UNIVERSITÉ NATIONALE DES SCIENCES, TECHNOLOGIES, INGÉNIERIE ET MATHÉMATIQUES
(UNSTIM)

ÉCOLE NATIONALE SUPÉRIEURE DE GÉNIE MATHÉMATIQUE ET MODÉLISATION
(ENSGMM)

Ant Colony Optimization

Application à la gestion des groupes pédagogiques sur le campus de Sogbo-Aliho

Présenté par :

AFFOUKOU Prosper
AKANNI Eskil
ADANLAO Laurinda
ASSOU Marguerite
DOHOU Alexis

Encadré par :

Dr DANDOGBESSI Bruno

Cours : Optimisation Continue et Discrète
Année académique : 2025–2026

Table des matières

1 Introduction Contextuelle

1.1 Contexte du problème

La gestion efficace des ressources pédagogiques constitue un défi majeur pour les établissements d'enseignement supérieur. Sur le campus de Sogbo-Aliho, comme dans de nombreuses institutions académiques, l'affectation optimale des classes aux salles disponibles représente une tâche complexe qui influence directement la qualité de l'enseignement et l'utilisation rationnelle des infrastructures.

La problématique peut être formulée comme suit : étant donné un ensemble de classes avec leurs emplois du temps respectifs et un ensemble de salles avec des capacités limitées, comment affecter les classes aux salles de manière à minimiser les conflits d'emploi du temps tout en respectant les contraintes de capacité et de compatibilité ?

Ce problème appartient à la classe des **problèmes de timetabling universitaire** (University Timetabling Problem), largement étudié dans la littérature [6, 7].

1.2 Nature du problème d'optimisation

Le problème d'affectation de classes à des salles appartient à la classe des **problèmes d'optimisation combinatoire NP-difficiles**. Plus précisément, il s'agit d'un problème avec les caractéristiques suivantes :

— **Variables de décision discrètes** : L'affectation d'une classe à une salle est une décision binaire (0 ou 1)

— **Fonction objectif continue** : Le nombre de conflits d'emploi du temps est une valeur entière, mais peut être normalisée en une valeur continue

— **Contraintes multiples** : Contraintes de capacité, de compatibilité, et de non-chevauchement temporel

— **Espace de recherche exponentiel** : Pour n classes et k salles, l'espace des solutions possibles est de l'ordre de k^n

1.3 Pourquoi l'algorithme de colonies de fourmis (ACO) ?

L'Ant Colony Optimization (ACO) est particulièrement adapté à notre problématique pour plusieurs raisons fondamentales :

1.3.1 Adéquation avec les problèmes de timetabling

L'ACO a démontré son efficacité sur divers problèmes de planification universitaire [6, 7]. Socha et al. ont montré d'excellents résultats pour la construction d'emplois du temps universitaires en utilisant MAX-MIN Ant System.

1.3.2 Flexibilité et adaptabilité

Contrairement aux méthodes exactes qui deviennent impraticables pour des instances de grande taille, l'ACO offre une approche métaheuristique capable de :

— S'adapter à différentes fonctions objectif et contraintes

— Intégrer facilement de nouvelles contraintes (capacité, compatibilité, préférences)

— Fournir des solutions de bonne qualité en temps raisonnable

1.3.3 Équilibre exploitation-exploration

Le mécanisme de phéromones artificielles de l'ACO permet un équilibre naturel entre :

— **L'exploitation** : Renforcement des bonnes affectations trouvées

— **L'exploration** : Recherche de nouvelles configurations prometteuses

Cette propriété est cruciale pour éviter la convergence prématurée vers des optimums locaux [4].

1.3.4 Gestion des contraintes complexes

L'ACO gère naturellement les contraintes hard (qui ne peuvent être violées) et soft (préférences) en les intégrant dans la fonction heuristique et la fonction objectif.

