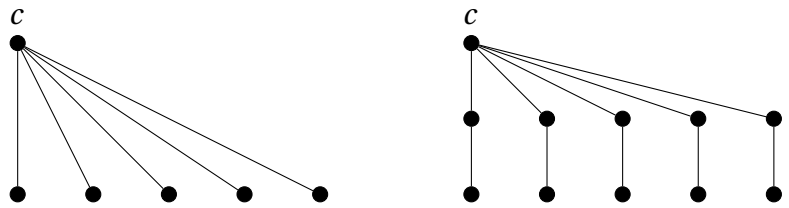


FIGURE 1.14 – L'étoile $K_{1,q}$ et L'étoile subdivisée $K^*_{1,q}$.

Double étoile

Double étoile est le graphe obtenu par deux étoiles $K_{1,r}$ et $K_{1,s}$ en ajoutant une arête reliant les deux centres, notée $S_{r,s}$. Un exemple de ce graphe est donné dans la Figure 1.15.

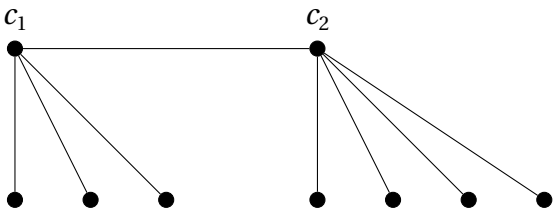


FIGURE 1.15 – Double étoile $S_{r,s}$.

Couronne d'un graphe

Soit G un graphe d'ordre n. La couronne de G notée par $G \circ K_1$ est obtenu à partir de G en joignant un sommet a chacun de ses sommets. Pour le cycle C_6, sa couronne est représentée par le graphe de la Figure 1.16 .

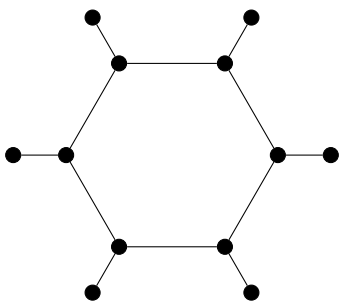


FIGURE 1.16 – La couronne d'un C_6.

Graphe joint de deux graphes

Soient G_1 et G_2 deux graphes. Le graphe $G_1 \vee G_2$ défini par l'ensemble de sommets $V(G_1) \cup V(G_2)$ et l'ensemble d'arêtes $V(G_1) \cup V(G_2) \cup E'$ où E' est l'ensemble de toutes les arêtes possibles joignant les sommets de $V(G_1)$ à $V(G_2)$, est appelé le graphe joint de G_1 et G_2. Un exemple d'un graphe joint de deux graphes est illustré dans la Figure 1.17.



FIGURE 1.17 – Le graphe joint de P_2 et P_3.

Graphe complémentaire

Soit G un graphe quelconque. Le graphe complémentaire d'un graphe G, noté $\overline{G}$, est le graphe avec le même ensemble des sommets $V(G)$, mais les arêtes si $uv \in E(G)$ si et seulement si $uv \notin E(\overline{G})$, c'est-à-dire que $E(\overline{G}) = \overline{E(G)}$ (Voir la Figure 1.18).

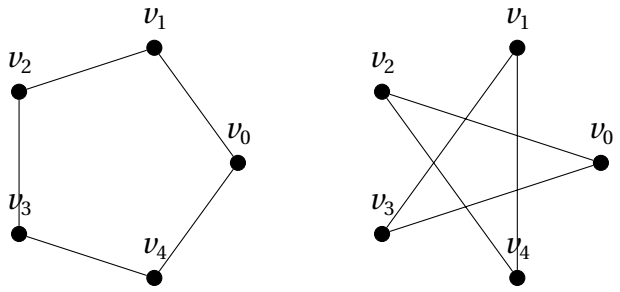


FIGURE 1.18 – Le cycle C_5 et son complémentaire $\overline{G}$.

Couplage

On appelle couplage S de G un sous-ensemble d'arêtes de E deux à deux non adjacentes. Un couplage maximum est un couplage de taille maximale et noté $\beta(G)$. L'ensemble $\{(x_1x_2),(x_3x_4)\}$ forme un couplage maximum (Voir la figure 1.19).

Stabilité

Un ensemble des sommets de G est stable (ou indépendant), si l'ensemble de ses sommets sont deux à deux non adjacents. Un stable maximum est un stable de cardinalité maximale, noté $\alpha(G)$. L'ensemble x_2, x_3 forme un stable maximum (Voir la figure 1.19).

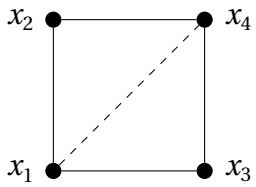


FIGURE 1.19 – Le graphe $C_4 + e$.

Clique

On appelle clique d'un graphe G un sous ensemble de sommets de V deux à deux adjacents. Une clique maximum de G est une clique de cardinalité maximale. La cardinalité d'une clique maximum est appelée nombre de clique, et notée $\omega(G)$. L'ensemble x_1, x_2, x_3 forme une clique maximale (Voir la figure 1.20).

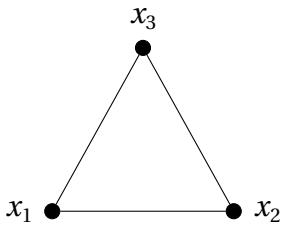


FIGURE 1.20 – Le graphe K_3.

Multigraphe

Un multigraphe est un graphe sans orientation. Une arête multiple est une arête e_1 d'extrémités x et y, telle qu'il existe une arête $e_1 \neq e_2$ avec les mêmes extrémités (Voir la Figure 1.21).[7]

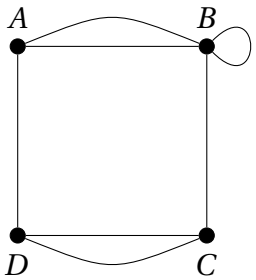


FIGURE 1.21 – Un multigraphe avec 4 sommets

Section 1.3

La coloration dans les graphes

1.3.1 Coloration des sommets

Définition 1.3 *[29]*

- *On appelle coloration des sommets d'un graphe toute attribution d'une couleur à chaque sommet. Une coloration utilisant k couleur est appelée une k-coloration. Une coloration est propre si chaque deux sommets voisins portent des couleurs différentes.*
- *Etant donné un graphe $G=(V,E)$. Une k-coloration propre des sommets de G peut être vue comme une partition de V en k stables.*
- *Une coloration triviale de G consiste à attribuer à chacun de ses sommets une couleur différente. (Tout graphe à n sommets admet une n-coloration).*

Définition 1.4

- ***Le nombre chromatique** d'un graphe G, noté $\chi(G)$, est le plus petit nombre de couleurs nécessaires à une coloration propre des sommets de G.*
- *On le définit aussi comme étant le plus petit entier k tel que G admet une k-coloration. C'est donc la taille minimale d'une partition de $V(G)$ en stables. Une $\chi(G)$-coloration d'un graphe G est dite coloration optimale (minimale) de G, il est dit aussi que G est $\chi(G)$-chromatique.*

Pour le graphe $G=K_4-e$ de la Figure 1.22, G est 3-colorable et $\chi(G)=3$.

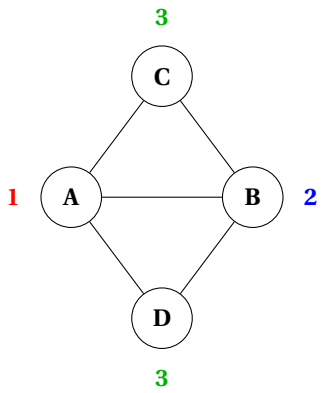


FIGURE 1.22 – Coloration propre du graphe $G=K_4-e$ avec $\chi(G)=3$

1.3.2 Bornes sur le nombre chromatique

Proposition 1.2 *Le nombre chromatique d'un graphe complet d'ordre n est n.*

Voir la Figure 1.23 :

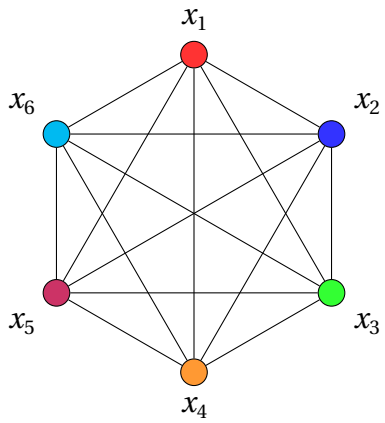


FIGURE 1.23 – Coloration propre du graphe complet K_6 avec $\chi(K_6)=6$.

Proposition 1.3 *Le nombre chromatique d'un graphe discret d'ordre n est* 1.

Voir la Figure 1.24 :

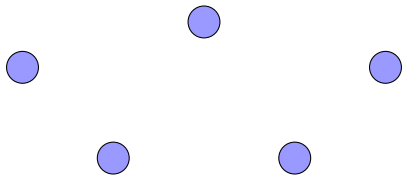


FIGURE 1.24 – Coloration propre du graphe discret $\overline{K_5}$ avec $(\chi(\overline{K_5})=1)$.

Proposition 1.4 *Le nombre chromatique d'un cycle d'ordre pair est* 2.

Voir la Figure 1.25 :

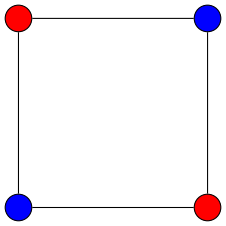


FIGURE 1.25 – Coloration propre du cycle pair C_4 avec $\chi(C_4)=2$.

Proposition 1.5 *Le nombre chromatique d'un cycle d'ordre impair* ≥ 3 *est* 3.

Voir la Figure 1.26 :

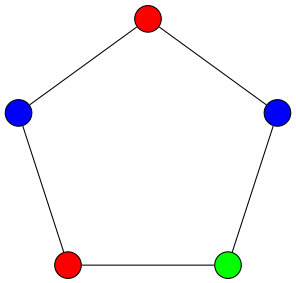


FIGURE 1.26 – Coloration propre du cycle impair C_5 avec $\chi(C_5)=3$.

Proposition 1.6 *Le nombre chromatique d'une chaîne d'ordre* ≥ 2 *est* 2.

Voir la Figure 1.27 :



FIGURE 1.27 – Coloration propre du graphe P_6 avec $\chi(P_6)=2$.

Pour davantage de détails concernant les démonstrations des propositions ci-dessus, le lecteur pourra se référer aux références [7, 10, 29, 43].

1.3.3 Coloration et complexité

La détermination du nombre chromatique d'un graphe constitue un problème notoirement difficile. En effet, décider si un graphe est k-coloriable est un problème **NPcomplet** pour tout $k \geq 3$ [23]. Par conséquent, lorsqu'on traite de graphes de grande taille ou à structure complexe, le recours à des **approches exactes** devient souvent impraticable. En pratique, on utilise donc des **heuristiques** ou des **algorithmes d'approximation**, tels que **l'algorithme glouton (greedy)**, les **méta-heuristiques** (comme les **algorithmes génétiques** ou le **recuit simulé**), **l'algorithme de Welsh-Powell**, ou encore **des méthodes fondées sur la théorie des contraintes**. Ces techniques permettent généralement de trouver des colorations proches de l'optimum dans un temps raisonnable, même si elles ne garantissent pas toujours la solution minimale.

1.3.3.1 Algorithme de coloration de Welsh et Powell (Algorithme glouton)

1. Classer les sommets suivant un ordre décroissant de leur degrés.
2. En parcourant la liste dans l'ordre, attribuer une couleur non encore utilisée au premier sommet non coloré, et attribuer cette même couleur à chaque sommet non encore coloré, et non adjacent à un sommet de cette couleur.
3. S'il reste des sommets non encore colorés dans le graphe, revenir à l'étape 2.

L'algorithme de de Welsh et Powell couramment utilisé permet d'obtenir une assez bonne coloration d'un graphe, c'est-à-dire une coloration n'utilisant pas un trop grand nombre de couleurs. Cependant, il n'assure pas que le nombre de couleurs utilisé soit minimum (et donc égal au nombre chromatique du graphe)[23].

1.3.4 Applications de la coloration

La coloration des graphes trouve de nombreuses applications concrètes dans des domaines variés. Elle est notamment utilisée pour **la génération automatique d'emplois du temps afin d'éviter les conflits entre groupes, enseignants et salles**[10]. Dans les télécommunications, elle permet d'attribuer des fréquences aux antennes relais de manière à éviter les interférences. En géographie, la coloration de cartes garantit que deux régions adjacentes soient distinguables par des couleurs différentes. Elle intervient également en ordonnancement industriel ou en informatique pour planifier des tâches qui ne doivent pas être exécutées simultanément.

Section 1.4

Propriétés et algorithmes fondamentaux

Les algorithmes de parcours et de recherche du chemin le plus court sont des concepts de base en théorie des graphes. Ces algorithmes sont utilisés pour résoudre tout un certains d'énoncés de problèmes, depuis la recherche de cycles jusqu'à l'optimisation des routes. Dans cette section, nous décrivons les algorithmes de parcours (DFS et BFS) et les algorithmes de recherche du chemin le plus court (Dijkstra et Bellman-Ford).

1.4.1 Parcours dans les graphes

Le parcours d'un graphe consiste à visiter tous ses sommets en empruntant ses arêtes. Deux méthodes principales sont utilisées : le parcours en profondeur (DFS) et le parcours en largeur (BFS). Ces algorithmes sont cruciaux pour de nombreuses applications, telles que la recherche de cycles, la détection des composantes connexes ou encore la résolution de labyrinthes.

Parcours en profondeur (DFS - Depth-First Search)

Cette exploration consiste à parcourir le graphe à partir du sommet de départ la racine (r) puis tracer une chaîne à partir de ce sommet puis choisir un autre sommet (parmi les voisins des sommets de cette chaîne dans l'ordre) et faire de même jusqu'à la fin[5].

Parcours en largeur (BFS - Breadth-First Search)

Cette exploration consiste à parcourir le graphe à partir du sommet de départ la racine (r) puis ses voisins puis les voisins des voisins non explorés et ainsi de suite jusqu'à la fin [5].

1.4.2 Algorithmes de recherche de chemin le plus court

Ces algorithmes sont utilisés pour trouver le chemin le plus court entre deux sommets dans un graphe pondéré.

Algorithme de Dijkstra

L'algorithme de Dijkstra est l'un des algorithmes les plus populaires pour résoudre le problème du plus court chemin. Il est utilisé pour trouver le chemin le plus court entre un sommet source et tous les autres sommets dans un graphe pondéré à poids non négatifs. Le principe de l'algorithme repose sur l'attribution d'une distance minimale à chaque sommet, initialement infinie sauf pour le sommet source (dont la distance est fixée à 0). Ensuite, il explore progressivement les sommets les plus proches du sommet source, mettant à jour les distances des sommets voisins si un chemin plus court est trouvé. Ce processus se répète jusqu'à ce que tous les sommets accessibles aient été visités [37, 24].

Algorithme de Dijkstra [5]

Début
$S \leftarrow \{r\}$; $k \leftarrow 1$; $f[k] \leftarrow r$;
Pour tout $x \in X$ **faire** $\pi[x] \leftarrow +\infty$, **faire** $\pi[r] \leftarrow 0$;
Tant que $(k < n)$ et $(\pi[x] < +\infty)$, **Faire**
  Pour tout $u \in U$ **tel que** $(I(u) = f[k])$ et $(T(u) \notin S)$, **Faire**
    $x \leftarrow T(u)$;
    Si $(\pi[x] > \pi[f[k]] + p[u])$,
    Alors $\pi(x) \leftarrow \pi[f[k]] + p[u]$; Pré$[x] \leftarrow f[k]$;
    Fin Si,
  Fait,
  $x \leftarrow y$ tel que $y \in X - S$ et $\pi[y]$ est minimal;
$k \leftarrow k+1$; $f[k] \leftarrow x$; $S \leftarrow S \cup \{x\}$;
Fait, **Fin**

Algorithme 1 – Algorithme de Dijkstra

Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford est plus général que celui de Dijkstra, car il permet de traiter des graphes avec des poids négatifs. Il offre également l'avantage de détecter les cycles de poids négatifs accessibles depuis le sommet source.

Son fonctionnement repose sur trois étapes principales. Tout d'abord, il initialise les distances en attribuant une valeur de 0 à la distance du sommet source, tandis que toutes les autres distances sont définies à l'infini. Ensuite, il répète $(V-1)$ fois un processus d'amélioration des distances : pour chaque arête (u,v) de poids w, il met à jour la distance de v si un chemin plus court passant par u est trouvé. Enfin, il vérifie la présence de cycles de poids négatifs en testant si une mise à jour est encore possible après ces $(V-1)$ itérations. Si une distance peut encore être réduite, alors un cycle négatif est présent dans le graphe [5].

Algorithme de Bellman-Ford [5]

Début
  $S \leftarrow \{r\}$;
  Pour tout $x \in X$, **faire** :
    $\pi[x] \leftarrow +\infty$;
    Pré$[x] \leftarrow$ NULL;
  Fait
  $\pi[r] \leftarrow 0$;
  Pour tout $(x \in X - S)$ **tel que** $(\forall u \in U,\ T(u) = x$ **on a** $I(u) \in S)$, **faire** :
    Pour tout $(y,x) \in U$, **faire** :
      Si $\pi[x] > \pi[y] + P[(y,x)]$ **alors**
        $\pi[x] \leftarrow \pi[y] + P[(y,x)]$;
        Pré$[x] \leftarrow y$;
      Fin Si
    Fait
  $S \leftarrow S \cup \{x\}$;
Fin

Algorithme 2 – Algorithme de Bellman-Ford

Section 1.5

Représentation des graphes en mémoire

Il existe deux façons classiques de représenter un graphe en machine [39] : par une matrice d'adjacence ou par un ensemble de listes d'adjacence.

1. Représentation par matrice d'adjacence

Soit le graphe $G = (S, A)$. On suppose que les sommets de S sont numérotés de 1 à n, avec $n = |S|$. La représentation par matrice d'adjacence de G consiste en une matrice booléenne M de taille $n \times n$ telle que :

$$M[i][j] = \begin{cases} 1 & \text{si } (i,j) \in A \\ 0 & \text{sinon} \end{cases}$$

Si le graphe est valué (par exemple, si des distances sont associées aux arcs), on peut utiliser une matrice d'entiers, de telle sorte que $M[i][j]$ soit égal à la valuation de l'arc (i, j) si $(i, j) \in A$. S'il n'existe pas d'arc entre deux sommets i et j, on peut placer une valeur particulière (par exemple 0, $-\infty$ ou `null`) dans $M[i][j]$.

Dans le cas de graphes non orientés, la matrice est symétrique par rapport à sa diagonale descendante. On peut alors ne mémoriser que la composante triangulaire supérieure de la matrice d'adjacence.

Taille mémoire nécessaire : La matrice d'adjacence d'un graphe ayant n sommets nécessite de l'ordre de $O(n^2)$ emplacements mémoire. Si le nombre d'arcs est très inférieur à n^2, cette représentation est donc loin d'être optimale.

Opérations :

- Le test de l'existence d'un arc est immédiat.
- Le calcul du degré d'un sommet ou l'accès à tous ses successeurs nécessite le parcours de toute une ligne.
- Le parcours de tous les arcs/arêtes prend un temps de l'ordre de n^2.

2. Représentation par listes d'adjacence

Soit le graphe $G = (S, A)$. On suppose que les sommets de S sont numérotés de 1 à n, avec $n = |S|$. La représentation par listes d'adjacence de G consiste en un tableau T de n listes, une pour chaque sommet de S.

Pour chaque sommet $s_i \in S$, la liste d'adjacence $T[s_i]$ est une liste chaînée de tous les sommets s_j tels qu'il existe un arc ou une arête $(s_i, s_j) \in A$. Autrement dit, $T[s_i]$ contient la liste de tous les successeurs de s_i. L'ordre des sommets dans chaque liste est généralement arbitraire.

Si le graphe est valué, on peut stocker dans chaque élément de liste, en plus du sommet, la valuation de l'arête.

Dans le cas de graphes non orientés, pour chaque arête (s_i, s_j), on aura $s_j \in T[s_i]$ et $s_i \in T[s_j]$.

Taille mémoire nécessaire :

- Graphe orienté : la somme des longueurs des listes est égale au nombre d'arcs $|A|$.
- Graphe non orienté : la somme des longueurs est $2 \times |A|$.

Donc, la représentation nécessite $O(n + m)$ emplacements mémoire, où m est le nombre d'arcs/arêtes.

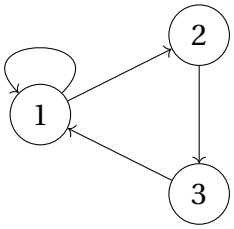
Opérations :

- Le test d'existence d'un arc (s_i, s_j) nécessite de parcourir la liste $T[s_i]$.
- Le calcul du degré d'un sommet ou l'accès à ses successeurs est efficace : il suffit de parcourir la liste correspondante.
- Le parcours de tous les arcs/arêtes se fait en $O(m)$.
- En revanche, le calcul des prédécesseurs est difficile et nécessite de parcourir toutes les listes. Une solution est de maintenir en plus une liste des prédécesseurs.

[39]

Exemple de graphe simple

Considérons le graphe orienté suivant :



Ce graphe possède 3 sommets et 4 arcs :

$$A = \{(1,1),(1,2),(2,3),(3,1)\}$$

Matrice d'adjacence

La matrice d'adjacence M associée est :

$$M = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

- $M[1][1] = 1$: boucle sur le sommet 1.
- $M[1][2] = 1$: arc de 1 vers 2.
- $M[2][3] = 1$, $M[3][1] = 1$

Listes d'adjacence

On peut représenter ce graphe par les listes suivantes :

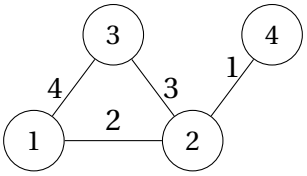
- $T[1] = \{1,2\}$
- $T[2] = \{3\}$

- $T[3] = \{1\}$

Chaque liste contient les successeurs du sommet correspondant.

Exemple de graphe non orienté valué

Considérons le graphe non orienté suivant, avec des poids représentant par exemple des distances :



Matrice d'adjacence valuée

La matrice d'adjacence valuée M est symétrique :

$$M = \begin{bmatrix} 0 & 2 & 4 & 0 \\ 2 & 0 & 3 & 1 \\ 4 & 3 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Chaque case $M[i][j]$ contient la distance entre les sommets i et j, ou 0 s'il n'y a pas d'arête.

Listes d'adjacence valuées

Les listes d'adjacence correspondantes sont :

- $T[1] = \{(2,2),(3,4)\}$
- $T[2] = \{(1,2),(3,3),(4,1)\}$
- $T[3] = \{(1,4),(2,3)\}$
- $T[4] = \{(2,1)\}$

Chaque paire (j, v) indique une arête entre le sommet courant et le sommet j, avec une valuation v.

Remarques

- La matrice est symétrique car le graphe est non orienté.

- Les listes sont redondantes : chaque arête apparaît deux fois (une fois dans chaque sens).
- La représentation par liste reste plus efficace pour les grands graphes peu denses.

Section 1.6

Représentation des graphes par l'utilisation de python

En Python, les graphes peuvent être représentés de différentes manières, notamment par des listes d'adjacence ou des matrices. Une approche consiste à utiliser un dictionnaire où chaque clé représente un sommet et est associée à ses sommets voisins.[42] La bibliothèque **NetworkX** propose une structure de données hautement performante pour modéliser des graphes complexes de manière efficace.[1] Elle facilite l'ajout de nœuds et d'arêtes, ainsi que l'exécution d'algorithmes complexes, tels que la recherche du plus court chemin ou la détection rapide de cycles. Python constitue un environnement adapté aux applications d'intelligence artificielle dans diverses circonstances et à l'analyse de réseaux. Voici un exemple simple de création et d'affichage de plus court chemin d'un graphe :

Code Python - Plus court chemin avec NetworkX

```
import networkx as nx
G = nx.Graph()
G.add_edge('A', 'B', weight=4)
G.add_edge('B', 'C', weight=2)
print(nx.shortest_path(G, 'A', 'C', weight='weight'))
```

Section 1.7

Complexité algorithmique dans les graphes

La complexité est une mesure de temps (**complexité en temps**) ou d'espace mémoire (**complexité en espace**) utilisée par un algorithme en fonction de la taille des données.

La complexité en temps d'un algorithme est le nombre d'étapes de calcul qu'il nécessite, et **sa complexité en espace** est la taille de mémoire nécessaire pour stocker les données intermédiaires au cours de son exécution [35].

Ces grandeurs sont indiquées en fonction de la taille des données du problème à résoudre, et généralement à une constante de proportionnalité près. Seul l'ordre de grandeur du nombre d'opérations, noté $\mathcal{O}$, est utilisé pour exprimer la complexité [35] :

– **Complexité logarithmique** ($\mathcal{O}(\log n)$) :

Exemple : recherche dichotomique dans un tableau trié.

- **Complexité linéaire** ($\mathcal{O}(n)$) :
  Exemple : recherche du minimum d'une liste de n éléments, calcul du produit scalaire de deux vecteurs de n éléments.
- **Complexité polynomiale** ($\mathcal{O}(n^k)$ avec $k > 1$) :
  Exemple : multiplication de deux matrices carrées d'ordre n.
- **Complexité exponentielle** ($\mathcal{O}(k^n)$ avec $k > 1$) :
  Exemple : problème du voyageur de commerce.

La complexité (maximale) du meilleur algorithme connu pour le résoudre permet aussi de **classifier les problèmes** [35] :

- **Classe P** : Problèmes qu'on peut résoudre en temps polynomial.
- **Classe E** : Problèmes qu'on peut résoudre en temps exponentiel.
- **Classe NP** : Problèmes qu'on peut vérifier en temps polynomial mais résoudre en temps exponentiel (problèmes non déterministes polynomiaux).

Voici quelques complexités courantes :

- **Parcours DFS/BFS** : $\mathcal{O}(V+E)$, où V est le nombre de sommets et E le nombre d'arêtes.
- **Dijkstra** : $\mathcal{O}(E + V \log V)$ en utilisant une file de priorité, où E est le nombre d'arêtes et V le nombre de sommets .
- **Bellman-Ford** : $\mathcal{O}(V.E)$.

La complexité spatiale dépend de la représentation [38] :

- **Matrice d'adjacence** : $\mathcal{O}(V^2)$.
- **Liste d'adjacence** : $\mathcal{O}(V+E)$.

Conclusion

À travers ce chapitre, nous avons présenté les concepts fondamentaux de la théorie des graphes, en abordant leurs types, leurs propriétés ainsi que quelques algorithmes essentiels liés aux parcours et à la recherche de chemins optimaux. Nous avons également introduit les principales méthodes de représentation des graphes, tant sur le plan théorique que pratique, notamment à l'aide du langage Python.

Dans le chapitre qui suit, nous allons nous intéresser aux techniques d'intelligence artificielle appliquées à l'optimisation mathématique, en présentant à la fois les approches classiques et les algorithmes inspirés de l'IA.

CHAPITRE 2

Travaux connexes

Dans ce chapitre, nous allons explorer l'intersection entre l'optimisation mathématique et l'intelligence artificielle. Nous présenterons d'abord les méthodes classiques d'optimisation, puis introduirons les approches basées sur l'IA, notamment les algorithmes heuristiques et métaheuristiques. Ensuite, Nous examinerons les applications de ces techniques dans la résolution de problèmes de graphes et l'optimisation du chemin le plus court.

Section 2.1

Introduction aux techniques d'intelligence artificielle en optimisation mathématique

L'optimisation mathématique est un domaine central en sciences et en ingénierie qui vise à trouver les meilleures solutions à des problèmes complexes en minimisant ou maximisant une fonction objectif [1] sous certaines contraintes. Traditionnellement, cette discipline repose sur des méthodes analytiques, basées sur des formules mathématiques exactes et des calculs rigoureux pour obtenir une solution précise, ainsi que sur des méthodes algorithmiques bien établis. Cependant, en combinant la puissance de l'intelligence artificielle avec ces approches, de nouvelles perspectives s'ouvrent [34] pour obtenir des solutions plus robustes et adaptatives. Les techniques issues de l'IA, inspirées du raisonnement humain et des mécanismes naturels, permettent d'explorer de vastes espaces de recherche, d'adapter dynamiquement les solutions et d'optimiser des systèmes complexes dans des délais raisonnables, tout en relevant les défis posés par les systèmes modernes.

1. Fonction objectif (ou fonction du coût) : est une fonction mathématique qui exprime le critère de performance ou de coût que l'on souhaite optimiser. Elle quantifie ce que l'on cherche à améliorer (maximiser ou minimiser) et permet d'évaluer et de comparer les solutions possibles.

Section 2.2

Algorithmes classiques d'optimisation

Les algorithmes classiques d'optimisation reposent sur des méthodes déterministes et analytiques, souvent efficaces pour des problèmes convexes et différentiables. Parmi les plus répandus, on peut citer :

1. **Méthode de la descente de gradient**

   La **descente de gradient** [33] est une technique itérative visant à trouver le minimum d'une fonction différentiable. L'idée principale est de se déplacer dans la direction opposée au gradient de la fonction au point actuel, car cette direction correspond à la pente la plus abrupte vers le bas.

   Procédure :

   (a). **Initialisation** : Choisir un point de départ x_0 et un taux d'apprentissage α.

   (b). **Itération** : Mettre à jour le point actuel selon la règle $x_{k+1} = x_k - \alpha \nabla f(x_k)$, où $\nabla f(x_k)$ est le gradient de la fonction f au point x_k.

   (c). **Convergence** : Répéter l'étape précédente jusqu'à ce que $\|\nabla f(x_k)\|$ soit inférieur à un seuil prédéfini, indiquant la proximité d'un minimum.

   Cette méthode est largement utilisée en apprentissage automatique pour l'entraînement de modèles, notamment dans la mise à jour des poids des réseaux de neurones. Cependant, elle peut être sensible au choix du taux d'apprentissage et peut converger lentement près des minima plats.

2. **Méthode de Newton**

   La **méthode de Newton** [33] est une approche d'optimisation qui utilise non seulement le gradient (dérivées premières) mais aussi la matrice hessienne (dérivées secondes) de la fonction objectif pour trouver ses extrema.

   Procédure :

   (a). **Initialisation** : Choisir un point de départ x_0.

   (b). **Itération** : Calculer la direction de descente en résolvant $H_f(x_k)d_k = -\nabla f(x_k)$, où $H_f(x_k)$ est la hessienne de f en x_k. Mettre à jour le point actuel selon $x_{k+1} = x_k + d_k$.

   (c). **Convergence** : Répéter jusqu'à ce que $\|\nabla f(x_k)\|$ soit suffisamment petit.

   La méthode de Newton offre une convergence quadratique près du minimum, ce qui signifie qu'elle est très rapide si l'on est proche de la solution. Toutefois, elle nécessite le calcul de la hessienne, ce qui peut être coûteux en termes de calcul et de mémoire, surtout pour des problèmes de grande dimension.

3. **Algorithme du simplexe**

Développé par George Dantzig [15], l'**algorithme du simplexe** est une méthode pour résoudre les problèmes de programmation linéaire. Il exploite la structure géométrique des problèmes linéaires en naviguant d'un sommet à l'autre du polytope des solutions admissibles pour trouver l'optimum.

Procédure :

(a). **Initialisation** : Identifier une solution de base admissible initiale.

(b). **Itération** : Déterminer si une amélioration est possible en examinant les coûts réduits. Si oui, choisir une variable entrante et une variable sortante pour pivoter vers une nouvelle solution de base.

(c). **Convergence** : Répéter jusqu'à ce qu'aucune amélioration ne soit possible, indiquant que l'optimum est atteint.

L'algorithme du simplexe est réputé pour sa performance en pratique, bien que, dans le pire des cas, il puisse nécessiter un nombre exponentiel d'itérations.

En résumé, ces algorithmes classiques constituent la base de nombreuses techniques d'optimisation utilisées aujourd'hui. Leur compréhension est essentielle pour aborder des problèmes d'optimisation plus avancés et pour apprécier les développements modernes en la matière.

Section 2.3

Intelligence artificielle et optimisation : approches et algorithmes

Les heuristiques et les métaheuristiques sont des méthodes de résolution de problèmes utilisées en optimisation, en intelligence artificielle et en recherche opérationnelle. Elles permettent de trouver des solutions approchées (souvent optimales ou quasi-optimales [2]) à des problèmes complexes où les méthodes exactes sont trop coûteuses en temps ou en ressources.

2.3.1 Les heuristiques

Définition 2.1 *[2] Une heuristique est une méthode de recherche qui exploite des informations spécifiques au domaine pour orienter le processus de résolution d'un problème, en évitant une exploration exhaustive de l'espace des solutions. Contrairement aux méthodes de recherche aveugle, qui examinent systématiquement tous les chemins possibles, les heuristiques sélectionnent les nœuds à explorer en fonction de critères pertinents, réduisant ainsi*

2. Quasi-optimale est une solution qui s'approche de l'optimum mais qui n'est pas nécessairement la meilleure possible.

l'effort de recherche. Cependant, bien qu'elles puissent accélérer la découverte d'une solution, elles ne garantissent pas toujours d'atteindre l'optimum. L'objectif est de minimiser à la fois le coût du chemin trouvé et celui de la recherche nécessaire pour y parvenir.

2.3.2 Algorithmes Heuristiques

2.3.2.1 Algorithme A*

L'**algorithme A*** introduit par Peter Hart, Nils Nilsson et Bertram Raphael en 1968 [27], est une méthode de recherche de chemin optimale dans un graphe pondéré. Il combine les avantages de l'algorithme de Dijkstra et des algorithmes de recherche heuristique pour trouver le chemin le plus court de manière efficace.

Fonctionnement :

1. **Initialisation** : On démarre avec un nœud initial et une liste ouverte (open list) contenant ce nœud. Une liste fermée (closed list) est également utilisée pour garder trace des nœuds déjà explorés.
2. **Évaluation des nœuds** : Pour chaque nœud n, on calcule :
   - $g(n)$: le coût réel du chemin depuis le nœud de départ jusqu'à n.
   - $h(n)$: une estimation heuristique du coût restant pour atteindre le nœud objectif depuis n.
   - $f(n) = g(n) + h(n)$: une estimation du coût total du chemin passant par n.
3. **Sélection du nœud à explorer** : Le nœud avec la plus petite valeur de $f(n)$ est sélectionné pour exploration.
4. **Expansion des nœuds** : Les nœuds voisins du nœud sélectionné sont évalués et ajoutés à la liste ouverte s'ils ne sont pas déjà dans la liste fermée ou s'ils offrent un chemin plus court que précédemment trouvé.
5. **Itération** : Le processus se répète jusqu'à ce que le nœud objectif soit atteint ou que la liste ouverte soit vide, indiquant qu'aucun chemin n'existe.

L'efficacité de l'algorithme A* dépend fortement du choix de l'heuristique $h(n)$. Si $h(n)$ est admissible (c'est-à-dire qu'elle ne surestime jamais le coût réel), A* garantit de trouver le chemin optimal. Cet algorithme est largement utilisé dans les domaines de la robotique, des jeux vidéo et des systèmes de navigation.

2.3.2.2 Méthodes gloutonnes (greedy algorithms)

Les **méthodes gloutonnes** [4] sont des algorithmes qui construisent une solution en effectuant une série de choix locaux optimaux, dans l'espoir que ces choix conduisent à une solution globale optimale. À chaque étape, l'algorithme sélectionne l'option qui semble la meilleure selon un critère spécifique, sans revenir sur les décisions précédentes. Cependant,

cette approche ne garantit pas toujours l'obtention de la solution optimale globale, car un choix localement optimal peut empêcher d'atteindre une meilleure solution globale.

Principe de fonctionnement :

1. **Initialisation** : Commencer avec une solution vide ou partielle.
2. **Sélection** : À chaque étape, choisir l'élément qui optimise le mieux le critère local.
3. **Validation** : Vérifier si l'ajout de cet élément respecte les contraintes du problème.
4. **Ajout** : Intégrer l'élément sélectionné à la solution en cours.
5. **Répétition** : Répéter les étapes 2 à 4 jusqu'à ce qu'aucun autre élément ne puisse être ajouté.

Cette méthode est efficace pour le problème du sac à dos fractionnaire, mais peut ne pas donner une solution optimale pour le problème du sac à dos 0-1, où chaque objet doit être pris en entier ou pas du tout.

2.3.2.3 Heuristiques de construction

Les **heuristiques de construction** [40] élaborent une solution en ajoutant progressivement des éléments, en suivant une certaine règle ou intuition. Ces méthodes sont souvent utilisées pour générer rapidement une solution initiale qui peut ensuite être améliorée par d'autres techniques.

Principe de fonctionnement :

1. **Initialisation** : Commencer avec une solution vide ou partielle.
2. **Sélection** : Choisir le prochain élément à ajouter en fonction d'une règle prédéfinie.
3. **Ajout** : Intégrer l'élément sélectionné à la solution en cours.
4. **Répétition** : Répéter les étapes 2 et 3 jusqu'à ce que la solution soit complète.

Cette approche ne garantit pas la solution optimale, mais fournit une solution rapidement exploitable.

2.3.3 Les métaheuristiques

Définition 2.2 *[19] Une métaheuristique est une méthode générique conçue pour résoudre des problèmes d'optimisation combinatoire, caractérisés par un nombre exponentiel de combinaisons possibles. À mesure que la taille du problème augmente, le nombre de combinaisons à explorer croît de manière exponentielle, rendant la résolution complexe et chronophage. Par exemple, dans la conception d'un emploi du temps, l'ajout de quelques cours peut considérablement augmenter les combinaisons possibles, compliquant ainsi la recherche de la meilleure solution. Les métaheuristiques visent à identifier l'optimum global en explorant efficacement l'espace de recherche, en se concentrant sur les zones les plus prometteuses. Contrairement aux heuristiques, qui sont des méthodes algorithmiques fournissant rapidement des solutions à divers problèmes décisionnels, les métaheuristiques englobent plusieurs algorithmes.*

2.3.4 Algorithmes Métaheuristiques

2.3.4.1 Algorithmes génétiques

Les **algorithmes génétiques (AG)** [28] sont des méthodes d'optimisation inspirées par le processus de sélection naturelle. Introduits par John Holland dans les années 1970, ils modélisent des mécanismes biologiques tels que la reproduction, la mutation et la sélection pour résoudre des problèmes complexes.

Principe de fonctionnement :

1. **Codage des solutions** : Chaque solution potentielle du problème est représentée sous forme d'un chromosome, souvent une chaîne binaire, bien que d'autres codages soient possibles selon la nature du problème.
2. **Population initiale** : Une population de chromosomes est générée aléatoirement ou à l'aide d'une heuristique spécifique.
3. **Évaluation (Fitness)** : Chaque individu est évalué à l'aide d'une fonction d'adaptation (fitness function) qui mesure sa qualité ou son adéquation par rapport au problème à résoudre.
4. **Sélection** : Les individus sont sélectionnés pour la reproduction en fonction de leur valeur d'adaptation. Les méthodes de sélection courantes incluent la roulette (roulette wheel selection), le tournoi (tournament selection) et la sélection par rang (rank selection).
5. **Croisement (Crossover)** : Des paires d'individus sélectionnés échangent des segments de leurs chromosomes pour créer une nouvelle descendance. Les techniques de croisement incluent le croisement à un point, à deux points et le croisement uniforme.
6. **Mutation** : Avec une faible probabilité, certains gènes des descendants sont modifiés aléatoirement pour introduire de la diversité génétique et éviter la convergence prématurée vers des optima locaux.
7. **Nouvelle génération** : Les nouveaux individus remplacent tout ou partie de la population actuelle, et le processus se répète jusqu'à ce qu'un critère d'arrêt soit atteint (nombre maximal de générations, convergence de la population, etc...).

Les algorithmes génétiques sont particulièrement adaptés aux problèmes où l'espace de recherche est vaste et mal défini, comme l'optimisation de fonctions complexes, la planification, ou encore la conception de réseaux de neurones.

2.3.4.2 Recuit simulé

Inspiré du processus de refroidissement lent des métaux en métallurgie, le **recuit simulé** [30] est une technique d'optimisation qui permet d'échapper aux minima locaux en acceptant temporairement des solutions moins bonnes.

Principe :

1. **Initialisation** : Définir une température initiale élevée et une solution de départ.
2. **Itération** : À chaque étape, générer une nouvelle solution voisine de la solution actuelle.
3. **Évaluation** : Calculer la variation de coût ΔE entre la nouvelle solution et la solution actuelle.
4. **Acceptation** : Si ΔE est négatif (meilleure solution), accepter la nouvelle solution. Sinon, accepter la nouvelle solution avec une probabilité $\exp(-\Delta E/T)$, où T est la température actuelle.
5. **Mise à jour de la température** : Réduire progressivement la température selon un schéma de refroidissement prédéfini.
6. **Arrêt** : Répéter les étapes précédentes jusqu'à ce que la température atteigne un seuil minimal ou qu'un critère d'arrêt soit satisfait.

Ce processus permet d'explorer efficacement l'espace des solutions, en équilibrant l'exploitation des solutions prometteuses et l'exploration de nouvelles régions pour éviter la convergence prématurée vers des optima locaux.

2.3.4.3 Algorithmes à colonies de fourmis (Ant Colony Optimization - ACO)

Inspirés du comportement collectif des colonies de fourmis, les **algorithmes à colonies de fourmis** [18] sont des métaheuristiques utilisées pour résoudre des problèmes d'optimisation combinatoire. Les fourmis réelles déposent des phéromones sur les chemins qu'elles empruntent vers une source de nourriture, influençant ainsi le choix des autres membres de la colonie et favorisant l'émergence de chemins optimaux.

Principe de fonctionnement :

1. **Initialisation** : Placer un certain nombre de fourmis virtuelles sur les nœuds du graphe représentant le problème à résoudre.
2. **Construction de solutions** : Chaque fourmi construit une solution en se déplaçant de nœud en nœud, en fonction de deux facteurs :
   - **Intensité des phéromones** : Les chemins avec une concentration plus élevée de phéromones sont plus attractifs.
   - **Attractivité heuristique** : Une mesure locale de l'intérêt de choisir un certain chemin, par exemple, la distance inverse dans le problème du voyageur de commerce.
3. **Mise à jour des phéromones** : Après que toutes les fourmis ont construit leurs solutions, les niveaux de phéromones sont mis à jour :
   - **Évaporation** : Une partie des phéromones s'évapore pour éviter la convergence prématurée.
   - **Dépôt** : Les fourmis déposent des phéromones sur les chemins proportionnellement à la qualité de leur solution.

4. **Itération** : Le processus est répété jusqu'à ce qu'un critère d'arrêt soit satisfait (nombre maximal d'itérations, convergence des solutions, etc.).

Les algorithmes ACO ont été appliqués avec succès à divers problèmes, notamment le problème du voyageur de commerce, la planification et l'ordonnancement, et l'optimisation de réseaux.

2.3.4.4 Recherche tabou (Tabu Search)

La **recherche tabou** [22] est une métaheuristique développée par Fred Glover dans les années 1980, visant à améliorer les méthodes de recherche locale en évitant les cycles et en échappant aux optima locaux. Elle utilise une mémoire adaptative pour enregistrer les mouvements ou solutions récemment explorés, les rendant "tabous" pour un certain nombre d'itérations.

Principe de fonctionnement :

1. **Initialisation** : Commencer par une solution initiale et définir une liste tabou vide.
2. **Recherche locale** : Explorer le voisinage de la solution actuelle pour identifier les meilleures solutions candidates.
3. **Application des critères tabous** : Éviter de revenir aux solutions ou mouvements présents dans la liste tabou, sauf si un critère d'aspiration est satisfait (par exemple, si une solution taboue est meilleure que toutes les solutions précédemment trouvées).
4. **Mise à jour de la liste tabou** : Ajouter le mouvement ou la solution actuelle à la liste tabou et, si nécessaire, retirer les éléments les plus anciens pour maintenir une taille fixe.
5. **Itération** : Répéter le processus jusqu'à ce qu'un critère d'arrêt soit atteint (nombre maximal d'itérations, amélioration insuffisante, etc.).

La recherche tabou est particulièrement efficace pour les problèmes d'optimisation combinatoire, tels que le problème du voyageur de commerce, la planification d'horaires et l'affectation de ressources.

Ces algorithmes métaheuristiques offrent des approches puissantes pour résoudre des problèmes complexes où les méthodes classiques d'optimisation sont inefficaces ou inapplicables.