1.4 Objectifs de l'étude

Ce travail vise à :

1. Développer un modèle mathématique formel du problème d'affectation de classes à des salles
2. Adapter l'algorithme ACO à ce contexte spécifique de timetabling
3. Implémenter une solution fonctionnelle en Python
4. Appliquer l'algorithme à un cas concret du campus de Sogbo-Aliho
5. Évaluer les performances et analyser les résultats
6. Comparer avec d'autres méthodes d'optimisation

2 Principe Biologique des Fourmis

2.1 Le comportement naturel des fourmis

L'algorithme ACO s'inspire directement du comportement de recherche de nourriture (*foraging*) observé chez certaines espèces de fourmis. Ce comportement remarquable illustre un principe fondamental d'intelligence collective : comment des agents simples, sans coordination centralisée, peuvent résoudre collectivement des problèmes complexes.

2.1.1 La découverte de nourriture

Lorsqu'une fourmi découvre une source de nourriture, elle retourne à la colonie en déposant sur son chemin une substance chimique appelée **phéromone** [9, 10]. Cette phéromone est sécrétée par des glandes exocrines situées dans l'abdomen postérieur de la fourmi [11].

2.1.2 Le mécanisme de renforcement

Les fourmis qui suivent utilisent leur perception chimique pour détecter les traces de phéromone. Elles ont tendance à suivre préférentiellement les chemins où la concentration de phéromone est plus élevée. Lorsqu'elles trouvent la nourriture à leur tour, elles renforcent le chemin en y déposant également de la phéromone [12].

Ce processus crée une **boucle de rétroaction positive** : plus un chemin est emprunté, plus il devient attractif pour les autres fourmis [13].

2.2 L'expérience du pont double

2.2.1 Protocole expérimental

Une expérience classique réalisée par Goss et al. (1989) [10] a démontré la capacité des fourmis à trouver le chemin le plus court. Le dispositif expérimental, appelé "pont double", consiste en :

— Une colonie de fourmis séparée d'une source de nourriture

— Deux branches de longueurs différentes reliant la colonie à la nourriture.

2.2.2 Résultats observés

Les résultats montrent que :

1. **Phase initiale** : Les fourmis explorent aléatoirement les deux branches avec une probabilité égale
2. **Phase d'accumulation** : Les fourmis empruntant la branche courte effectuent plus d'allers-retours dans le même temps, déposant plus de phéromone
3. **Phase de convergence** : La concentration de phéromone devient significativement plus élevée sur la branche courte

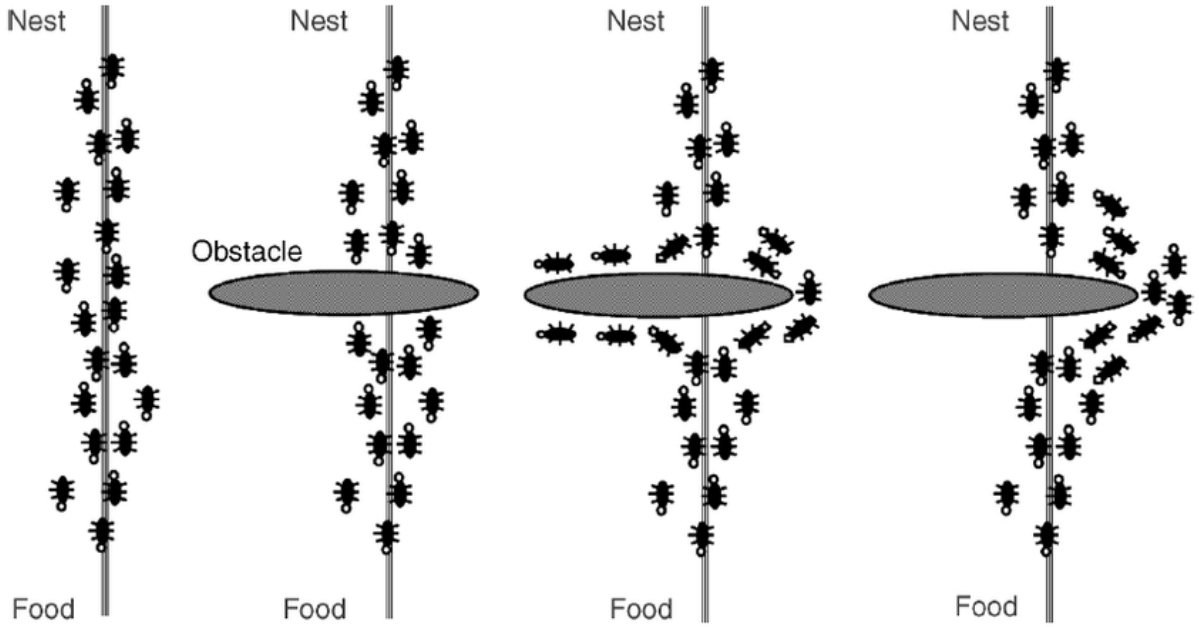

Figure 2.1 Expérience du pont double

4. **Phase d'optimisation** : Après un certain temps, la majorité des fourmis utilisent exclusivement le chemin court