Section 2.4

Applications de l'IA dans la résolution de problèmes de graphes

Les problèmes de graphes occupent une place centrale dans de nombreux domaines liés à l'optimisation, à la planification et à l'analyse de systèmes complexes. En représentant un système à l'aide d'un graphe composé de nœuds et d'arêtes, il devient possible de modéliser efficacement des situations concrètes telles que les réseaux de transport, les circuits électroniques, les réseaux sociaux ou encore la gestion des emplois du temps. Cette modélisation permet de formaliser les entités du système et leurs relations, ce qui conditionne la pertinence des solutions obtenues.

Traditionnellement, ces problèmes étaient résolus à l'aide d'algorithmes exacts comme ceux de Dijkstra, Bellman-Ford pour les plus courts chemins, ou encore à travers des méthodes de programmation linéaire et de coloration de graphe. Bien qu'efficaces, ces méthodes montrent leurs limites dès que la complexité du graphe augmente fortement.

L'intelligence artificielle vient alors enrichir le champ des solutions possibles grâce à des approches plus flexibles et puissantes. Des heuristiques comme l'algorithme A introduisent une estimation du coût restant, accélérant la recherche tout en conservant l'optimalité. Les métaheuristiques, telles que les algorithmes génétiques, permettent d'explorer efficacement de vastes espaces de solutions en s'inspirant de l'évolution naturelle. L'apprentissage par renforcement, quant à lui, permet à un agent d'apprendre à maximiser une récompense au fil de ses interactions avec le graphe, ce qui le rend adapté aux environnements dynamiques.

Enfin, les réseaux de neurones de type Graph Neural Networks (GNNs) offrent une manière innovante d'exploiter directement la structure du graphe [25] en mettant à jour les représentations de chaque nœud en fonction de son voisinage. Ces avancées permettent à l'IA non seulement de résoudre des problèmes complexes, mais aussi de les reformuler et de les explorer de manière plus intelligente et efficace que les approches traditionnelles.

Section 2.5

État de l'art sur l' optimisation du chemin le plus court

L'optimisation du chemin le plus court consiste à trouver, dans un graphe pondéré, la trajectoire la moins coûteuse reliant deux sommets. Ce problème fondamental, présent dans des domaines tels que les transports, les télécommunications ou la planification, a d'abord été traité par des algorithmes classiques comme Dijkstra ou Bellman-Ford. Bien que ces méthodes garantissent des solutions optimales, leur efficacité diminue rapidement face à la croissance de la taille du graphe ou à l'ajout de contraintes, ce qui limite leur applicabilité

dans des contextes dynamiques ou à grande échelle.

Pour répondre à ces défis, plusieurs approches issues de l'intelligence artificielle ont été développées. L'algorithme A* est une heuristique informée : à chaque étape, il évalue non seulement le coût déjà parcouru mais aussi une estimation du coût restant (grâce à une fonction heuristique, souvent basée sur la distance euclidienne ou une approximation du coût minimal). Cette anticipation permet à A* de diriger intelligemment la recherche vers les nœuds les plus prometteurs, réduisant considérablement le nombre d'explorations inutiles.

Les algorithmes métaheuristiques, comme les algorithmes génétiques, cherchent le plus court chemin en codant chaque chemin potentiel comme un individu dans une population. À chaque génération, les meilleurs chemins sont sélectionnés selon leur longueur ou leur validité, puis combinés et mutés pour générer de nouvelles pistes[31]. Ce processus d'évolution progressive permet de découvrir des chemins courts sans avoir à explorer toutes les possibilités de manière exhaustive.

Les colonies de fourmis, autre méthode métaheuristique, reposent sur un principe inspiré de la nature ; plusieurs agents (fourmis virtuelles) parcourent le graphe en déposant une trace (appelée phéromone) sur les chemins empruntés. Plus un chemin est court et emprunté fréquemment, plus il est renforcé, ce qui incite les autres fourmis à le privilégier. Ce mécanisme collectif converge vers des chemins efficaces, bien que non nécessairement optimaux.

L'apprentissage par renforcement, quant à lui, permet à un agent d'explorer le graphe et de tester différents itinéraires. Chaque déplacement entraîne une récompense (positive ou négative), et l'agent apprend progressivement, par essais et erreurs, quelles actions le conduisent au chemin le plus avantageux. Ce type d'approche est particulièrement pertinent dans des environnements incertains ou en évolution.

Enfin, les Graph Neural Networks (GNNs) apportent une innovation importante ; ils permettent d'entraîner un modèle neuronal à comprendre la structure du graphe et à prédire, à partir des caractéristiques des nœuds et de leurs connexions, les relations de proximité ou de coût entre les sommets. Plutôt que d'explorer explicitement les chemins, les GNNs apprennent à inférer rapidement le chemin le plus pertinent selon les motifs observés dans les d'entraînement.

Section 2.6

Différents types d'optimisation

L'optimisation est un domaine fondamental des mathématiques appliquées et de l'informatique. Elle consiste à rechercher la meilleure solution possible à un problème donné, en respectant un ensemble de contraintes. Les types d'optimisation varient en fonction de la nature des fonctions utilisées et des contraintes imposées. Cette section présente les principaux types d'optimisation utilisés dans la recherche et l'industrie.

2.6.1 Programmation linéaire (PL)

La **programmation linéaire (PL)** est une méthode d'optimisation mathématique visant à *maximiser ou minimiser une fonction objectif linéaire*, tout en respectant un ensemble de *contraintes linéaires*. Ce cadre est largement utilisé dans des domaines tels que la gestion de production, la planification des transports, la logistique ou l'allocation optimale des ressources. Le *modèle mathématique standard* d'un problème de programmation linéaire se présente comme suit :

$$\text{Maximiser (ou minimiser) sous les contraintes} \begin{cases} z = c^T x \\ Ax \leq b, \quad x \geq 0 \end{cases}$$

où :

- $x \in \mathbb{R}^n$ est le vecteur des variables de décision,
- $c \in \mathbb{R}^n$ est le vecteur des coefficients de la fonction objectif,
- $A \in \mathbb{R}^{m \times n}$ est la matrice des contraintes,
- $b \in \mathbb{R}^m$ est le vecteur des ressources disponibles.

L'objectif est donc de déterminer la valeur de x qui optimise la fonction $z = c^T x$, tout en respectant les contraintes d'inégalités. Parmi les *méthodes de résolution les plus utilisées*, on retrouve l'**algorithme du simplexe**, très efficace pour de nombreux cas pratiques, ainsi que les **méthodes des points intérieurs**, mieux adaptées aux problèmes de grande taille. Dans la pratique, ces méthodes sont souvent implémentées à l'aide de **solveurs performants** comme `CPLEX`, `Gurobi` ou `GLPK`, capables de traiter des instances de grande dimension avec efficacité et précision [3].

2.6.2 Programmation quadratique (QP)

La **programmation quadratique (QP)** est une généralisation de la programmation linéaire dans laquelle la *fonction objectif est quadratique*, c'est-à-dire qu'elle inclut des termes au carré, tandis que les *contraintes demeurent linéaires*. Cette classe de problèmes

est très répandue dans des domaines tels que la **finance** (ex. : optimisation de portefeuilles) et l'**apprentissage automatique**, notamment dans la formulation des *machines à vecteurs de support* (SVM) [11]. Le modèle mathématique standard d'un problème de programmation quadratique se formule de la manière suivante :

$$\text{Minimiser sous les contraintes} \begin{cases} f(x) = \frac{1}{2} x^T Q x + c^T x \\ Ax \leq b \end{cases}$$

où :

- $x \in \mathbb{R}^n$ est le vecteur des variables de décision,
- $Q \in \mathbb{R}^{n \times n}$ est une matrice symétrique définie positive (ou semi-définie positive),
- $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ sont les vecteurs et matrices de paramètres du modèle.

Le terme quadratique $\frac{1}{2} x^T Q x$ permet de modéliser des effets de courbure ou des interactions entre variables. Ce type d'optimisation est plus complexe que la programmation linéaire mais reste convexe si Q est définie positive, ce qui garantit un unique minimum global. Les **méthodes de résolution** incluent des algorithmes spécialisés comme le *gradient conjugué*, les *méthodes de points intérieurs pour QP*, ou encore des *solveurs dédiés* tels que `OSQP`, `quadprog` (MATLAB), ou `Gurobi`.

2.6.3 Optimisation non linéaire

L'**optimisation non linéaire** s'applique lorsque la fonction objectif, les contraintes, ou les deux, sont *non linéaires*. Ce type de problème est souvent plus difficile à résoudre que les modèles linéaires ou quadratiques, en raison de la présence possible de plusieurs *minima locaux* et d'une surface d'optimisation non convexe. Le modèle mathématique général d'un problème d'optimisation non linéaire se formule ainsi :

$$\text{Minimisersous les contraintes} \begin{cases} f(x) \\ g_i(x) \leq 0, \quad i = 1, \ldots, m \\ h_j(x) = 0, \quad j = 1, \ldots, p \end{cases}$$

où :

- $x \in \mathbb{R}^n$ est le vecteur des variables de décision,
- $f(x)$ est la fonction objectif à minimiser, généralement non linéaire,
- $g_i(x)$ sont les contraintes inégalitaires non linéaires,
- $h_j(x)$ sont les contraintes égalitaires non linéaires.

Ce type d'optimisation est courant dans des domaines comme l'**ingénierie**, la **robotique**, ou les **réseaux de neurones**. Selon la nature du problème, différentes méthodes de

résolution peuvent être utilisées. Parmi les approches déterministes, on trouve la *descente de gradient*, les *méthodes de Newton*, ou encore les *méthodes quasi-Newton* comme BFGS. Lorsqu'il s'agit de problèmes complexes, discontinus ou avec de nombreuses variables, on utilise des **méthodes heuristiques** ou **métaheuristiques**, comme les *algorithmes génétiques*, l'*optimisation par essaim de particules* (PSO) ou le *recuit simulé* [33].

2.6.4 Programmation convexe

La **programmation convexe** est un cas particulier d'*optimisation non linéaire* dans lequel la *fonction objectif est convexe* et les *contraintes définissent un ensemble convexe*. Cette configuration garantit que **tout minimum local est également un minimum global**, ce qui rend ce type de problème particulièrement attrayant tant sur le plan théorique que pratique.

Le problème général peut être formulé comme suit :

$$\text{Minimiser sous les contraintes} \begin{cases} f(x) \\ g_i(x) \leq 0, \quad i = 1, \ldots, m \\ h_j(x) = 0, \quad j = 1, \ldots, p \end{cases}$$

avec :

- $f : \mathbb{R}^n \to \mathbb{R}$ une fonction convexe,
- $g_i : \mathbb{R}^n \to \mathbb{R}$ des fonctions convexes,
- $h_j : \mathbb{R}^n \to \mathbb{R}$ des fonctions linéaires (ou affines).

Ce type de problème est omniprésent dans de nombreux domaines, notamment en **apprentissage automatique** (ex. : régularisation, régression logistique), en **finance** (optimisation de portefeuilles) et dans le **contrôle des systèmes dynamiques**. Les méthodes de résolution comprennent les **multiplicateurs de Lagrange**, les **conditions de Karush-Kuhn-Tucker (KKT)**, ainsi que les **méthodes de points intérieurs**, qui sont très efficaces pour les grands problèmes convexes [11].

2.6.5 Problèmes à plusieurs objectifs (multi-objectifs)

Les **problèmes d'optimisation multi-objectifs** visent à optimiser simultanément plusieurs fonctions objectif souvent en conflit. L'objectif est alors de déterminer un *ensemble de compromis* appelé **front de Pareto**, où une solution est dite *optimale au sens de Pareto* si aucune amélioration sur un des objectifs n'est possible sans en détériorer un autre [16]. Plusieurs approches existent pour obtenir ce front :

$$\textbf{Pondération :} \quad \begin{cases} F(x) = \sum_{k=1}^{p} w_k f_k(x), \\ w_k > 0 \end{cases}$$

$$\textbf{Contrainte } \varepsilon : \quad \begin{cases} \min\limits_{x} f_i(x), \\ \text{s.t.} \quad f_j(x) \leq \varepsilon_j, \quad j \neq i \end{cases}$$

$$\textbf{Évolutionnaires :} \quad \{\text{NSGA-II, SPEA2, MOEA/D}$$

Ces méthodes trouvent des applications en ingénierie multidisciplinaire, planification urbaine, gestion environnementale, etc.

Conclusion

Ce chapitre nous a permis d'explorer les principales techniques d'intelligence artificielle appliquées à l'optimisation mathématique, en passant en revue les approches classiques et modernes, ainsi que leurs applications dans la résolution de problèmes de graphes. Nous avons également présenté un état de l'art sur l'optimisation du chemin le plus court et les différentes formes d'optimisation.

Le chapitre suivant portera sur la conception de notre propre approche, en détaillant les algorithmes d'intelligence artificielle sélectionnés, les outils utilisés pour leur implémentation, ainsi que la méthode adoptée pour évaluer leurs performances.

CHAPITRE 3

Méthodologie

Après avoir établi, dans les chapitres précédents, un état de l'art approfondi des concepts, techniques et approches en lien avec notre problématique, ce chapitre est consacré à la phase de conception des algorithmes d'intelligence artificielle que nous avons retenus pour résoudre notre problème d'optimisation. L'objectif est de présenter les choix méthodologiques et techniques qui ont guidé notre démarche, en s'appuyant sur des critères à la fois théoriques et expérimentaux.

Section 3.1

Motivation et objectifs

La planification des emplois du temps dans les universités est une tâche complexe et cruciale, en raison du grand nombre de contraintes à prendre en considération. Ces contraintes peuvent être d'ordre pédagogique (comme l'indisponibilité de certains enseignants, les horaires préférentiels, ou les regroupements de cours), matériel (disponibilité des salles et des équipements spécifiques), ou encore organisationnel (éviter les conflits d'horaires, équilibrer les charges hebdomadaires, etc.). Avec la croissance constante du nombre d'étudiants, de filières et d'unités d'enseignement, cette complexité ne cesse d'augmenter, rendant la planification manuelle ou semi-automatique de plus en plus inefficace, chronophage, et sujette à l'erreur.

Dans ce contexte, il devient impératif de recourir à des solutions informatisées et intelligentes capables de produire des emplois du temps optimisés, tout en respectant l'ensemble des contraintes imposées. C'est dans cette optique que s'inscrit notre projet, dont l'objectif principal est de concevoir et de développer un système performant, basé sur des

algorithmes d'intelligence artificielle, capable de générer automatiquement des plannings fiables, équilibrés et adaptés aux réalités du terrain universitaire.

Plus précisément, nous avons choisi d'implémenter des algorithmes heuristiques et métaheuristiques, en raison de leur capacité démontrée à résoudre efficacement des problèmes complexes d'optimisation combinatoire. Les algorithmes heuristiques permettent de trouver rapidement des solutions acceptables, même dans des espaces de recherche vastes, tandis que les métaheuristiques offrent une meilleure exploration de l'espace de solutions en évitant les minima locaux. Ce choix se justifie donc par leur robustesse, leur flexibilité, et leur efficacité à produire des solutions quasi optimales dans des délais raisonnables, ce qui est essentiel dans un contexte réel de gestion universitaire.

Notre ambition à travers ce travail est non seulement de proposer un outil technique performant, mais aussi de contribuer à une meilleure gestion du temps et des ressources dans les établissements d'enseignement supérieur, en réduisant la charge de travail liée à la planification manuelle et en assurant une plus grande satisfaction des différents acteurs impliqués (enseignants, étudiants, administrateurs).

Section 3.2

Choix des algorithmes d'IA à étudier

Le choix des algorithmes d'IA à implémenter dans le cadre de notre projet reposera sur la nature du problème à résoudre, à savoir : la génération d'emplois du temps universitaires en respectant un ensemble complexe de contraintes. Ce type de problème étant classé parmi les problèmes d'optimisation combinatoire, le nombre de combinaisons possibles augmentera de manière exponentielle avec la taille des données (enseignants, groupes, modules, créneaux horaires, salles, journée libre). Il sera donc fondamental de s'appuyer sur des méthodes capables d'explorer efficacement l'espace des solutions possibles.

Dans ce contexte, nous opterons pour trois types d'algorithmes en raison de leur complémentarité et de leur pertinence face à notre problématique :

- **L'algorithme de force brute avec heuristique** sera sélectionné en premier lieu car il permettra de poser une base simple de génération aléatoire contrôlée. Bien qu'il ne garantisse pas toujours une solution optimale, il sera utile pour comprendre la structure du problème, tester la validité des données, et générer rapidement des emplois du temps partiellement satisfaisants. Il sera enrichi par des heuristiques de tri et un contrôle des conflits, dans le but de réduire l'espace de recherche et d'améliorer les chances de placement.
- **L'algorithme hybride basé sur la théorie des graphes** sera choisi pour sa capacité à représenter explicitement les conflits entre les séances sous forme de graphe. La technique de la coloration de graphe, centrale dans cette approche, permettra de

modéliser l'attribution des créneaux horaires de manière cohérente. L'approche sera dite "hybride" car elle combinera plusieurs techniques : un modèle mathématique (graphe de conflits), des stratégies heuristiques pour optimiser l'ordre de coloration, et des règles de filtrage intelligentes permettant de restreindre l'espace de recherche en fonction des contraintes spécifiques (type de salle, nombre maximal de séances par jour, journée libre). Ce choix répondra au besoin de mieux gérer les dépendances entre les cours et de limiter les conflits de manière plus structurée qu'avec une simple force brute.

- **L'algorithme génétique** sera intégré pour sa capacité à explorer de grands espaces de recherche tout en évitant les solutions locales. Grâce à des mécanismes inspirés de l'évolution naturelle (sélection, croisement, mutation), il permettra d'optimiser progressivement les emplois du temps en tenant compte d'une fonction de fitness adaptée à nos contraintes. Ce choix s'expliquera par la flexibilité de l'algorithme et sa capacité à s'adapter à différents types de contraintes (dures ou souples), ce qui en fera un excellent candidat pour traiter des cas complexes et améliorer la qualité des emplois du temps générés.

Ainsi, chaque algorithme sera retenu non seulement pour sa pertinence théorique, mais aussi pour sa capacité à apporter une solution concrète et complémentaire au problème de planification. Leur combinaison et leur comparaison permettront d'évaluer les compromis entre temps de calcul, qualité de solution, et respect des contraintes pédagogiques.

Section 3.3

Description détaillée des algorithmes sélectionnés

Dans cette section, nous présentons en détail les algorithmes retenus, en partant du traitement du dataset jusqu'à leurs principes de fonctionnement.

3.3.1 Les données

Pour la phase de développement et de conception, nous utiliserons la base de données interne de l'université, qui devra être entièrement sécurisée et confidentielle. À partir de celle-ci, nous extrairons un fichier Excel structuré en six colonnes :

- **Module** : intitulé du module.
- **Type** : nature de la séance [Cours (course), TD (DW), TP (PW)].
- **Section** : identifiant de la section (ex. : section1, section2, ...).
- **Groupe** : nom du groupe (ex. : G1, G2,... et $*$ si c'est un cours de toute la section).
- **Id** : identifiant unique de chaque séance.
- **Enseignant** : nom de l'enseignant responsable du module.

Lors de la phase de déploiement, une interface dédiée sera mise en place, permettant à un administrateur de saisir directement les données nécessaires.

Avant toute utilisation, un prétraitement simple des données sera effectué, consistant à supprimer les espaces inutiles en début et en fin de chaque nom de colonne. Cette opération sera réalisée à l'aide d'une boucle parcourant l'ensemble des colonnes et appliquant la fonction prédéfinie strip() de Python.

3.3.2 L'algorithme Force brute avec heuristique

L'algorithme de force brute (ou brute force) est une méthode de résolution de problèmes qui consiste à explorer systématiquement toutes les combinaisons possibles afin d'identifier une solution correcte. Il s'agit d'une stratégie simple à implémenter, mais souvent coûteuse en temps de calcul, car elle ne repose sur aucune optimisation préalable ni filtrage intelligent.

Dans le contexte de la génération d'un emploi du temps, une approche de force brute consiste à :

1. Tenter d'affecter chaque séance à un créneau horaire et à une salle disponible.
2. Tester plusieurs positions possibles pour chaque séance.
3. Vérifier à chaque tentative l'absence de conflits (enseignant, groupe, salle, etc.).
4. Répéter le processus jusqu'à ce qu'une affectation valide soit trouvée, ou abandonner après plusieurs essais inefficaces.

Dans notre cas, nous proposerons un algorithme qui agira selon une approche de force brute combinée à des heuristiques intelligentes afin de résoudre efficacement le problème. Cette méthode reposera sur une exploration partielle de l'espace des solutions, guidée par des règles de priorisation, ce qui permettra d'obtenir rapidement une solution réalisable.

L'algorithme se structurera autour de quatre étapes clés :

- Chargement et prétraitement des données.
- Tri heuristique des cours à placer : organisation des données selon des critères de priorité.
- Génération de l'emploi du temps par force brute guidée : tentative d'affectation progressive des séances à des créneaux valides.
- Calcul des jours libres.

3.3.2.1 Chargement et prétraitement des données

Avant d'exploiter les données, un prétraitement simple sera effectué : nous vérifierons d'abord que toutes les colonnes essentielles sont bien présentes dans le fichier de notre dataset. Ensuite, les données seront converties en une liste de dictionnaires, où chaque dictionnaire représente une séance à planifier. Cette structure permet de manipuler facilement les informations tout au long de l'algorithme.

Voici un exemple explicatif :

À partir de la ligne suivante du fichier Excel du dataset :

```
['Analyse1', 'dw', 'Section1', 'G1', '123', 'ens1']
```

Le dictionnaire Python suivant sera généré :

```
{
    'module': 'Analyse1',
    'type': 'dw',
    'section': 'Section1',
    'groupe': 'G1',
    'id': '123',
    'enseignant': 'ens1'
}
```

3.3.2.2 Tri heuristique des cours à placer

Avant de procéder à l'affectation des séances dans l'emploi du temps, une étape de tri sera appliquée aux données. Cette étape constituera une heuristique de priorisation. Bien qu'elle ne garantisse pas une solution optimale, elle permettra d'améliorer l'efficacité de l'algorithme de force brute en traitant en premier les cas les plus contraignants. L'objectif sera de réduire les risques de conflits lors de l'affectation et de guider la recherche vers des solutions plus stables.

La liste des séances à planifier sera triée selon deux critères principaux :

1. **Fréquence de l'enseignant** : les enseignants ayant le plus grand nombre de séances à assurer seront traités en premier. Cela permettra de leur réserver des créneaux adéquats dès le début, ce qui contribuera à éviter les conflits futurs. Il s'agit d'une heuristique basée sur la charge des ressources critiques.
2. **Type de séance** : en cas d'égalité sur le nombre de séances par enseignant, un second critère sera pris en compte : le type de séance (cours, TD, TP), afin de maintenir une certaine cohérence ou priorité pédagogique.

La fonction suivante illustre le fonctionnement du tri.

Fonction `Trier_Cours`

Fonction :
Entrée : cours : liste de dictionnaires représentant les séances à planifier ;
Sortie : cours_trié : liste de dictionnaires triée ;

Début
Si *cours est vide* **Alors**
Retourner liste_vide
Fin
cours_trié ← liste triée de cours selon les critères suivants ;
Pour *chaque cours dans cours* **Faire**
nb ← nombre de séances ayant le même enseignant ; critere_tri ← (-nb, type) ; `▷ ordre décroissant (on met un signe moins pour que les enseignants les "plus chargés soient traités en premier").`
`▷ en cas d'égalité sur le nombre de séances, on trie ensuite selon le type de séance (cours, td, tp), en ordre "alphabétique croissant".`
Fin
Appliquer le tri sur la liste cours en utilisant `critere_tri` comme clé (fonction `sorted` de Python) ;
Retourner cours_trié ;

Algorithme 3 – Fonction `Trier_Cours`

3.3.2.3 Génération de l'emploi du temps par force brute guidée

Ce sera l'étape principale de notre algorithme, où les séances seront placées dans l'emploi du temps tout en respectant les contraintes définies. Elle reposera sur une stratégie de force brute guidée par une heuristique, ce qui signifiera que, pour chaque séance, plusieurs tentatives seront effectuées afin de lui attribuer un créneau et une salle sans provoquer de conflit. En cas d'échec après un certain nombre de tentatives (MAX_TENTATIVES), la séance sera placée dans une liste à part pour être traitée ultérieurement.

L'algorithme commencera par trier les cours selon une heuristique de priorité (en plaçant les enseignants les plus chargés en premier), puis tentera de les placer aléatoirement dans les créneaux et salles disponibles. À chaque tentative, au lieu de choisir un seul créneau aléatoire, il génère une liste exhaustive des créneaux possibles, qu'il mélange aléatoirement pour conserver une exploration variée, puis teste chaque créneau dans cet ordre avant de passer au suivant.

Lors de chaque placement, plusieurs types de conflits sont vérifiés : deux séances ne peuvent pas être assignées au même enseignant dans un même créneau ; un groupe d'étu-

diants ne peut pas avoir deux séances simultanément ; une salle ne peut accueillir qu'un seul cours à la fois pour un même type d'activité (cours, TD ou TP) ; et, en particulier, lorsqu'un cours est destiné à toute une section, aucun TD ni TP ne peut être planifié pour les groupes de cette section au même créneau. Ces contraintes permettent d'assurer la cohérence globale de l'emploi du temps généré, tout en limitant les conflits les plus critiques.

L'algorithme suivant est chargé de la génération du planning :

Algorithme : `generer_emploi_du_temps`

```
ALGORITHME :
    Entrée : cours : Liste de dictionnaires ;
    Sortie : planning, cours_non_places : liste de dictionnaires ;
    Variables :
    MAX_TENTATIVES, NB_CRENEAUX, NB_SALLES, JOURS_SEMAINE, CRENEAUX_PAR_JOUR : Constantes ;
    place : Booléen ;
    Début
    Si cours est vide Alors
        Retourner liste_vide, liste_vide ;
    cours_tries ← trier_cours(cours) ; ▷ Fonction présentée dans la partie précédente
    planning ← liste_vide ;
    cours_non_places ← liste_vide ;
    Pour chaque cours dans cours_tries Faire
        place ← FAUX ;
        salle_max ← NB_SALLES selon le type de cours ;
        Pour tentative de 1 à MAX_TENTATIVES Faire
            creneaux ← liste de 1 à NB_CRENEAUX ;
            mélanger aléatoirement creneaux ;
            Pour chaque creneau dans creneaux Faire
                Pour salle de 1 à salle_max Faire
                    conflit ← FAUX ;
                    Pour chaque c dans planning Faire
                        Si ( creneau == c.creneau ET ( cours.enseignant == c.enseignant
                          OU (cours.groupe == c.groupe ET cours.groupe != '*'
                          ET c.groupe != '*') OU (cours.section == c.section ET
                          (cours.groupe == '*' OU c.groupe == '*')) OU (cours.section
                          == c.section ET ( (cours.groupe == * ET c.groupe != '*') OU
                          (c.groupe == '*' ET cours.groupe != '*') )) OU (cours.type ==
                          c.type ET salle == c.salle) ) ) alors
                            conflit ← VRAI ;
                            SORTIR ; ▷ sortir de la boucle vérification des conflits
                        SORTIR ; ▷ sortie salle
                    Si place Alors
                        SORTIR ; ▷ sortie créneau
                Si place Alors
                    SORTIR ; ▷ sortie tentative
            Si non place Alors
                Ajouter cours à cours_non_places ;
    Retourner planning, cours_non_places ;
```

Algorithme 4 – Algorithme `generer_emploi_du_temps`

3.3.2.4 Calcul des jours libres

Après la génération de l'emploi du temps, il sera important de s'assurer que chaque section bénéficiera d'au moins une journée libre dans la semaine, conformément aux contraintes pédagogiques. La fonction calculer_jours_libres s'occupera de cette tâche.

Son objectif sera de déterminer, pour chaque groupe, les jours où il aura des cours, puis de déduire un jour libre commun à tous les groupes d'une même section. Ce jour sera ensuite enregistré dans chaque séance de la section. Si aucun jour commun n'est trouvé (cas rare), la valeur 'Aucun' sera attribuée.

Cette fonction reposera sur un raisonnement en deux étapes :

1. Le calcul des jours occupés pour chaque groupe en parcourant toutes les séances du planning.
2. La déduction du jour libre commun pour chaque section, puis la mise à jour du planning.

Voici son fonctionnement :

Fonction `calculer_jours_libres`

ALGORITHME :
Entrée : planning : Liste de dictionnaires ;
JOURS_SEMAINE : Liste de chaînes de caractères ;
Sortie : planning avec journées libres ;
Début
jours_utilises ← dictionnaire avec ensemble vide par groupe ;
Pour *chaque cours dans planning* **Faire**
  Si *cours.groupe* != *'*'* **Alors**
    groupes ← [cours.groupe] ;
  Sinon
    groupes ← [c.groupe **Pour** c DANS planning **Si** c.section == cours.section **ET** c.groupe != '*'] ;
  Pour *chaque groupe dans groupes* **Faire**
    jours_utilises[groupe].ajouter(cours.jour)
    ▷ `récupérer les jours où les groupes ont des séances`
jours_libres ← dictionnaire vide ;
sections ← ensemble des sections DANS le planning ;
Pour *chaque section dans sections* **Faire**
  groupes_section ← liste des groupes de cette section ;
  jours_libres_section ← ensemble de tous les JOURS_SEMAINE ;
  Pour *chaque groupe DANS groupes_section* **Faire**
    jours_libres_section ← jours_libres_section − jours_utilises[groupe] ;
  Si *jours_libres_section n'est pas vide* **Alors**
    jour_libre ← premier élément de jours_libres_section ;
  Sinon
    jour_libre ← 'Aucun' ;
  Pour *chaque cours DANS planning* **Faire**
    Si *cours.section == section* **Alors**
      cours.journee_libre_section ← jour_libre ;
      Si *cours.groupe* != *'*'* **Alors**
        cours.journee_libre_groupe ← jour_libre ;
      Sinon Si *groupes_section n'est pas vide* **Alors**
        cours.journee_libre_groupe ← jour_libre ;
Retourner planning ;
Fin

Algorithme 5 – Algorithme calculer_jours_libres

3.3.3 L'algorithme hybride fondé sur la coloration en théorie des graphes

Le cœur de notre algorithme reposera sur les principes de la théorie des graphes, en particulier sur l'algorithme de coloration. L'idée générale consiste à modéliser chaque séance à planifier comme un sommet dans un graphe, et à établir des arêtes entre les séances incompatibles, c'est-à-dire celles qui ne peuvent pas être programmées simultanément. L'implémentation de cet algorithme se structurera en quatre étapes principales :

1. Construction du graphe de conflits.
2. Application de l'algorithme de coloration.
3. Affectation des salles selon le type de séance.
4. Prise en compte des contraintes pédagogiques supplémentaires.

3.3.3.1 La construction du graphe de conflits

À l'aide de la bibliothèque NetworkX de Python, nous modéliserons chaque séance (cours, TD, TP) comme un nœud du graphe, en lui associant ses propriétés (type, groupe, enseignant, section et identifiant). Deux nœuds sont reliés par une arête dès qu'un conflit d'incompatibilité est détecté entre eux, c'est-à-dire lorsqu'ils ne peuvent pas se dérouler au même moment. Ces conflits incluent notamment :

- l'implication du même enseignant dans les deux séances,
- l'appartenance au même groupe (sauf dans le cas d'un cours destiné à toute une section),
- la présence d'un cours pour une section en parallèle avec un TD ou un TP destiné à un groupe de cette même section.

Ce graphe constituera ainsi une représentation abstraite des conflits, indispensable pour appliquer efficacement la coloration et garantir un emploi du temps sans chevauchements.

Cet algorithme décrit la construction de notre graphe :

Algorithme `Construction_Graphe_Conflits`

Variables :
G : graphe vide ;
cours_list : Liste de dictionnaires contenant les données des séances (chargées depuis le dataset) ;
Début
Pour *chaque cours dans cours_list* **Faire**
node_id ← concaténation de (id, Module, Groupe, Section, Enseignant) ;
G.ajouterNoeud(node_id, attributs = cours) ;
Fin
Pour *chaque paire (cours1, cours2) dans G.noeuds* **Faire**
Si `Conflit(`*cours1, cours2*`)` **Alors**
G.ajouterArete(cours1, cours2) ;
Fin
Fin

Fonction `Conflit(`*cours1, cours2*`)` **:**
Si *cours1.Enseignant == cours2.Enseignant* **Alors**
Retourner Vrai ;
Fin
Si *cours1.Groupe == '*'* **et** *cours1.Section == cours2.Section* **Alors**
Retourner Vrai ;
Fin
Si *cours2.Groupe == '*'* **et** *cours2.Section == cours1.Section* **Alors**
Retourner Vrai ;
Fin
Si *cours1.Groupe* != *'*'* **Alors**
groupes1 ← [cours1.Groupe] ;
Fin
Sinon
groupes1 ← Tous les groupes de cours1.Section ;
Fin
Si *cours2.Groupe* != *'*'* **Alors**
groupes2 ← [cours2.Groupe] ;
Fin
Sinon
groupes2 ← Tous les groupes de cours2.Section ;
Fin
Si *groupes1* ∩ *groupes2* != ∅ **Alors**
Retourner Vrai ;
Fin
Retourner Faux ;
Fin

Algorithme 6 – Algorithme `Construction_Graphe_Conflits`

3.3.3.2 Application de l'algorithme de coloration

Une fois la structure du graphe de contraintes établie, nous mettrons en œuvre un algorithme de coloration optimisé reposant sur une heuristique gloutonne à base de degré décroissant, conçue pour répondre efficacement aux contraintes pratiques de la génération d'un emploi du temps. Il attribuera à chaque nœud du graphe (représentant une séance d'enseignement) une couleur correspondant à un créneau horaire, de manière à ce qu'aucun deux nœuds adjacents (en conflit) ne partagent la même couleur. Pour ce faire, les nœuds seront d'abord triés selon leur degré décroissant, ce qui donnera la priorité aux séances ayant le plus de conflits potentiels (voisins dans le graphe). Ce tri initial permettra de limiter les risques de blocage lors des étapes suivantes.

Pour chaque nœud, l'algorithme identifiera les couleurs déjà utilisées par ses voisins, puis sélectionnera la plus petite couleur disponible parmi celles restantes. Ce choix contribuera à réduire le nombre total de créneaux utilisés, ce qui est essentiel pour optimiser l'utilisation des ressources, telles que les salles et les plages horaires. En cas d'indisponibilité de couleurs (situation rare si le nombre de créneaux est bien dimensionné), une couleur sera choisie de manière aléatoire, et un avertissement sera généré pour signaler cette situation.

Bien qu'inspirée du principe de saturation, cette méthode ne calculera pas dynamiquement le nombre de couleurs différentes déjà attribuées à ses voisins, mais suivra plutôt une stratégie gloutonne simple et rapide. Cette approche offrira un compromis efficace entre simplicité d'implémentation, rapidité d'exécution et respect des contraintes de base (enseignants, groupes, salles). Elle sera également capable de traiter des graphes de grande taille, ce qui la rendra particulièrement adaptée aux structures universitaires complexes où le nombre de séances et de contraintes est élevé. Enfin, elle restera suffisamment flexible pour intégrer des règles supplémentaires si nécessaire.

L'algorithme ci-après illustre le fonctionnement du processus de coloration :

Algorithme : `Coloration_Optimisée`

```
ALGORITHME
    Entrée : G : graphe ; NB_CRENEAUX : constante ;
    Sortie : coloration : dictionnaire ;
    Variables
        noeuds_tries ← liste vide ;
        coloration ← Dictionnaire (noeud → couleur) ;
    couleurs_disponibles ← {1..NB_CRENEAUX} ;
    Début
    noeuds_tries ← Trier(G.noeuds) ;
    ▷ par degré décroissant
    Pour chaque noeud dans noeuds_tries Faire
        couleurs_interdites ← ensemble vide ;
        Pour chaque voisin dans G.voisins(noeud) Faire
            Si voisin dans coloration Alors
                couleurs_interdites.ajouter(coloration[voisin]) ;
            Fin
        Fin
        couleurs_possibles ← couleurs_disponibles − couleurs_interdites ;
        Si couleurs_possibles non vide Alors
            couleur ← min(couleurs_possibles) ;
        Fin
        Sinon
            couleur ← Random(couleurs_disponibles) ;
        Fin
        coloration[noeud] ← couleur ;
    Fin
    Retourner coloration ;
Fin
```

Algorithme 7 – Algorithme `Coloration_Optimisée`

3.3.3.3 Affectation des salles selon le type de séance

Après la phase de coloration, nous passerons à l'affectation des salles, en tenant compte du type de séance (cours, TD, TP), chacun étant associé à un ensemble de salles distinctes, définies en nombre par des paramètres spécifiques (NB_SALLES_COURS, NB_SALLES_DW, NB_SALLES_PW).

L'algorithme parcourra l'ensemble des séances ayant déjà reçu un créneau horaire. Pour chacune, il vérifiera la disponibilité des salles compatibles avec son type au créneau attribué. Il sélectionnera la première salle libre et l'associera à la séance. Ce choix suit une heuristique de premier ajustement (first-fit), couramment utilisée dans les problèmes d'allocation de ressources ou de bin packing : le programme parcourt toutes les salles disponibles pour

le type de cours concerné et assigne simplement la première salle libre, en s'arrêtant dès qu'une solution acceptable est trouvée.

La séance est ensuite intégrée à la structure finale de l'emploi du temps, en y précisant le jour, l'heure de début (calculée à partir de l'index du créneau), la salle, et les informations pédagogiques associées. Dans le cas contraire – c'est-à-dire si aucune salle adaptée n'est libre à ce créneau – la séance sera temporairement mise de côté et enregistrée dans une liste des cours non placés. Ce mécanisme permettra d'identifier les séances posant problème, en vue d'un traitement ultérieur ou d'un ajustement des ressources.

Grâce à cette logique, l'algorithme garantira que deux séances ne seront jamais affectées à la même salle au même créneau, tout en optimisant l'occupation des ressources. De plus, cette approche permet une extension facile pour intégrer d'autres contraintes, comme des préférences d'enseignants ou des restrictions de salles particulières. Voici l'algorithme utilisé pour faire l'affectation des salles :

Algorithme : `Affectation_Salles`

```
ALGORITHME
    Entrée : coloration : dictionnaire des cours avec leurs créneaux attribués ;
    G : graphe dont les nœuds contiennent les informations sur les cours ;
    Sortie : emploi_final : liste de dictionnaires (emplois du temps placés) ;
    cours_non_places : liste des cours non placés faute de salle disponible ;
    Variables :
    salles_utilisees : dictionnaire associant chaque créneau à l'ensemble des salles occupées ;
    nb_salles : entier ;
    salle_trouvee : booléen ;
    Début
    Initialiser salles_utilisees comme dictionnaire vide
    Initialiser emploi_final comme liste vide
    Initialiser cours_non_places comme liste vide
    Pour chaque (cours_id, creneau) dans coloration Faire
        cours_data ← G.attributs(cours_id) ;
        type_salle ← cours_data.Type.lower() ;
        ▷ récupérer les types en minuscule
        Si type_salle == "course" Alors
            nb_salles ← NB_SALLES_COURS ;
        Fin
        Sinon Si type_salle == "dw" Alors
            nb_salles ← NB_SALLES_DW ;
        Fin
        Sinon
            nb_salles ← NB_SALLES_PW ;
        Fin
        salle_trouvee ← Faux ;
        Pour s de 1 à nb_salles Faire
            Si (creneau, s) non dans salles_utilisees[creneau] Alors
                salles_utilisees[creneau].ajouter((creneau, s)) ;
                salle_trouvee ← Vrai ;
                ▷ Conversion créneau → jour/heure
                jour_index ← (creneau - 1) div CRENEAUX_PAR_JOUR ;
                heure_index ← (creneau - 1) mod CRENEAUX_PAR_JOUR ;
                jour ← JOURS_SEMAINE[jour_index] ;
                heure_debut ← 8h00 + heure_index × 90 min ; ▷ durée d'une séance 1h30
                emploi_final.ajouter({
                    "ID" : cours_data.id ;
                    "Module" : cours_data.Module ;
                    "Jour" : jour ;
                    "Heure" : heure_debut ;
                    "Salle" : "Salles" ;
                    "Section" : cours_data.section ;
                "Groupe" : cours_data.groupe ; })
            Fin
        Fin
        Si non salle_trouvee Alors
            cours_non_places.ajouter(cours_id) ;
        Fin
    Fin
```

Algorithme 8 – Algorithme `Affectation_Salles`

3.3.3.4 Prise en compte des contraintes pédagogiques supplémentaires

Après l'attribution des créneaux horaires et des salles aux différentes séances, l'algorithme appliquera deux contraintes pédagogiques supplémentaires, destinées à améliorer la qualité et l'équilibre de l'emploi du temps généré, à savoir :

- Un groupe ne devra pas avoir plus de quatre (4) séances par jour.
- Tous les groupes d'une même section bénéficieront de la même journée libre.

Concernant la contrainte du nombre de séances maximum par jour, bien qu'elle ne sera pas directement codée dans la génération initiale, elle sera implicitement gérée via la coloration de graphe et le nombre total de créneaux disponibles. Chaque jour étant divisé en 6 créneaux (1h30 chacun), le nombre de séances pour un groupe ou un enseignant ne dépassera généralement pas 4 par jour, car le graphe empêchera que plusieurs cours impliquant le même enseignant ou groupe soient placés au même moment. Si un contrôle strict de la limite de 4 séances est requis, une vérification ou une optimisation supplémentaire devra être ajoutée après la coloration pour inspecter la répartition des créneaux par groupe par jour, et réassigner certains cours en cas de dépassement.

La vérification s'effectuera à l'aide de cette fonction :

Fonction `Verifier_Limite_Seances`

```
Fonction :
    Entrée : emploi_final : liste de dictionnaires (ID, Module, Groupe, Jour, etc.) ;
      ▷ résultat des trois étapes
    Sortie : messages d'erreur ;
    groupes ← Ensemble des groupes dans emploi_final ;
    Pour chaque groupe dans groupes Faire
        Pour chaque jour dans JOURS_SEMAINE Faire
            nb_seances ← Compter les éléments de emploi_final où Groupe =
              groupe et Jour = jour ;
            Si nb_seances > 4 Alors
                Afficher ("Erreur : Le groupe", groupe , "a" , nb_seances , "séances le"
                  , jour) ;
            Fin
        Fin
    Fin
Fin
```

Algorithme 9 – Fonction `Verifier_Limite_Seances`

Le traitement de la journée libre reposera sur l'analyse des jours occupés par chaque groupe en fonction de l'emploi du temps qui sera généré. Une fois les jours où chaque groupe aura des séances identifiés, on en déduira les jours potentiellement libres. Pour assurer une cohérence pédagogique, tous les groupes appartenant à une même section

devront partager un même jour libre. Le programme calculera donc l'intersection des jours libres de tous les groupes d'une section, puis fixera ce jour comme étant le jour de repos commun. Cette information sera ensuite ajoutée au tableau final de l'emploi du temps, et une vérification sera effectuée pour garantir que tous les groupes d'une section respecteront bien ce jour de repos commun.

La fonction suivante permettra de déterminer les jours libres :

Fonction `Calculer_Jour_Libre_Section`

```
Fonction :
    Entrée : section : chaine de caractères ;
    Sortie : jour : chaîne de caractères ;
    groupes ← ListeGroupes(section) ;
    jours_occupes ← Union des jours utilisés par tous les groupes ;
    Pour chaque jour dans JOURS_SEMAINE Faire
        Si jour non dans jours_occupes Alors
            Retourner jour ;
        Fin
    Fin
    Retourner "Aucun" ;
Fin
```

Algorithme 10 – Fonction `Calculer_Jour_Libre_Section`

3.3.4 L'algorithme génétique

Après notre choix de la coloration du graphe avec heuristiques, nous proposerons sa coloration avec l'algorithme génétique. Ce sera une combinaison entre la théorie des graphes et les métaheuristiques, dans le but d'explorer plus efficacement l'espace des solutions et de surmonter les limitations des méthodes purement déterministes. Cette approche repose sur trois étapes principales :

1. La construction du graphe représentant les conflits d'emploi du temps.
2. L'application de l'algorithme génétique pour réaliser la coloration du graphe, c'est-à-dire l'affectation des créneaux horaires aux différentes séances.
3. Affectation des salles et des journées libres, en respectant les contraintes des groupes et des sections.