2.2.3 Facteurs clés du succès

L'évaporation de la phéromone

La phéromone s'évapore naturellement au fil du temps [**?**]. Ce mécanisme est crucial car :

— Il permet d'oublier les mauvaises décisions initiales

— Il évite la stagnation sur des chemins sous-optimaux

— Il maintient une capacité d'adaptation aux changements

Le comportement probabiliste

Les fourmis ne suivent pas systématiquement le chemin le plus marqué. La probabilité de choisir un chemin dépend de la concentration de phéromone selon une relation non-linéaire. Cette stochasticité permet l'exploration continue de nouvelles routes.

La communication indirecte (stigmergie)

Le concept de **stigmergie** décrit cette forme de communication indirecte où l'environnement sert de support de communication. Les actions d'une fourmi influencent les décisions des autres via la modification de l'environnement.

2.3 Transposition au domaine algorithmique

La transposition de ce comportement naturel en algorithme informatique repose sur plusieurs analogies [5] :

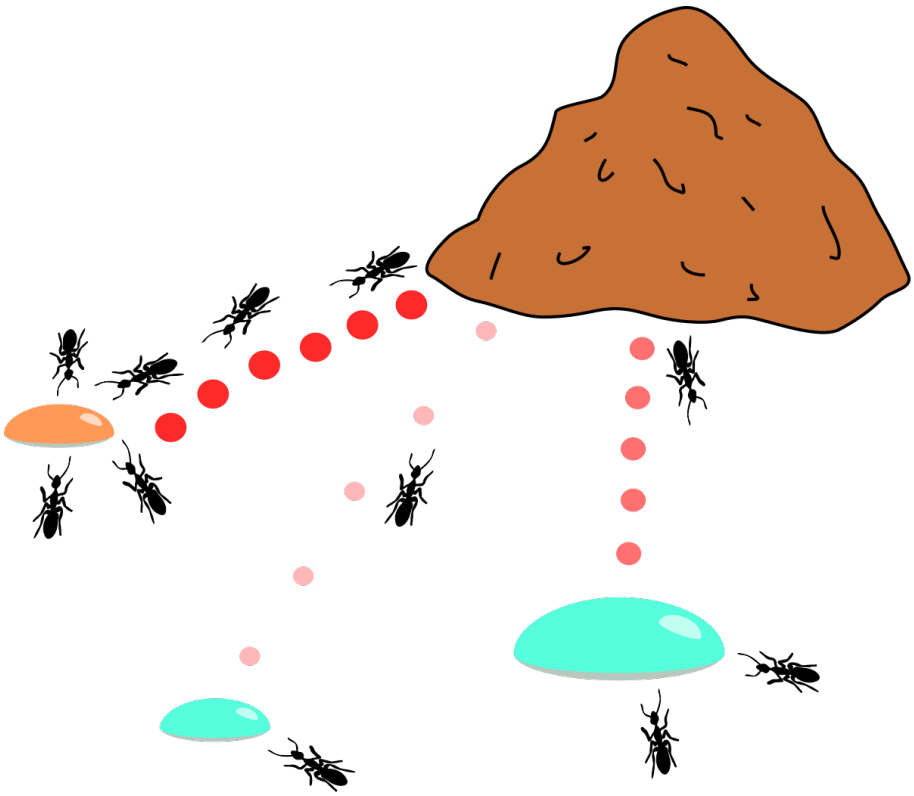

Figure 2.2 Traces de phéromone par des fourmis

Système naturel	**Système artificiel**
Fourmi réelle	Fourmi artificielle (agent)
Phéromone chimique	Phéromone numérique (matrice)
Chemin physique	Solution candidate
Concentration de phéromone	Qualité de la solution
Évaporation naturelle	Facteur d'évaporation ρ
Probabilité de choix	Règle de transition probabiliste