3.3.4.1 La construction du graphe

Cette étape reprend intégralement le processus de construction du graphe décrit dans l'approche précédente. Le graphe ainsi obtenu est préparé pour être coloré lors de l'étape

suivante.

3.3.4.2 L'application de l'algorithme génétique pour la coloration

L'algorithme génétique utilisé dans notre solution reposera sur les principes de l'évolution naturelle. Il explorera l'espace des solutions possibles à travers des générations successives d'individus, chacun représentant une solution candidate à l'affectation des séances à des créneaux horaires. L'objectif sera de minimiser les conflits d'emploi du temps tout en optimisant l'utilisation des créneaux et des salles.

Cette approche suivra les étapes classiques d'un algorithme génétique : génération de la population initiale, évaluation des individus, sélection, croisement, mutation et itération sur plusieurs générations. Chaque solution possible sera représentée par un individu, c'est-à-dire une affectation complète de créneaux horaires à l'ensemble des séances.

À chaque génération, les solutions les plus prometteuses seront conservées, croisées et légèrement modifiées, dans l'espoir de converger vers une configuration optimale. Nous détaillerons ci dessous chacune de ces étapes :

1. **Représentation des individus**
   Chaque individu de la population correspondra à une solution potentielle, c'est-à-dire à une affectation de créneaux horaires à toutes les séances. Cette représentation se fait sous forme d'une liste : la position i de la liste correspond au i-ème nœud du graphe, et la valeur stockée indique le créneau horaire attribué à cette séance.
2. **Évaluation des individus**
   La fonction d'évaluation calculera trois métriques clés :
   (a). **Nombre de conflits :** Compte les arêtes du graphe où les deux nœuds ont le même créneau (pénalisé).
   (b). **Nombre de créneaux utilisés :** Minimisé pour une meilleure utilisation des ressources.
   (c). **Pénalité des salles :** Ce critère vérifie la disponibilité des salles selon le type de séance (cours, TD, TP). Cette vérification est nécessaire, car sans elle, l'algorithme pourrait placer plus de séances simultanées que le nombre réel de salles disponibles. Par exemple, il pourrait affecter plus de sept cours au même créneau, alors que seulement sept amphithéâtres sont disponibles, rendant ainsi l'emploi du temps irréalisable.

Fonction `Evaluer`

```
Fonction :
    Entrée : individu : liste ;
    Sortie : conflits, nb_creneaux, penalite_salles : entier ;
    Début
    conflits ← 0 ;
    penalite_salles ← 0 ;
    salles_utilisees ← dictionnaire vide de la forme [creneau][type] = ensemble
    des salles utilisées ;
    nb_creneaux ← nombre de valeurs distinctes dans individu ;
    Pour i de 0 à longueur(individu) - 1 Faire
        node ← i-ème nœud dans G ;
        creneau ← individu[i] ;
        type_cours ← type associé au nœud dans G (converti en minuscules) ;
        // Vérification des conflits
        Pour chaque voisin v de node dans G Faire
            Si individu[node_to_index[v]] == creneau Alors
                conflits ← conflits + 0.5 ;
            Fin
        Fin
        // Vérification des salles
        salles_dispo ← NB_SALLES[type_cours] ou 1 si non défini ;
        salle_trouvee ← Faux ;
        Pour s allant de 1 à salles_dispo Faire
            Si s ∉ salles_utilisees[creneau][type_cours] Alors
                salle_trouvee ← Vrai ;
                Ajouter s à salles_utilisees[creneau][type_cours] ; Quitter ;
            Fin
        Fin
        Si salle_trouvee == Faux Alors
            penalite_salles ← penalite_salles + 1 ;
        Fin
    Fin
    Retourner (conflits, nb_creneaux, penalite_salles) ;
```

Algorithme 11 – Fonction `Evaluer`

3. **Mutation**

La mutation consistera à modifier aléatoirement le créneau attribué à une séance (un sommet du graphe). Cette étape permettra d'introduire de la diversité dans la population, évitant ainsi une convergence prématurée vers une solution sous-optimale. Plutôt qu'une mutation purement aléatoire, l'opérateur :

(a). Identifie d'abord les nœuds impliqués dans des conflits

(b). Modifie prioritairement ces nœuds

(c). Choisit un nouveau créneau qui réduit les conflits locaux

Fonction `Muter`

```
Fonction :
    Entrée : individu : liste ;
    Sortie : individu muté : liste ;
    Début
    conflits ← liste vide.
    Détecter les nœuds en conflit
    Pour i de 0 à longueur(individu) - 1 Faire
        node ← i-ème nœud dans G ;
        Pour chaque voisin v de node Faire
            Si individu[i] == individu[node_to_index[v]] Alors
                Ajouter i à conflits ;
                Quitter ;
            Fin
        Fin
    Fin
    Si conflits n'est pas vide Alors
        idx ← élément aléatoire de conflits ;
        node ← idx-ième nœud dans G ;
        voisins ← indices des voisins de node ;
        creneaux_voisins ← ensemble des individu[v] pour chaque v dans voisins ;
        creneaux_possibles ← liste des entiers de 1 à NB_CRENEAUX ne figurant
        pas dans creneaux_voisins ;
        Si creneaux_possibles n'est pas vide Alors
            individu[idx] ← élément aléatoire de creneaux_possibles ;
        Fin
        Sinon
            individu[idx] ← nombre aléatoire entre 1 et NB_CRENEAUX ;
        Fin
    Fin
    Retourner individu ;
Fin
```

Algorithme 12 – Fonction `Muter`

4. **Croisement**

Le croisement sera réalisé entre deux individus à l'aide d'un croisement à deux points (Two-Point Crossover). Il consistera à échanger des segments de la représentation entre deux parents pour générer deux enfants. Cette opération favorisera la combinaison des bonnes caractéristiques des deux parents.

Fonction `Croiser`

Fonction :
Entrée : parent1, parent2 : listes ;
Sortie : enfant1, enfant2 : listes ;
Début
p1, p2 ← deux indices aléatoires tels que $p1 < p2$;
enfant1 ← copie de parent1 ;
enfant2 ← copie de parent2 ;
Pour *i de* $p1$ *à* $p2$ **Faire**
échanger enfant1[i] enfant2[i] ;
Fin
Retourner (enfant1, enfant2) ;
Fin

Algorithme 13 – Fonction `Croiser`

5. **Sélection**

La sélection des individus pour la génération suivante se fera par tournoi (tournament selection). Cette méthode sélectionnera les meilleurs individus parmi des sous-groupes de la population, favorisant ainsi ceux ayant une meilleure fitness tout en conservant une certaine diversité.

Fonction `Selectionner`

Fonction :
Entrée : population : liste ;
taille du tournoi : entier ;
Sortie : sélection : liste ;
Début
sélection ← liste vide ;
Pour *i de 1 à taille* **Faire**
tournoi ← sous-ensemble aléatoire de la population de taille du tournoi ;
meilleur ← individu du tournoi avec la plus petite fitness ;
ajouter meilleur à sélection ;
FinPour
Retourner sélection ;
Fin

Algorithme 14 – Fonction `Selectionner`

6. **Boucle principale de l'algorithme**

La fonction principale constituera le cœur du processus évolutif. Elle initialisera une population de solutions candidates (individus) et leur attribuera une valeur de fitness à l'aide de la fonction d'évaluation. À chaque génération, l'algorithme sélectionnera les meilleurs individus (élites), puis générera de nouveaux descendants par croisement et mutation. Seuls les individus modifiés seront réévalués afin d'optimiser les performances. Ensuite, la population sera mise à jour avec les nouveaux individus générés. L'évolution se poursuivra pendant un nombre défini de générations.

Parallèlement, des statistiques seront collectées à chaque étape, et la meilleure solution rencontrée jusqu'à présent sera conservée dans une structure appelée **Hall of Fame** (abrégée hof), fournie par la bibliothèque **DEAP** (Distributed Evolutionary Algorithms in Python). Cette structure permet de mémoriser les meilleurs individus tout au long de l'exécution, indépendamment de leur présence dans la population courante. Enfin, la solution optimale trouvée sera retournée à l'issue du processus évolutif..

Fonction `exécuter_algorithme_génétique`

Fonction :
- **Entrée :** aucune ;
- **Sortie :** meilleur_individu : une solution représentée par une liste de créneaux ;
- **Début**
- 1. Initialisation
- **Initialiser** la population `pop` avec `POPULATION_SIZE` individus ;
- **Initialiser** `hof` comme Hall of Fame pour stocker les 5 meilleurs individus ;
- 2. Évaluation initiale
- **Pour** *chaque individu de `pop`* **Faire**
  - **evaluer** (individu) ;
  - Stocker sa valeur de fitness ;
- 3. Boucle principale d'évolution
- **Pour** *gen de 1 à `GENERATIONS`* **Faire**
  - Sélectionner les 5 meilleurs individus comme `elite` ;
  - Sélectionner le reste des individus avec une stratégie de sélection ;
  - **Fusionner** `elite` avec les individus sélectionnés pour créer `offspring` ;
  - 4. Croisement
  - **Pour** *chaque paire (child1, child2) dans `offspring`* **Faire**
    - **Si** *une probabilité aléatoire* < *`CX_PROB`* **Alors**
      - **Croiser** (child1, child2) ;
      - **Supprimer** leur fitness ;
  - 5. Mutation
  - **Pour** *chaque mutant dans `offspring`* **Faire**
    - **Si** *une probabilité aléatoire* < *`MUT_PROB`* **Alors**
      - **muter** (mutant) ;
      - **Supprimer** sa fitness ;
  - 6. Réévaluation des individus modifiés
  - **Pour** *chaque individu invalide dans `offspring`* **Faire**
    - **evaluer** (individu) ;
    - **Mettre à jour** sa fitness ;
  - 7. Mise à jour de la population
  - Remplacer la population `pop` par `offspring` ;
  - 8. Mise à jour du Hall of Fame
  - **Mettre à jour** `hof` avec les meilleurs individus de `pop` ;
  - 9. Enregistrement des statistiques de la génération
  - **Enregistrer** les statistiques de la génération `gen` ;
- **Retourner** le meilleur individu dans `hof` ;
- **Fin**

Algorithme 15 – Fonction `exécuter_algorithme_génétique`

3.3.5 L'affectation des salles et des journées libres

Dans cette partie, nous appliquerons le même traitement que celui utilisé dans notre deuxième approche. L'étape d'affectation des salles s'effectuera après la génération du planning par coloration de graphe. Chaque cours déjà placé dans un créneau horaire recevra une salle compatible avec son type (cours, TD, TP). Le script utilisera une structure de dictionnaire pour suivre, pour chaque créneau, les salles déjà occupées, ce qui permettra d'éviter toute collision. Pour chaque séance, l'algorithme parcourra les salles disponibles correspondant au type de la séance, et attribuera la première salle libre trouvée dans le créneau concerné. Une fois la salle assignée, l'ensemble des informations (jour, heure, salle, enseignant, groupe, module.) sera compilé et ajouté à l'emploi du temps final. Si aucune salle compatible ne sera disponible dans le créneau prévu, la séance sera temporairement placée dans une liste spéciale en attente de traitement.

La phase d'affectation des journées libres reposera quant à elle sur une fonction dédiée qui analysera les jours d'occupation de chaque groupe à partir du planning généré. Dans un premier temps, elle construira une structure regroupant les groupes par section, puis recensera, pour chaque groupe, les jours durant lesquels des séances seront programmées. Ensuite, l'algorithme tentera d'identifier un jour libre commun à tous les groupes d'une même section.

Section 3.4

Méthode de validation et de mesure des performances

Afin d'évaluer l'efficacité et la robustesse de nos algorithmes de génération d'emploi du temps, nous utiliserons une méthode de validation reposant sur plusieurs métriques quantitatives et qualitatives. Ces indicateurs permettront de comparer objectivement les performances des approches choisies.

- **Taux de placement** : il mesurera la proportion de cours placés par rapport au total à traiter. Un taux élevé indiquera une bonne capacité de l'algorithme à satisfaire les contraintes.
- **Cohérence des jours libres** : nous vérifierons que tous les groupes appartenant à une même section bénéficieront d'un jour libre commun, et nous identifierons les éventuelles incohérences.
- **Distribution des créneaux** : nous analyserons l'écart-type de l'occupation horaire pour détecter les déséquilibres temporels (comme une surcharge à certains moments de la semaine), ainsi que le nombre de créneaux **saturés**, c'est-à-dire ceux dépassant 80 % de la capacité totale des salles disponibles. Cela permettra d'évaluer la dispersion et l'efficacité de la répartition des cours sur la semaine.
- **Taux de conflits résolus** (dans les approches à base de graphe) : à partir du graphe de

conflits, nous calculerons le pourcentage d'arêtes correctement colorées, représentant les cours incompatibles placés à des créneaux différents.

En complément, nous sauvegarderons automatiquement des visualisations graphiques pour illustrer les résultats obtenus : distribution des cours par créneau, charge par enseignant et taux d'utilisation des salles.

Un bon emploi du temps généré se caractérisera par :

- Un taux de placement proche de 100 % ;
- Un faible écart-type dans la répartition des créneaux ;
- Aucun conflit dans le graphe ;
- Et aucune incohérence dans les jours libres.

Ces résultats seront accompagnés de graphiques de performance qui seront automatiquement générés et sauvegardés afin de faciliter l'analyse comparative entre les différentes approches.

Conclusion

Dans ce chapitre, nous avons exposé en détail la conception des approches retenues pour la génération de l'emploi du temps. Chaque méthode a été présentée avec son principe fondamental ainsi que les différentes étapes qui la composent. Le chapitre suivant sera consacré aux aspects liés à l'implémentation pratique et à l'expérimentation de ces algorithmes.

CHAPITRE 4

Résultats et Discussions

Dans ce chapitre, nous présenterons les aspects liés au développement de nos algorithmes, ainsi que les résultats issus de leur évaluation sur différents jeux de données. Nous décrirons également l'interface développée pour faciliter le déploiement et l'utilisation de nos programmes.

Section 4.1

Environnement de développement

4.1.1 Environnement matériel

Afin de mener à bien nos expérimentations, nous avons utilisé à la fois un ordinateur personnel et un environnement d'exécution virtuel.

	Système d'exploitation	**RAM**	**Processeur**	**Autres**
Machine 1	Windows 10 (64 bits)	4 Go	Intel Celeron N4000 @ 1.10 GHz	/
Environnement virtuel (Google Colab)	Ubuntu 18.04 (Linux)	13 Go	Intel Xeon @ 2.20 GHz	GPU Tesla T4 : 15 Go TPU : 180 téraflops

TABLE 4.1 – Caractéristiques des environnements d'exécution

4.1.1.1 Langage de programmation et logiciel

Nous avons choisi les outils suivants :



Pour l'implémentation de nos algorithmes, nous avons choisi le langage de programmation **Python**, version **3.12.1**, en raison de sa simplicité, de sa riche documentation et de ses nombreuses bibliothèques facilitant la manipulation de graphes et le développement d'algorithmes d'optimisation.



Pour la conception de l'interface graphique, nous avons utilisé **Qt Designer**, un outil visuel inclus dans le framework **PyQt6.** Il permet de créer rapidement des interfaces utilisateur en glissant-déposant les composants, sans écrire manuellement le code.

4.1.2 Librairies et bibliothèques

Le tableau suivant présente les bibliothèques utilisées pour nos algorithmes :

TABLE 4.2 – Bibliothèques Python utilisées et leurs descriptions

Bibliothèque	Description
NumPy	Bibliothèque open source de Python destinée à la manipulation de tableaux multidimensionnels et aux fonctions mathématiques appliquées à ces tableaux.
NetworkX	Librairie Python permettant de créer, manipuler et analyser des graphes (nœuds, arêtes, algorithmes de coloration, etc.).
itertools.combinations	Sert à générer toutes les paires possibles d'éléments, utile notamment pour créer les arêtes d'un graphe.
Pandas	Bibliothèque open source de Python facilitant l'analyse et la manipulation de données grâce à des structures comme les DataFrames.
Sys	Fournit un accès aux fonctions système (ex. : gestion des arguments de ligne de commande, arrêt du programme).
Tabulate	Permet de formater des données sous forme de tableau pour un affichage clair dans la console.

xlrd, xlwt	Bibliothèques utilisées pour lire (`xlrd`) et écrire (`xlwt`) dans des fichiers Excel au format `.xls`.
DEAP (base, creator, tools, algorithms)	Fournit les composants nécessaires à la création et à l'exécution d'algorithmes génétiques (sélection, croisement, mutation, etc.).
Matplotlib	Bibliothèque open source utilisée pour tracer des graphiques et visualiser des données de manière flexible.
PyQt6	Module Python permettant de créer des interfaces graphiques en liant Python à la bibliothèque Qt.

Section 4.2

Description des données

Comme mentionné dans le chapitre précédent, le jeu de données utilisé dans nos trois algorithmes comporte les colonnes suivantes : Module, Type, Section, Groupe, ID et Enseignant. Ce dataset englobe l'ensemble des niveaux du département de mathématiques et informatique, et comprend un total de 277 enregistrements.

La carte de répartition (treemap) ci-dessus présente la manière dont les cours sont répartis entre les 15 sections :

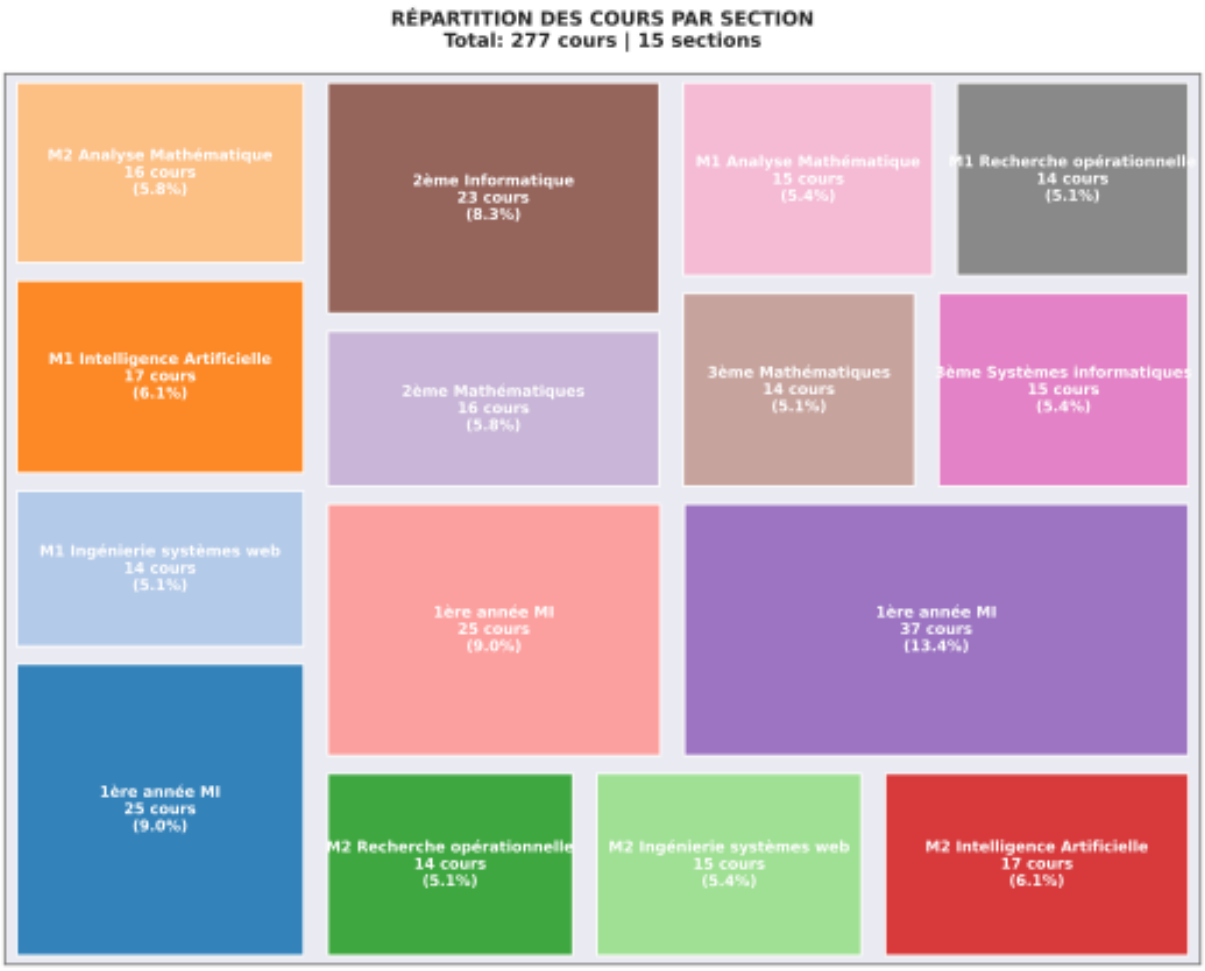


FIGURE 4.1 – Carte de répartition du Dataset

Section 4.3

Evaluation des algoritmes

L'évaluation des algorithmes a été réalisée en analysant leur aptitude à générer un emploi du temps sans conflits tout en respectant l'ensemble des contraintes imposées. Le choix des paramètres utilisés a été guidé par une approche expérimentale, basée sur des essais successifs visant à optimiser les performances. Certains paramètres ont été maintenus constants à travers les trois algorithmes testés, tandis que d'autres ont été ajustés selon la méthode employée. Les paramètres communs et invariants sont définis comme suit :

⇒ **NB_CRENEAUX = 30** : le nombre total de créneaux disponibles dans la semaine (5 créneaux par jour × 6 jours effectifs + 1 jour optionnel). Chaque créneau correspond à une séance de 1h30.

⇒ **NB_SALLES = {'course' : 7, 'dw' : 12, 'pw' : 8}** : dictionnaire indiquant le nombre de salles disponibles par type de séance. Il permet de contrôler la distribution des ressources selon le type de session.

4.3.1 Evaluation de l'algorithme de force brute

4.3.1.1 Ajustement de paramètre

Une série de tests empiriques a été conduite en faisant varier les valeurs du paramètre MAX_TENTATIVES, considéré comme l'un des éléments clés de l'algorithme de force brute. L'objectif était d'évaluer son influence sur les performances de l'algorithme et d'en identifier la valeur optimale. Ce paramètre définit le nombre maximal de tentatives autorisées pour affecter une séance à un créneau horaire et à une salle compatible avant qu'elle ne soit considérée comme non placée. Il joue ainsi un rôle central dans le compromis entre temps de calcul et taux de placement des séances.

Les résultats obtenus sont les suivants :

- Lorsque `MAX_TENTATIVES = 5`, le nombre de cours non placés augmente considérablement, atteignant **61 cours non affectés** sur un total de 277, ce qui traduit une efficacité insuffisante de l'algorithme dans ces conditions.
- En fixant `MAX_TENTATIVES = 10`, le taux de placement s'améliore sensiblement pour atteindre environ **79,42%**, bien que le résultat demeure encore perfectible.
- À partir de `MAX_TENTATIVES = 15`, aucune amélioration notable n'est observée, ce qui suggère que cette valeur constitue un **seuil optimal** pour notre configuration expérimentale.

4.3.1.2 Analyse de la complexité de l'algorithme

Notre algorithme de génération d'emploi du temps repose sur une approche itérative pour placer les cours dans des créneaux horaires tout en évitant les conflits. Sa complexité peut être décomposée comme suit :

1. **Tri initial des cours (`trier_cours`)**
   - La fonction `trier_cours` utilise un tri basé sur le nombre de cours par enseignant et le type de cours.
   - **Complexité** : $\mathcal{O}(N \log N)$, où N est le nombre de cours, en raison de l'opération de tri.
2. **Boucle principale sur tous les cours**
   - Pour chaque cours, l'algorithme effectue jusqu'à MAX_TENTATIVES tentatives de placement.
   - **Complexité par cours** :
     - Choix aléatoire d'un créneau : $\mathcal{O}(1)$.
     - Vérification des conflits avec les cours déjà placés : $\mathcal{O}(M)$, où $M \leq N$ (dans le pire cas, $M \approx N$).

- Boucle sur les salles disponibles : $\mathcal{O}(S)$, où S est le nombre maximal de salles pour un type de cours.

3. **Complexité globale**
   - Dans le pire cas, chaque cours est comparé à tous les autres cours déjà placés pour chaque salle.
   - La complexité totale est donc : $\mathcal{O}(N \log N) + \mathcal{O}(N \times MAX_TENTATIVES \times S \times M)$.
   - Pour M peut atteindre N dans le pire cas, et en considérant MAX_TENTATIVES et S comme des constantes, et la complexité se simplifie en : $\mathcal{O}(N^2)$.

D'un point de vue computationnel, l'algorithme présente des performances très satisfaisantes. Le temps d'exécution moyen est de $0,24$ secondes, ce qui le rend rapide et réactif. Du côté de la mémoire, la consommation reste faible, avec une moyenne de $935,57 Ko$, traduisant une bonne stabilité et une légèreté appréciable, même sur des machines peu puissantes.

4.3.1.3 Évaluation des résultats : taux de placement, distribution des créneaux et cohérence des jours libres

En ce qui concerne le taux de placement, l'algorithme atteint une performance optimale de 100% lorsque la taille du jeu de données est inférieure à 150 cours. Cependant, au-delà de ce seuil, les performances se dégradent progressivement : le taux de placement chute à $79,78\%$, ce qui met en évidence la sensibilité de l'approche actuelle à la complexité et au volume des données. Ainsi, l'algorithme se révèle efficace pour des ensembles de données de taille modérée, mais rencontre des limites lorsqu'il est confronté à une charge plus importante.

La cohérence des journées libres est également assurée pour l'ensemble des sections. Cette contrainte garantit qu'une même journée sans cours est attribuée à tous les groupes d'une section donnée. Chaque section dispose donc d'au moins une journée hebdomadaire sans aucune séance programmée, ce qui favorise une meilleure organisation du temps et contribue à un équilibre dans la charge pédagogique.

Enfin, la distribution des créneaux témoigne d'un équilibre globalement satisfaisant. Bien que l'algorithme ne mobilise pas de manière parfaitement uniforme l'ensemble des créneaux horaires, les écarts demeurent relativement faibles.

Cela se reflète visuellement à travers l'histogramme, qui illustre cette répartition :

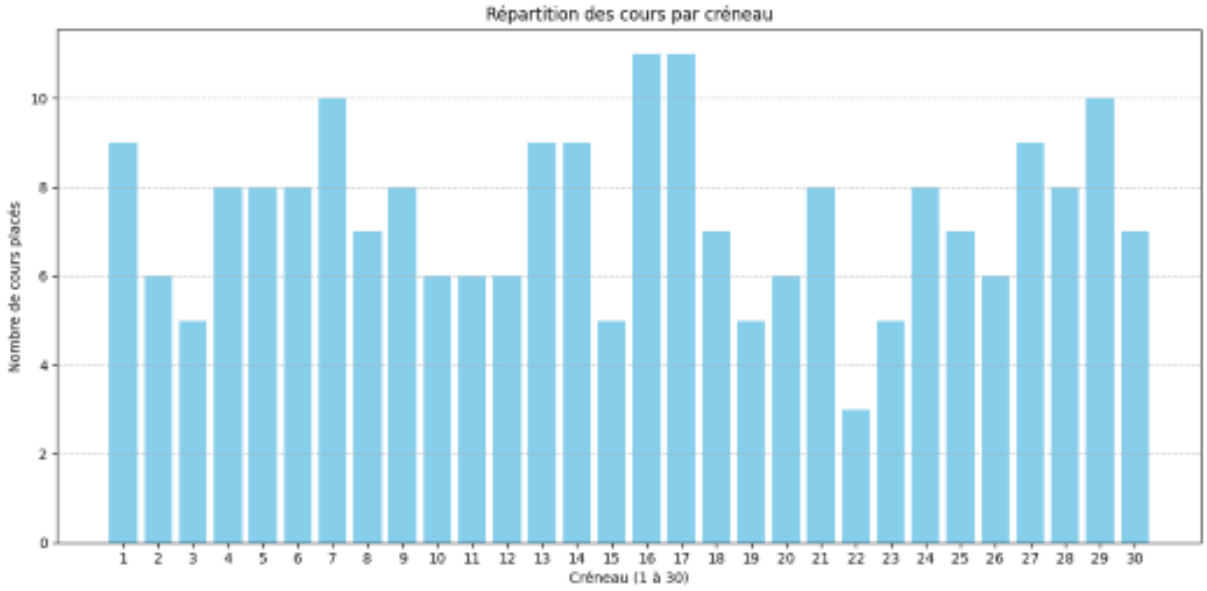


FIGURE 4.2 – Histogramme de répartition des cours par créneau de l'algorithme force brute

4.3.2 Evaluation de l'algorithme de coloration de graphe avec heuristiques

4.3.2.1 Le graphe

En exploitant la bibliothèque NetworkX de Python, nous avons généré le graphe de conflits représentant l'ensemble du dataset avant l'étape de coloration. Chaque nœud de ce graphe correspond à un cours et intègre l'ensemble de ses caractéristiques (module, groupe, enseignant, etc.). Le graphe ainsi obtenu comporte 277 nœuds et 19561 arêtes, traduisant la densité élevée des conflits potentiels dans la répartition des cours.

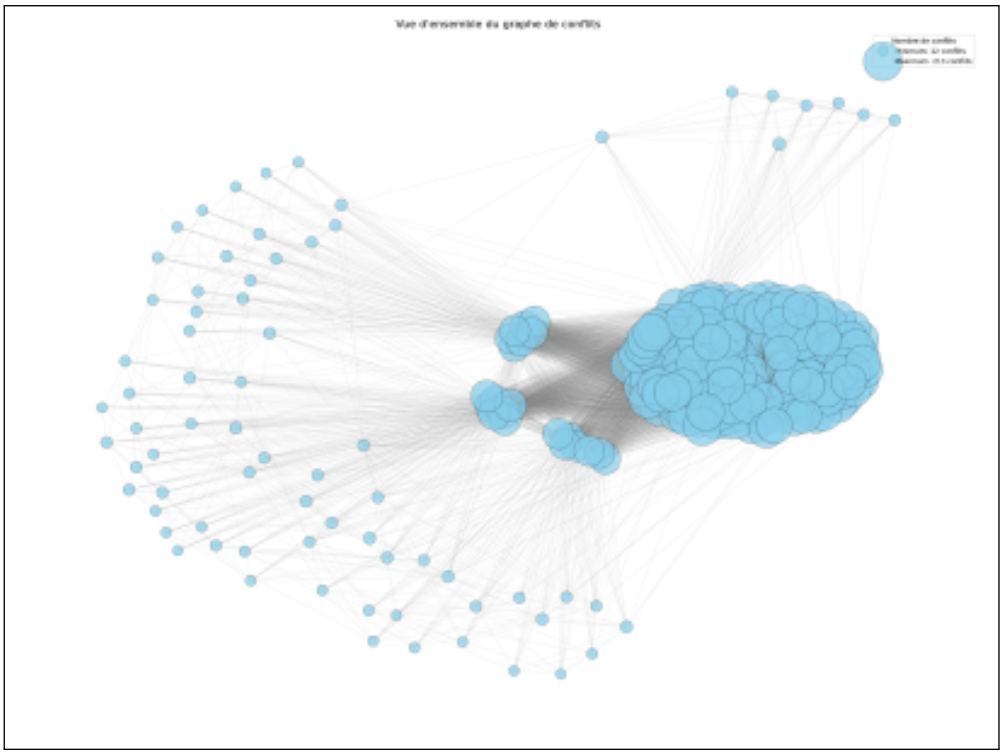


FIGURE 4.3 – Graphe avant coloration généré à partir du Dataset

À titre d'exemple, nous présentons une illustration détaillée du contenu d'un nœud ainsi que de ses voisins, afin de mieux visualiser les relations de conflit identifiées dans le graphe.

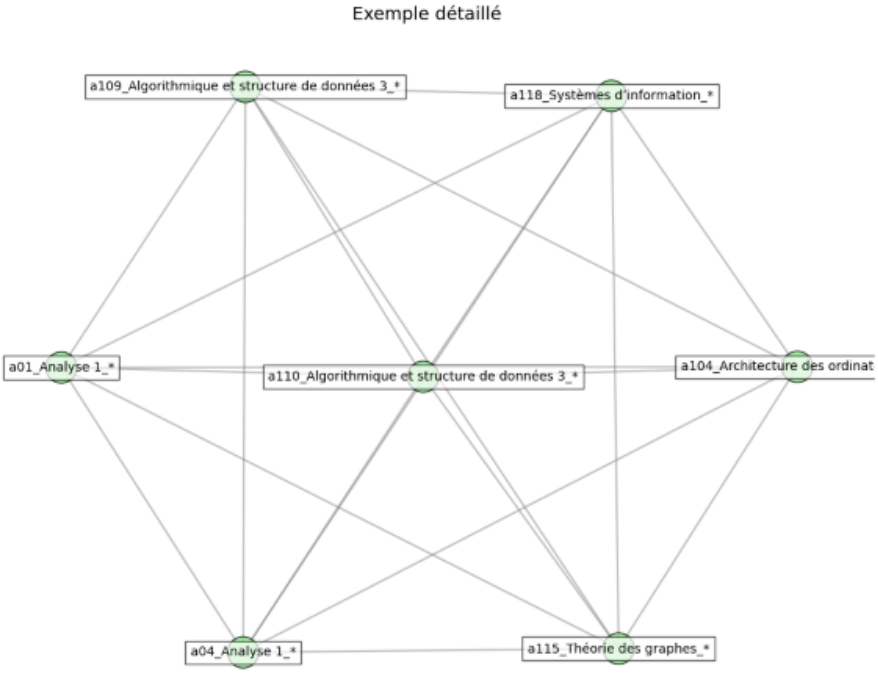


FIGURE 4.4 – Exemple détaillé d'un noeud du graphe

Bien que ce graphe ne soit pas massif au sens des très grands réseaux complexes, il reste

modérément grand et particulièrement dense. Le nombre élevé d'arêtes indique une forte connectivité, traduisant un grand nombre de contraintes à satisfaire pour éviter les conflits. Cette complexité rend l'utilisation d'algorithmes efficaces et optimisés pour sa coloration indispensable.

Après application de notre algorithme de coloration, le graphe présente une distribution de couleurs où chaque couleur correspond à un créneau horaire spécifique. Cette visualisation permet d'identifier les cours pouvant être programmés simultanément (nœuds de même couleur, sans conflit) et ceux devant être séparés dans le temps (nœuds adjacents de couleurs différentes).

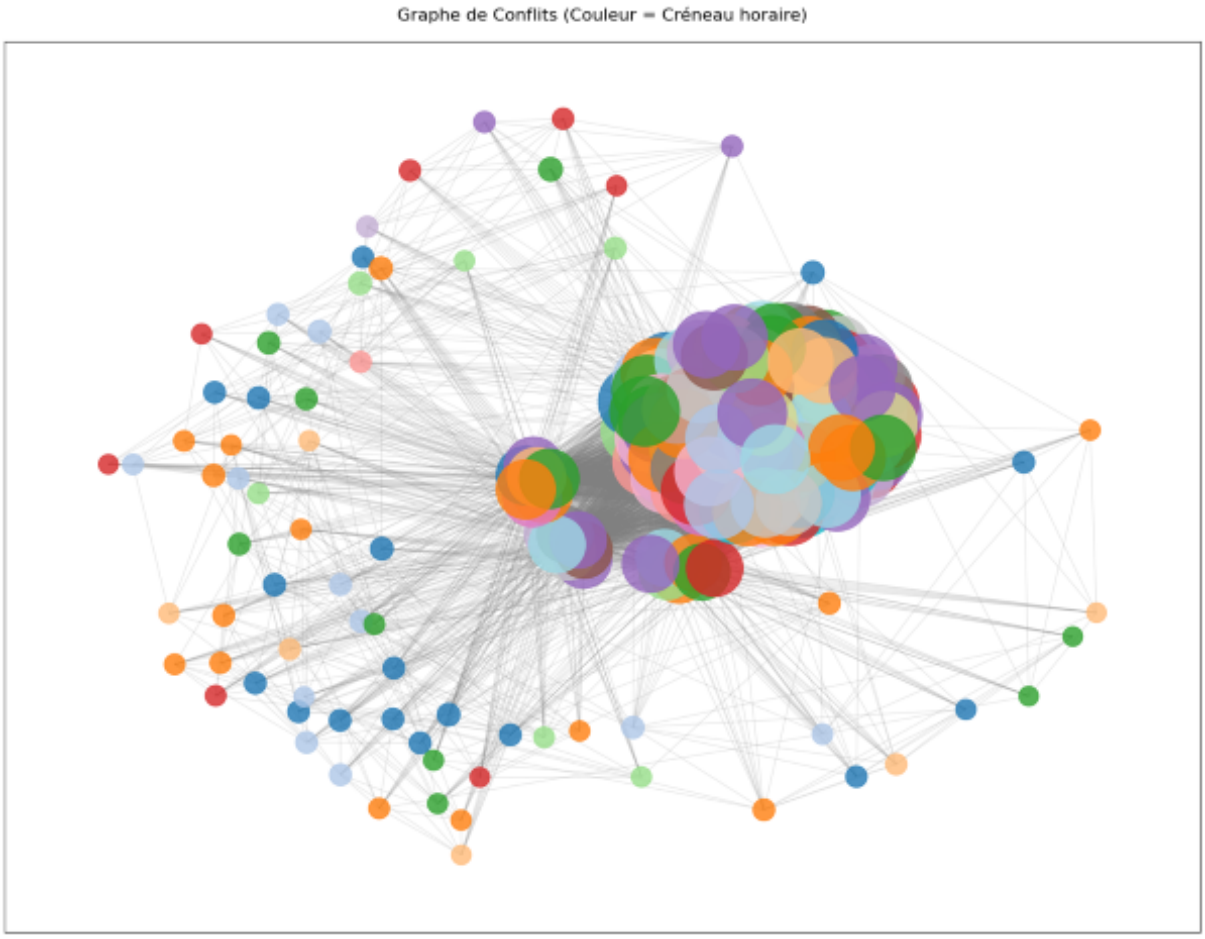


FIGURE 4.5 – Graphe après coloration

4.3.2.2 Analyse de la complexité de l'algorithme

L'algorithme de coloration implémenté repose sur une approche gloutonne optimisée : les nœuds sont triés par degré décroissant, puis chaque nœud se voit attribuer la première couleur disponible non utilisée par ses voisins.

La complexité théorique se décompose comme suit :

- Le tri initial des nœuds selon leur degré est en $\mathcal{O}(n \log n)$, avec n le nombre de nœuds.

- L'attribution des couleurs parcourt ensuite pour chaque nœud ses voisins, ce qui fait $O(m)$ au total, avec m le nombre d'arêtes.

La complexité totale est donc : $O(n \log n + m)$.

Pour valider ces estimations, nous avons mesuré les performances réelles sur notre graphe (277 nœuds, 19561 arêtes). Les résultats obtenus sont les suivants :

- Temps d'exécution : 0.4549 secondes
- Mémoire utilisée : 124.12 Ko

Ces mesures confirment que l'algorithme est efficace, même pour un graphe de taille modérément grande. Le temps d'exécution est inférieur à une demi-seconde, et la mémoire consommée reste faible, rendant l'approche bien adaptée pour la génération d'emplois du temps en contexte universitaire réel.

4.3.2.3 Évaluation des résultats : taux de placement, distribution des créneaux et cohérence des jours libres

L'algorithme de génération d'emploi du temps fondé sur la coloration de graphe donne des résultats satisfaisants en termes de performance et d'équilibre.

Premièrement, le taux de placement atteint 100%, avec un taux de conflits résolus égal à 96,65%. Cela signifie que toutes les séances ont pu être insérées dans l'emploi du temps sans conflit. Chaque nœud du graphe, représentant une séance (cours, TD ou TP), a reçu un créneau horaire respectant les contraintes d'incompatibilité (voisinage).

Deuxièmement, l'analyse de la répartition des couleurs montre une distribution équilibrée des créneaux. Les séances sont globalement bien réparties entre les différents créneaux disponibles, sans surcharge ni sous-exploitation. Ce résultat est favorisé par le tri initial des nœuds par degré décroissant, qui traite en priorité les séances les plus contraignantes, évitant ainsi un regroupement excessif des conflits.

Troisièmement, la cohérence des jours libres est vérifiée pour toutes les sections. Cette contrainte garantit que chaque section bénéficie d'au moins une journée sans aucune séance programmée, et que cette journée libre est commune à tous les groupes d'une même section. Ceci facilite une meilleure organisation et un équilibre dans la charge pédagogique hebdomadaire.

La répartition des créneaux obtenue illustre bien cette organisation efficace et équilibrée :

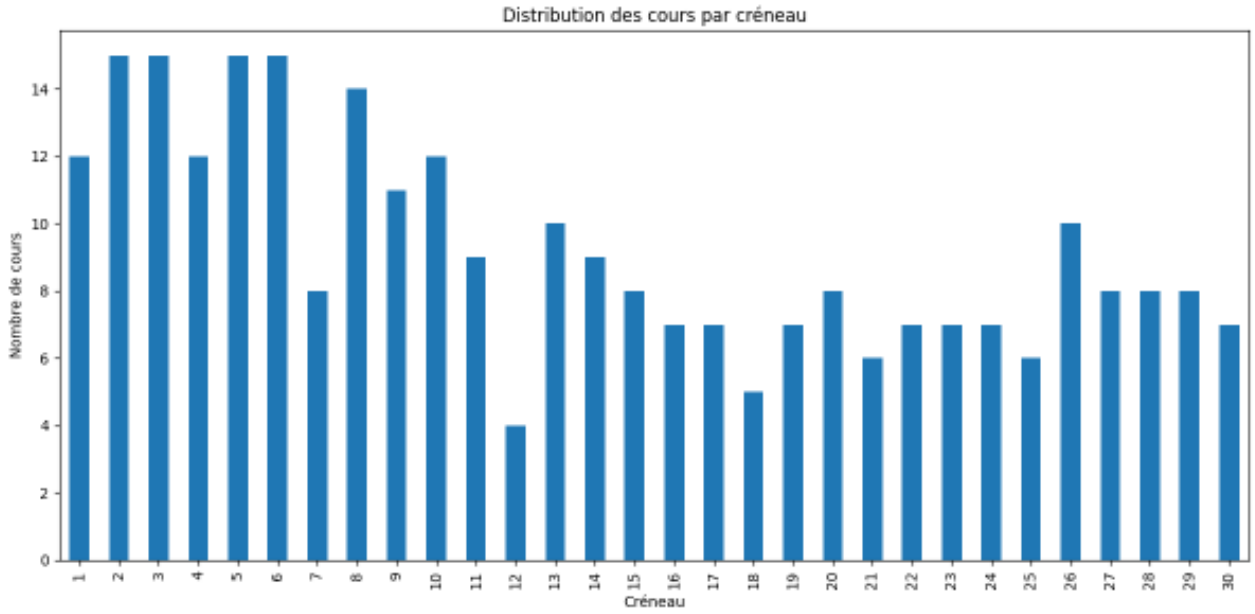


FIGURE 4.6 – Histogramme de répartition des cours par créneau de l'algorithme de coloration avec heuristiques

4.3.3 Evaluation de l'approche de coloration de graphe avec l'algorithme génétique

4.3.3.1 Le graphe

Le graphe utilisé dans cette approche est identique à celui de la deuxième méthode. Il s'agit d'un graphe comportant 277 nœuds et 19561 arêtes, construit à partir de notre jeu de données.

4.3.3.2 Ajustement des paramètres

Divers paramètres influencent directement les performances de cet algorithme. Dans cette perspective, le choix de leurs valeurs a été guidé par une série d'essais expérimentaux visant à optimiser les résultats. Nous justifions dans le tableau ci-dessous les choix retenus :

Paramètre	Valeur	Justification principale
POPULATION_SIZE	300	Pour 277 nœuds, 300 individus permettent une bonne couverture de l'espace de recherche tout en restant gérable en mémoire.
GENERATIONS	150	Permet une exploration approfondie tout en limitant le temps d'exécution.
CX_PROB	0.8	Pour favoriser les croisements et accélérer l'exploration de l'espace de recherche. Nous avons retenu une valeur élevée, conformément aux recommandations classiques situées entre 0,6 et 0,9 pour les problèmes complexes.
MUT_PROB	0.15	Un taux légèrement réduit (par rapport au standard de 0,2) car : la mutation intelligente cible déjà les nœuds en conflit. Et avec un graphe de 277 nœuds, un taux de mutation de 0,15 correspond à environ 41 mutations par individu en moyenne, ce qui s'avère suffisant pour maintenir une bonne diversité génétique au sein de la population.
TOURNSIZE	3	NOUS imposons une sélection plus stricte favorisant les individus les plus performants. Toutefois, lors des tests préliminaires, un paramètre `TOURNSIZE = 5` a conduit à une perte prématurée de diversité génétique.

TABLE 4.3 – Paramètres utilisés dans l'algorithme génétique

4.3.3.3 Analyse des performances de l'évolution

Le temps d'exécution de l'algorithme a été d'environ 3 minutes et 0,17 seconde, ce qui peut être considéré comme relativement long, principalement en raison du nombre élevé de générations fixées à 150. Cependant, l'analyse de la courbe d'évolution des critères d'évaluation montre une progression satisfaisante vers les objectifs de minimisation définis.

La figure suivante illustre cette convergence :

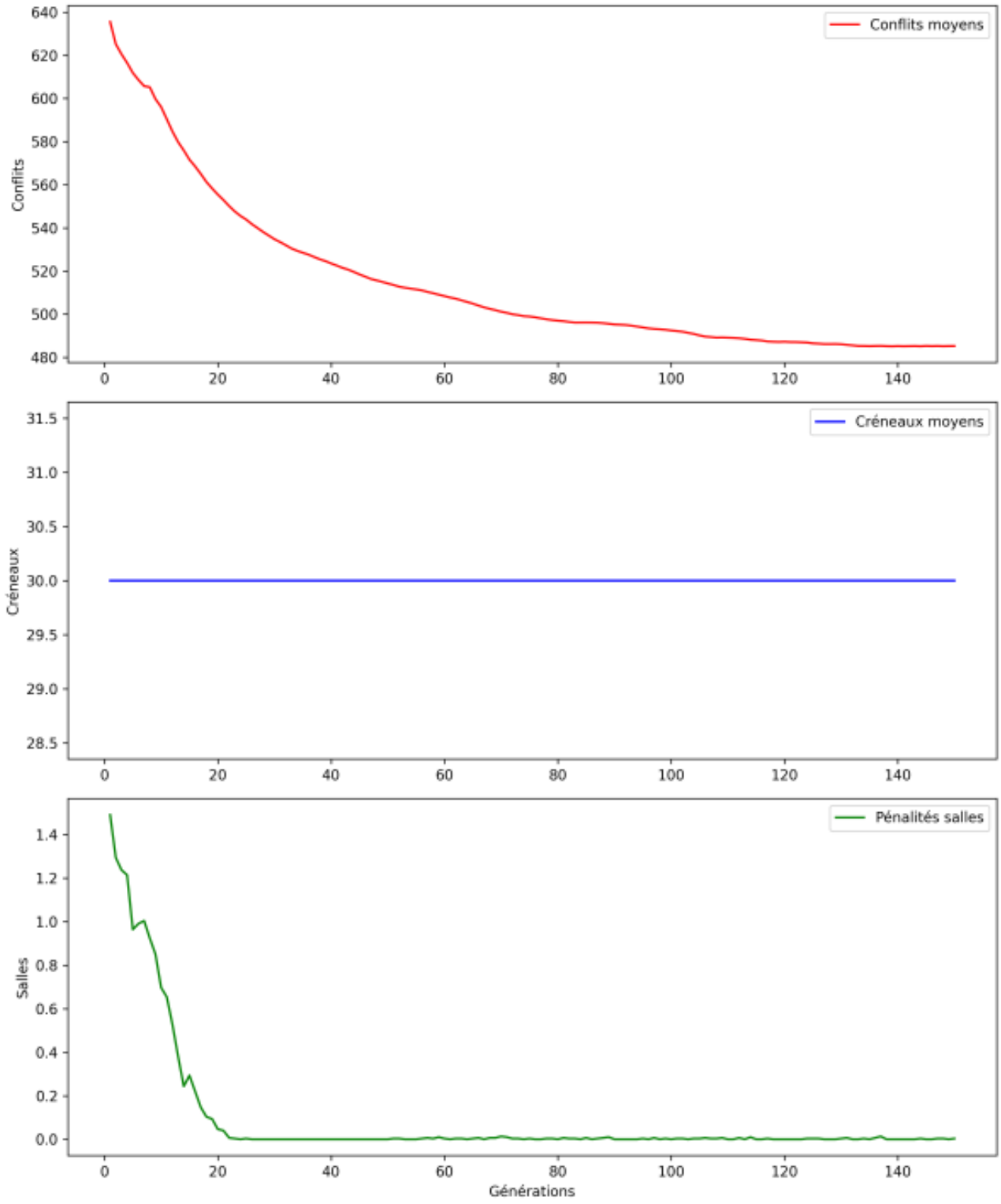


FIGURE 4.7 – Courbes d'évolution de l'algorithme génétique

Concernant la courbe de convergence de la fitness, elle illustre l'évolution des performances de l'algorithme génétique au fil des générations. Nous observons une diminution progressive des valeurs de fitness, tant minimales que moyennes, ce qui traduit une amélioration continue de la qualité des solutions générées. Durant les premières générations, la

baisse est significative, indiquant une phase d'exploration efficace de l'espace des solutions. Par la suite, la diminution devient plus lente, marquant une phase de convergence vers des solutions optimisées. À partir d'environ la 80ème génération, les courbes se stabilisent, suggérant que l'algorithme atteint un équilibre et que la population devient homogène en termes de qualité. Cette stabilité indique une convergence vers un optimum (local ou global), ce qui confirme la capacité de l'algorithme à trouver des solutions satisfaisantes au problème de génération d'emploi du temps.

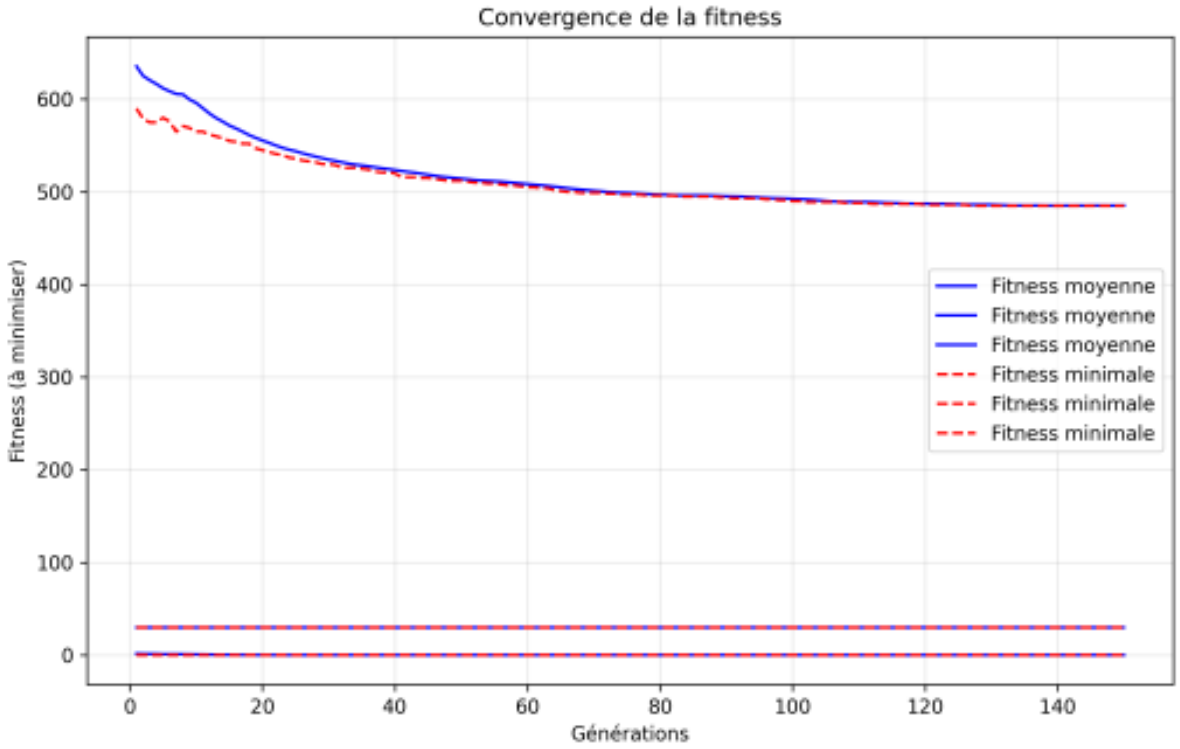


FIGURE 4.8 – Courbe de convergence de la fitness

4.3.3.4 Évaluation des résultats : taux de placement, distribution des créneaux et cohérence des jours libres

L'algorithme de génération d'emploi du temps basé sur la coloration de graphe combinée à un algorithme génétique s'est révélé performant et équilibré à plusieurs niveaux.

Tout d'abord, le taux de placement atteint 100%, ce qui signifie que toutes les séances ont pu être insérées dans l'emploi du temps, sans qu'aucune ne soit laissée de côté. Sur les 19561 conflits potentiels identifiés, 97,5% ont été résolus (soit 19074 conflits éliminés), un taux légèrement supérieur à celui obtenu avec la première méthode de coloration assistée par heuristique.

Ensuite, nous observons une bonne répartition des séances sur l'ensemble des créneaux horaires disponibles. Il n'y a ni surcharge (accumulation excessive de séances sur certains

créneaux), ni sous-exploitation (créneaux laissés vides sans nécessité), ce qui montre que l'espace de recherche a été exploré de manière efficace par l'algorithme.

Par ailleurs, la contrainte liée à la cohérence des jours libres pour chaque section est respectée. Tous les groupes d'une même section partagent le même jour libre, conformément aux contraintes définies. On constate d'ailleurs que le jeudi est majoritairement attribué comme jour libre, ce qui reflète une organisation logique et centralisée de l'emploi du temps.

Enfin, la répartition des créneaux obtenue met en évidence une organisation à la fois efficace et équilibrée, démontrant que l'approche par algorithme génétique, appliquée à un graphe de 277 nœuds et plus de 19000 arêtes, est capable de produire des solutions de haute qualité dans un problème complexe de génération d'emploi du temps.

La répartition des créneaux obtenue illustre bien cette organisation efficace et équilibrée :

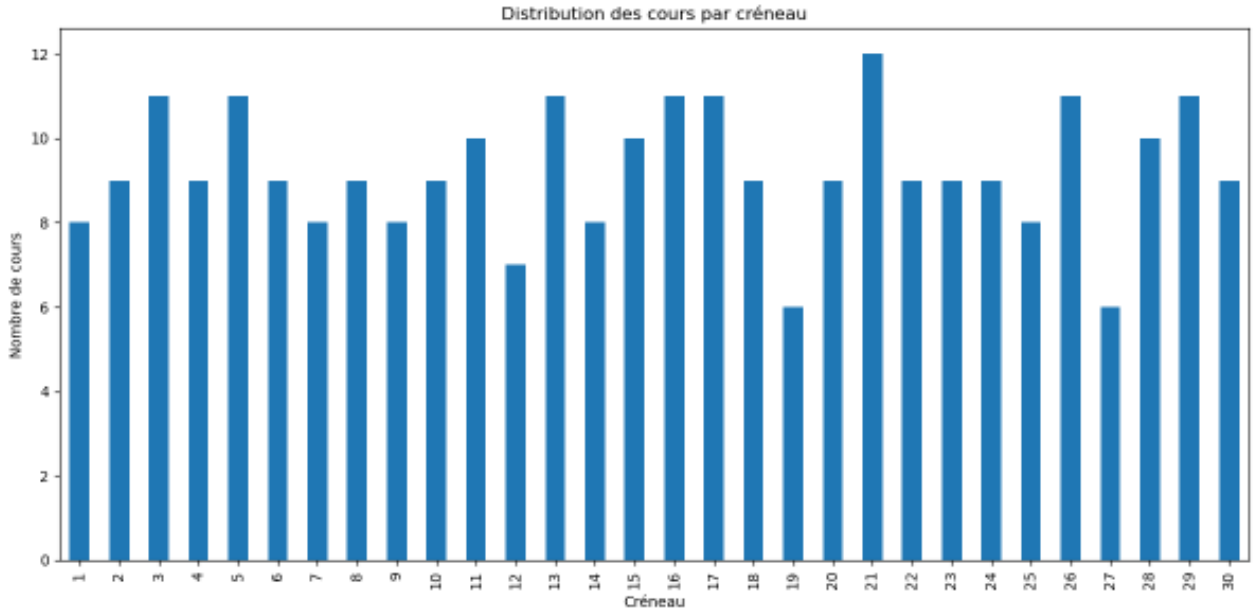


FIGURE 4.9 – Histogramme de répartition des cours par créneau avec l'algorithme génétique

Section 4.4

Comparaison entre les trois approches

Le tableau ci-dessous met en évidence les différences entre les approches proposées, en les appliquant à différents ensembles de données et en comparant plusieurs indicateurs de performance.

Approche	Taille des données traitées	Taux de placement (%)	Taux de conflits résolus (%)	Écart-type d'utilisation des créneaux	Temps d'exécution (sur colab) (secondes)
Force bruteavec heuristique	120 (120 nœuds / 1174 arêtes)	**100**	/	**8.29**	**0.00**
	277 (277 nœuds / 19561 arêtes)	79.78		8.78	0.05
	368 (368 nœuds / 20016 arêtes)	85.60		8.74	0.13
Coloration de graphe avec heuristique	120	**100**	**100**	**1.68**	**0.0035**
	277	**100**	**96.65**	2.66	0.0098
	368	**94.84**	**97.07**	2.19	0.0122
Coloration de graphe avec algorithme génétique	120	**100**	**100**	4.90	**8.65**
	277	**100**	**97.50**	**1.30**	175.15
	368	**100**	**97.88**	2.42	191.67

TABLE 4.4 – tableau comparatif entre les trois approches

Les résultats obtenus montrent que l'approche basée sur la coloration combinée à un algorithme génétique offre les performances globales les plus élevées. Elle parvient à maintenir un taux de placement de 100%, quel que soit le volume de données traité, y compris pour les ensembles les plus volumineux (jusqu'à 368 entrées), tout en assurant un taux de résolution des conflits supérieur à 97%. Toutefois, cette efficacité s'accompagne d'un temps d'exécution élevé, en raison du nombre important de générations nécessaires dans l'algorithme génétique.

En deuxième position vient l'approche par coloration de graphe avec heuristique, qui se distingue par sa rapidité d'exécution et sa capacité à produire des solutions de qualité, notamment sur des jeux de données de taille moyenne.

Section 4.5

Discussion générale des résultats obtenus

Notre travail visait principalement à concevoir des algorithmes performants pour la génération automatique d'emplois du temps universitaires, en respectant un ensemble complexe de contraintes. Trois approches d'optimisation ont été développées : une méthode de force brute avec heuristique, une approche par coloration de graphe accompagnée d'une heuristique, et enfin une méthode hybride combinant la théorie des graphes avec un algorithme génétique. Les tests réalisés sur ces algorithmes ont permis de dégager les constats suivants :

La méthode de Force brute avec heuristique garantit un taux de placement de 100% uniquement sur les petites tailles de données (120 entrées). Cependant, à mesure que la taille augmente (277 et 368), le taux de placement chute respectivement à 79.78% et 85.60%, ce qui montre une difficulté de l'approche à maintenir ses performances sur des instances plus complexes. L'écart-type de l'utilisation des créneaux reste élevé (environ 8.7), traduisant une mauvaise répartition des séances dans le temps. En termes de temps d'exécution, cette approche est extrêmement rapide (inférieure à 0.2 seconde), ce qui en fait une solution envisageable pour des jeux de données simples, mais insuffisante pour des scénarios complexes.

L'approche de coloration de graphe avec heuristique apporte une nette amélioration. Elle atteint un taux de placement de 100% jusqu'à 277 entrées, avec un taux de conflits résolus avoisinant les 97%, traduisant une très bonne capacité à générer des solutions sans conflits majeurs. L'écart-type, quant à lui, est significativement réduit (entre 1,68 et 2,66), témoignant d'une meilleure répartition des créneaux. Le temps d'exécution est ici remarquablement bas (entre 0.0035 et 0.0122 secondes), ce qui illustre l'efficacité de cette méthode pour des graphes denses, tout en garantissant un excellent compromis entre qualité et rapidité.

L'approche hybride, combinant coloration de graphe et algorithme génétique, présente les meilleurs résultats globaux. Elle conserve un taux de placement de 100% quelle que soit la taille du jeu de données, même pour les 368 entrées. Le taux de résolution des conflits dépasse les 97%, illustrant la robustesse de l'algorithme. L'écart-type modéré (de 1,30 à 4,90) reflète un bon équilibre dans l'utilisation des créneaux, tout en respectant les contraintes imposées. Cependant, cette robustesse a un **coût computationnel élevé**, avec des temps d'exécution allant de 8 à 192 secondes.

L'évolution des performances à travers ces approches souligne l'intérêt d'introduire des techniques d'optimisation évolutives. Alors que la méthode naïve atteint ses limites sur des volumes de données importants, les approches basées sur la coloration de graphe, et plus encore leur couplage avec des algorithmes génétiques, offrent des solutions évolutives,

équilibrées et efficaces pour la génération d'emplois du temps complexes.

Section 4.6

Présentation de l'interface

Nous avons développé une interface homme-machine de type bureau pour chaque approche, permettant à l'utilisateur d'exploiter les résultats de nos algorithmes et de les visualiser de manière claire et structurée. Chaque interface comporte cinq onglets : *Accueil, Configuration des modules, Emploi du temps, Modification et Visualisation PDF*. Nous décrivons ci-dessous le rôle et les fonctionnalités de chacun de ces onglets :

4.6.1 Onglet Accueil

Cet onglet présente une vue d'ensemble de l'interface. Il fournit une description détaillée du fonctionnement du système, accompagnée d'un guide d'utilisation complet expliquant les étapes à suivre dans les quatre autres onglets. La figure suivante illustre la phase d'entrée de notre système :



FIGURE 4.10 – Onglet "Acceuil" de l'interface

4.6.2 Onglet configuration des modules

Il permet à l'administrateur de sélectionner un niveau d'études parmi ceux enseignés à la faculté. Une fois le niveau choisi, les modules correspondants s'affichent automatiquement.

L'administrateur peut alors configurer le nombre de sections et de groupes pour ce niveau, puis attribuer les enseignants aux cours (par section), aux travaux dirigés (TD) et aux travaux pratiques (TP) (par groupe). La liste des enseignants disponibles est définie par le système, ce qui garantit une sélection cohérente et conforme à la base de données existante.

Une fois la configuration terminée, il peut enregistrer les données en cliquant sur le bouton « Sauvegarder la configuration ». En cas de besoin, il peut réinitialiser tous les champs à l'aide du bouton « Réinitialiser ».

Les données sont sauvegardées dans un fichier nommé BDD.xls, structuré selon les colonnes suivantes : Module, Type, Section, Groupe, ID et Enseignant. Ce fichier sert ensuite de base pour la génération des emplois du temps à l'aide de nos algorithmes. La figure suivante illustre cet onglet :

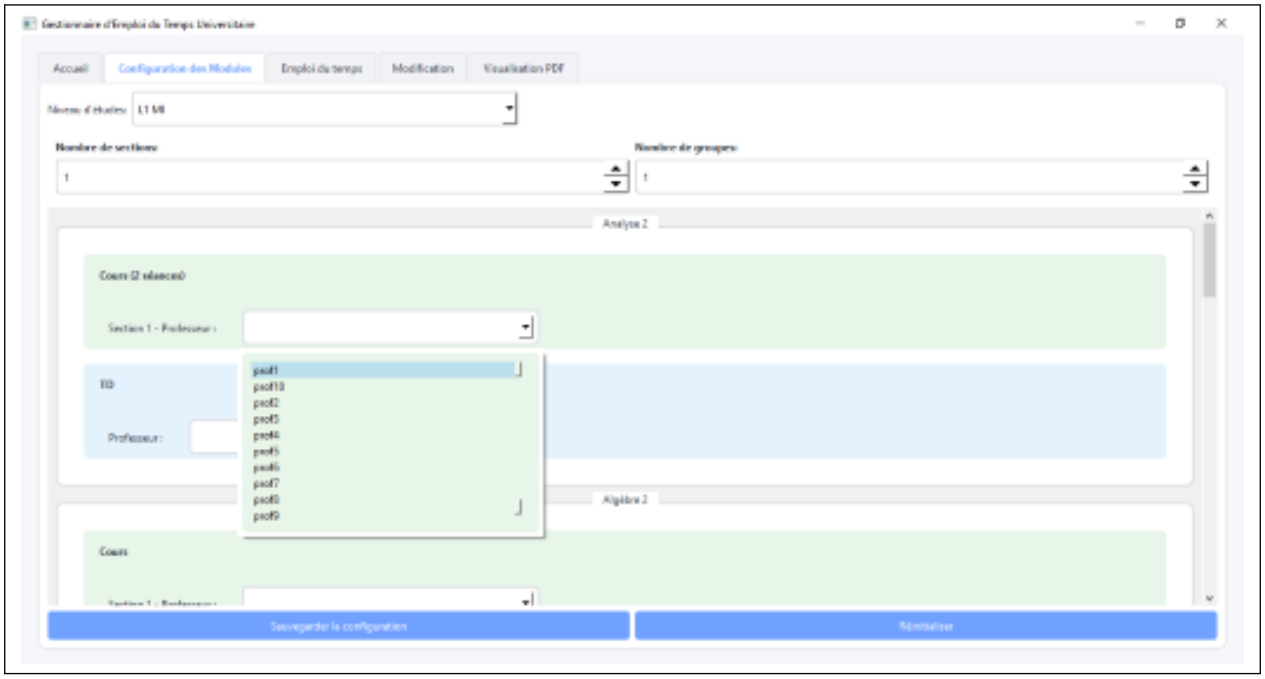


FIGURE 4.11 – Onglet "Configuration des Modules" de l'interface

4.6.3 Onglet Modification

Cet onglet permet à l'administrateur de sélectionner le niveau d'études à modifier. Dès la sélection, les données existantes sont automatiquement chargées depuis le fichier BDD.xls. L'administrateur peut ensuite modifier les informations relatives aux sections, aux groupes ainsi qu'aux noms des enseignants. Les principales fonctionnalités offertes dans cet onglet sont : la suppression de lignes, la modification des affectations et la sauvegarde des changements apportés.

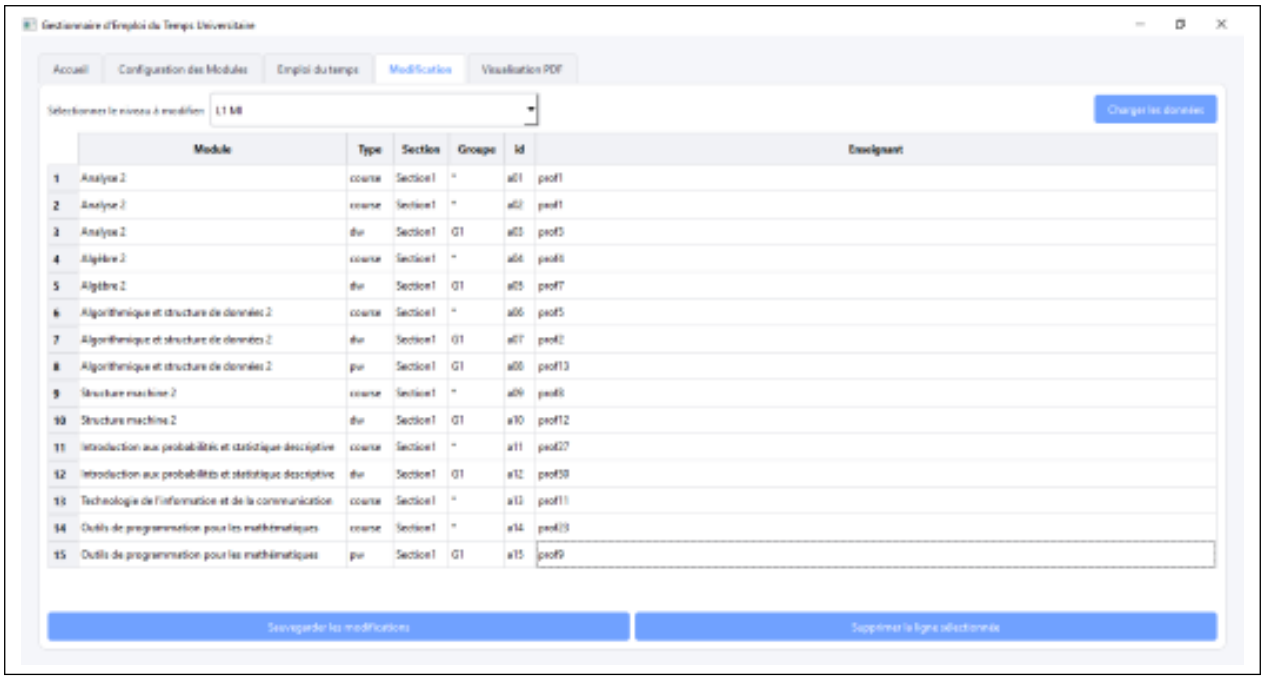


FIGURE 4.12 – Onglet "Modification" de l'interface

4.6.4 Onglet Emploi du temps

Dans cet onglet, l'utilisateur peut générer un emploi du temps en utilisant l'approche associée à l'interface actuelle : force brute, coloration de graphe avec heuristiques, ou algorithme génétique. Après avoir configuré les paramètres nécessaires, il lui suffit de cliquer sur le bouton « Générer l'emploi du temps ». Une fenêtre s'affiche alors pour indiquer l'état d'avancement du processus de génération. Une fois l'opération terminée, les statistiques suivantes sont présentées : le nombre de séances placées, le nombre de séances non placées.

En dessous de ces statistiques, une grille interactive représentant l'emploi du temps généré est affichée. Cette grille couvre l'ensemble des sections, mais l'utilisateur peut appliquer des filtres pour visualiser uniquement les informations relatives à une section spécifique, à un groupe donné, ou à un type de séance (Cours, TD, TP, ou Tous). Nous prenons comme exemple la génération d'emploi avec l'approche de l'algorithme génétique :

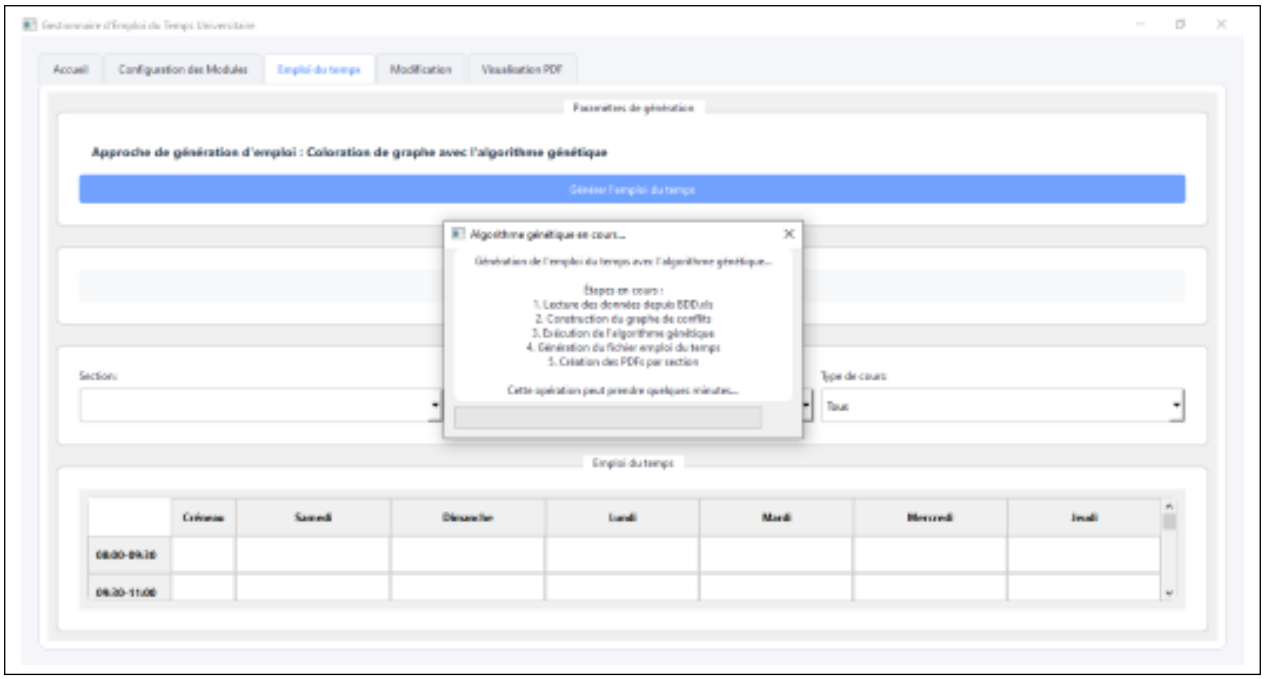


FIGURE 4.13 – Onglet "Emploi du temps" lors de la génération

Voici le résultat :

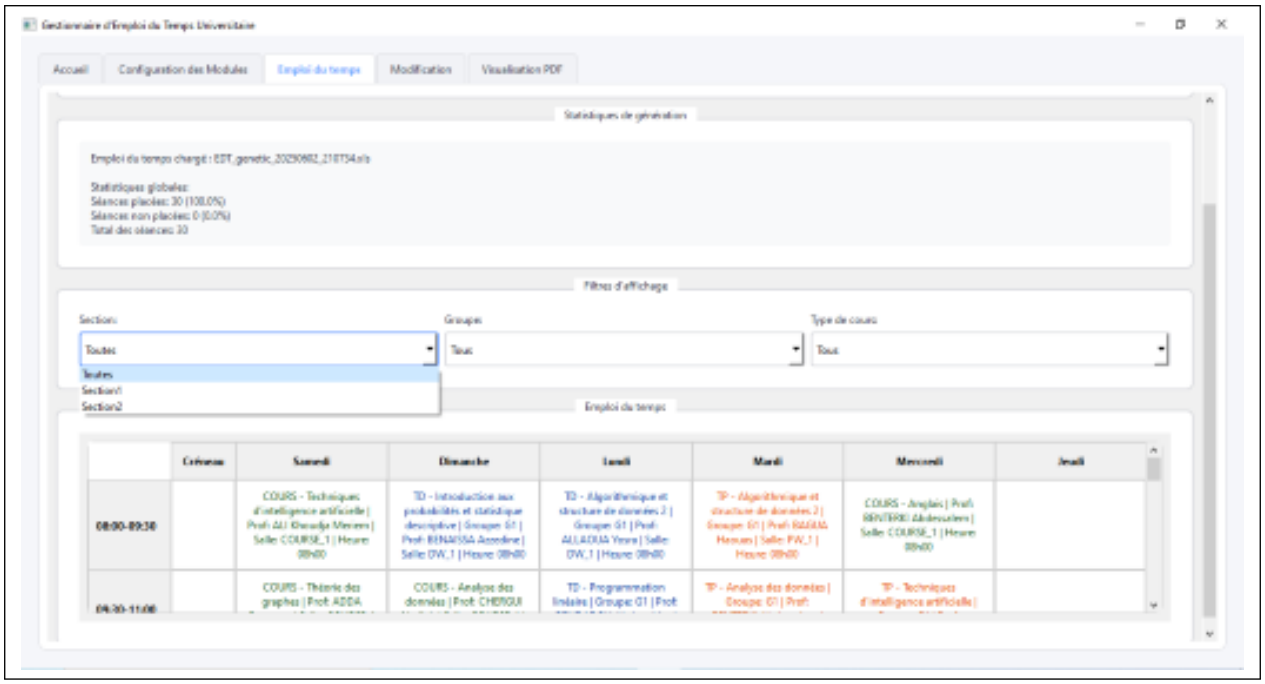


FIGURE 4.14 – Onglet "Emploi du temps" après la génération

4.6.5 Onglet Visualisation PDF

Après la génération des emplois du temps, un fichier PDF est automatiquement créé pour chaque section. Dans cet onglet, l'utilisateur peut sélectionner la section souhaitée afin de prévisualiser son emploi du temps au format PDF, puis le télécharger en cliquant sur le bouton « Télécharger le PDF ». La figure suivante présente un exemple illustratif de cette fonctionnalité :

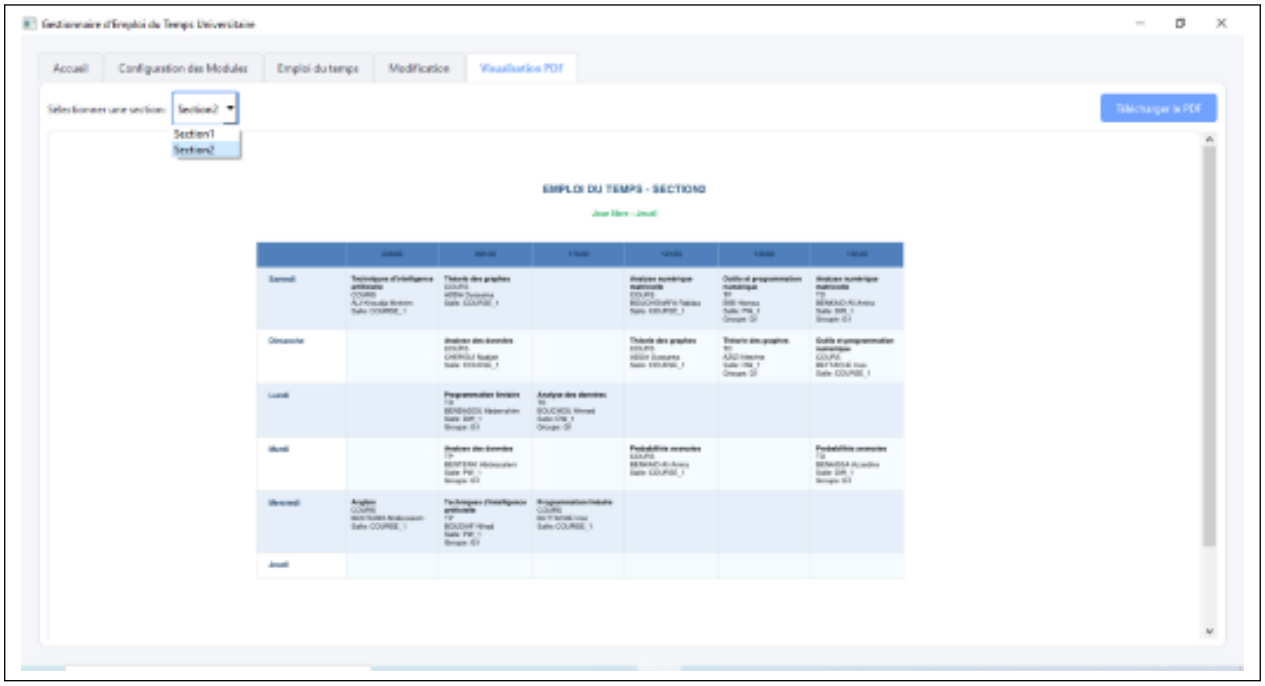


FIGURE 4.15 – Onglet "Visualisation PDF"

Conclusion

Dans ce chapitre, nous avons défini les aspects techniques liés au développement de nos algorithmes de génération d'emploi du temps, en présentant l'environnement ainsi que les outils de développement utilisés. Nous avons ensuite décrit les paramètres adoptés et analysé les résultats obtenus sur différents jeux de données à l'aide de métriques appropriées. Enfin, nous avons présenté l'interface développée, en détaillant ses principales fonctionnalités.

Conclusion et Perspective

La planification des emplois du temps universitaires représente un défi majeur en raison de la multitude de contraintes à prendre en compte. Face à la complexité croissante de cette tâche, notre travail s'est inscrit dans une démarche de recherche opérationnelle enrichie par l'intelligence artificielle, dans le but de proposer des solutions automatiques, efficaces et adaptées aux besoins concrets du terrain universitaire.

Dans cette optique, nous avons conçu et comparé trois approches d'optimisation : une méthode de force brute guidée par heuristique, une stratégie fondée sur la coloration de graphe avec heuristiques, et enfin une méthode de coloration de graphe couplée à un algorithme génétique. L'évaluation expérimentale a permis de mesurer leurs performances sur des jeux de données de tailles variables.

Les résultats montrent que, si la méthode de force brute reste rapide et efficace pour des cas simples, elle peine à gérer des situations complexes. L'approche basée sur la coloration de graphe apporte une nette amélioration, notamment en termes de répartition des créneaux et de réduction des conflits. Toutefois, c'est la méthode de coloration avec algorithme génétique qui s'est révélée la plus performante, réussissant à maintenir un taux de placement optimal tout en respectant les contraintes les plus exigeantes.

Ces constats soulignent l'intérêt de recourir à des techniques d'optimisation avancées, en particulier les méthodes métaheuristiques, dans la résolution de problèmes combinatoires liés à la gestion des ressources universitaires. L'intersection entre la recherche opérationnelle et l'intelligence artificielle, cœur de notre spécialisation, a démontré toute sa pertinence dans ce type de problématique.

Bien que notre travail ait montré des performances prometteuses, il ouvre la voie à plusieurs prolongements. Les algorithmes développés pourraient être adaptés à la planification des examens et des surveillances, en tenant compte de contraintes spécifiques à ces contextes. Par ailleurs, une amélioration des approches existantes reste envisageable, notamment par l'ajout de contraintes dynamiques, ou l'exploration d'autres paradigmes d'optimisation comme les algorithmes de renforcement ou les métaheuristiques hybrides. L'interface actuelle pourrait également être perfectionnée pour offrir un outil plus complet, mieux intégré aux systèmes des établissements. Enfin, l'intégration de techniques

d'apprentissage profond (deep learning) constitue une perspective prometteuse, capable de proposer des solutions prédictives intelligentes à partir de données historiques ou de comportements planifiés.

Bibliographie

[1] A. AHMED. *An introduction to graph analysis and NetworkX*. [Online]. Available, Medium, Jan. 2024. URL : https://medium.com/@ahmedali/an-introduction-to-graph-analysis-and-networkx-2024.

[2] H. AZZOUNE. *Résolution des problèmes*.

[3] Mokhtar S. BAZARAA, John J. JARVIS et Hanif D. SHERALI. *Linear Programming and Network Flows*. 4e éd. Wiley, 2009. ISBN : 9780470462720.

[4] Witold BEDNORZ. *Greedy Algorithms*. Creative Commons CC BY 3.0, eBook, 586 pp. IN-TECH, 2008. ISBN : 9789537619275.

[5] H. BENKAOUHA. *cours de théorie des Graphe L3*. U.S.T.H.B, 2020/2021.

[6] C. BERGE. *Graphes et hypergraphes (2ème éd.)* 1973.

[7] Claude BERGE. *Théorie des graphes et ses applications*. French. Paris : Dunod, 1958. URL : https://www.sudoc.fr/004865871.

[8] M. BLIDIA. *cours de théorie des graphes*. Université Saad Dahlab de Blida.

[9] BONDY et U.S.R. J.A. MURTY. *Graph Theory with Applications*. London : Macmillan, 1976. URL : https://books.google.com/books/about/Graph_Theory_with_Applications.html?id=jCqUswEACAAJ.

[10] A. BOUCHOU. *Graphe et modélisation*. université de Médéa, 2018.

[11] Stephen BOYD et Lieven VANDENBERGHE. *Convex Optimization*. Cambridge University Press, 2004. ISBN : 9780521833783.

[12] R. L. BROUKS. "On colouring the nodes of a network". In : *Mathematical Proceedings of the Cambridge Philosophical Society*. T. 37. Cambridge University Press, 1941, p. 194-197.

[13] Gary CHARTRAND. *Introductory Graph Theory*. See Chapter 2, Section 2.1, Theorem 2.1 : The Degree of a Vertex. New York : Dover Publications, 1977. URL : https://archive.org/details/introductorygrap0000char.

[14] M. CHELLALI. *cours de théorie des graphes*. Université Saad Dahlab de Blida.

[15] George B. DANTZIG. *Linear Programming and Extensions*. Princeton University Press, 1963. ISBN : 9780691030010.

[16] Kalyanmoy DEB. *Multi-Objective Optimization Using Evolutionary Algorithms*. Springer, 2001. ISBN : 9783540656146.

[17] Reinhard DIESTEL. *Graph Theory*. 6th. Berlin : Springer, 2025. URL : https://diestel-graph-theory.com/.

[18] Marco DORIGO et Thomas STÜTZLE. *Ant Colony Optimization*. MIT Press, 2004. ISBN : 9780262132906.

[19] Christian DREO et al., éd. *Metaheuristics for Hard Optimization : Simulated Annealing, Tabu Search, Evolutionary and Genetic Algorithms, Ant Colonies ... Methods and Case Studies*. Natural Computing Series. Springer, 2008.

[20] Zhenan FAN et al. *Artificial Intelligence for Operations Research : Revolutionizing the Operations Research Process*. arXiv preprint arXiv :2401.03244. 2024. URL : https://arxiv.org/abs/2401.03244.

[21] L. FOISSY. *Cours théorie des graphes*. 2010.

[22] Fred GLOVER. "Tabu Search—Part I". In : *ORSA Journal on Computing* 1.3 (1989), p. 190-206.

[23] Michel GONDRAN et Michel MINOUX. *Graphes et algorithmes*. Eyrolles, Paris, 1979.

[24] T. GRABENER. *Calcul d'itinéraire multimodal et multiobjectif en milieu urbain. Doctoral dissertation*. Université des Sciences Sociales-Toulouse I, 2010.

[25] William L. HAMILTON. *Graph Representation Learning*. Morgan & Claypool Publishers, 2020.

[26] Frank HARARY. *Graph Theory*. Addison-Wesley, 1969. URL : https://books.google.com/books/about/Graph_Theory.html?id=FY5BvgAACAAJ.

[27] Peter E. HART, Nils J. NILSSON et Bertram RAPHAEL. "A formal basis for the heuristic determination of minimum cost paths". In : *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), p. 100-107.

[28] John H. HOLLAND. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992. ISBN : 9780262581110.

[29] N. IKHLEF-ESCHOUF. *cours de théorie des graphes*. université de Médéa.

[30] S. KIRKPATRICK, C. D. GELATT et M. P. VECCHI. "Optimization by Simulated Annealing". In : *Science* 220.4598 (1983), p. 671-680.

[31] Amgad MADKOUR et al. "A Survey of Shortest-Path Algorithms". In : *arXiv preprint arXiv :1705.02044* (2017).

[32] W. MEYER. "Equitable coloring". In : *American Mathematical Monthly* 80.8 (1973), p. 920-922.

[33] Jorge NOCEDAL et Stephen J. WRIGHT. *Numerical Optimization*. 2e éd. Springer, 2006. ISBN : 9780387303031.

[34] Panos M. PARDALOS et Mauricio G. C. RESENDE. "Mathematical Optimization and Artificial Intelligence : Challenges and Opportunities". In : *Handbook of Applied Optimization*. Sous la dir. de Panos M. PARDALOS et Mauricio G. C. RESENDE. Oxford University Press, 2002, p. 1-18.

[35] Sylvain PERIFEL. *Complexité algorithmique*. https://www.irif.fr/~sperifel/complexite.pdf. Consulté en juin 2025. 2014.

[36] Saeid SADEGHI et Maghsoud AMIRI. "Artificial Intelligence and Its Application in Optimization". In : *Data Mining : Concepts and Applications*. IntechOpen, 2022, p. 113-130. URL : https://www.intechopen.com/chapters/78912.

[37] Y. SINGH. *Towards use of Dijkstra algorithm for optimal navigation of an unmanned surface vehicle in a real-time marine environment with results from artificial potential field*. 2018.

[38] Christine SOLNON. *Algorithmique avancée pour les graphes*. https://perso.liris.cnrs.fr/christine.solnon/supportAlgoGraphes.pdf. Consulté en juin 2025.

[39] Christine SOLNON. *Théorie des graphes et optimisation dans les graphes*.

[40] El-Ghazali TALBI. *Metaheuristics : From Design to Implementation*. John Wiley & Sons, 2009. ISBN : 9780470278581.

[41] Said TKATEK et al. "Artificial intelligence for improving the optimization of NP-hard problems : a review". In : *International Journal of Advanced Trends in Computer Science and Applications* 9.5 (2020). URL : https://www.warse.org/IJATCSE/static/pdf/file/ijatcse27952020.pdf.

[42] TUTORIALSPOINT. *Generate a Graph Using Dictionary in Python*. [En ligne]. Disponible, 2019. URL : https://www.tutorialspoint.com/generate-a-graph-using-dictionary-in-python.

[43] Douglas B. WEST. *Introduction to Graph Theory*. 2nd. Prentice Hall, 2001. URL : https://dwest.web.illinois.edu/igt/.