3 Description de l'Algorithme ACO

3.1 Structure générale de l'algorithme

L'Ant Colony Optimization est une métaheuristique itérative basée sur une population d'agents (fourmis artificielles) qui construisent des solutions de manière probabiliste [5]. La structure générale peut être décrite selon le schéma suivant :

Algorithm 1 Structure générale de l'ACO

```
Initialiser les paramètres
Initialiser les phéromones τ_ij = τ_0 pour tous les composants
while condition d'arrêt non atteinte do
    for chaque fourmi k = 1 à m do
        Construire une solution S^k en utilisant les phéromones et l'information heuristique
    end for
    (Optionnel) Appliquer une recherche locale aux solutions construites
    Mettre à jour les phéromones selon la qualité des solutions
    Appliquer l'évaporation des phéromones
end while
return Meilleure solution trouvée
```

3.2 Composants clés de l'algorithme

3.2.1 Représentation du problème

Pour appliquer l'ACO à un problème, il faut définir :

— **Graphe de construction** $G = (C, L)$ où C est l'ensemble des composants de solution et L l'ensemble des connexions possibles

— **Contraintes du problème** Ω définissant les solutions réalisables

— **Fonction objectif** $f : S \to \mathbb{R}$ à minimiser ou maximiser

3.2.2 Information phéromonale

Les phéromones artificielles sont représentées par une matrice τ :

— τ_{ij} : niveau de phéromone associé à l'affectation de la classe i à la salle j

— τ_0 : valeur initiale de phéromone

— τ_{min}, τ_{max} : bornes (MAX-MIN Ant System)

3.2.3 Information heuristique

L'information heuristique η_{ij} représente la désirabilité a priori d'affecter la classe i à la salle j, basée sur :

— Compatibilité de capacité

— Absence de conflits horaires

— Préférences pédagogiques

3.2.4 Règle de transition probabiliste

La probabilité qu'une fourmi k affecte la classe i à la salle j est :

$$p_{ij}^k = \begin{cases} \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [\tau_{il}]^\alpha \cdot [\eta_{il}]^\beta} & \text{si } j \in N_i^k \\ 0 & \text{sinon} \end{cases} \tag{3.1}$$

où :

— N_i^k : ensemble des salles compatibles pour la classe i

— α : paramètre d'influence de la phéromone ($\alpha > 0$)

— β : paramètre d'influence de l'heuristique ($\beta \geq 0$)

3.2.5 Mise à jour des phéromones

La mise à jour des phéromones combine deux mécanismes :

Évaporation

$$\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij} \tag{3.2}$$

où $\rho \in (0, 1]$ est le taux d'évaporation.

Dépôt de phéromone

Seule la meilleure fourmi dépose de la phéromone :

$$\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau_{ij}^{best} \tag{3.3}$$

où :

$$\Delta\tau_{ij}^{best} = \begin{cases} \frac{Q}{f(S^{best})} & \text{si } (i,j) \text{ est dans } S^{best} \\ 0 & \text{sinon} \end{cases} \tag{3.4}$$

3.3 MAX-MIN Ant System

Le MAX-MIN Ant System (MMAS) [4] améliore l'ACO classique en :

— Imposant des bornes $[\tau_{min}, \tau_{max}]$ sur les phéromones

— Ne laissant que la meilleure fourmi déposer de la phéromone

— Initialisant les phéromones à τ_{max} pour favoriser l'exploration initiale

Formule de mise à jour :

$$\tau_{ij} = \max\{\tau_{min}, \min\{\tau_{max}, (1 - \rho)\tau_{ij} + \Delta\tau_{ij}^{best}\}\} \tag{3.5}$$

4 Modélisation Mathématique du Problème

4.1 Définition formelle du problème

Le problème d'affectation de classes à des salles peut être formulé comme un problème d'optimisation combinatoire. Nous disposons d'un ensemble de classes $C = \{c_1, c_2, \ldots, c_n\}$ que nous devons affecter à un ensemble de salles $R = \{r_1, r_2, \ldots, r_k\}$ avec $k < n$.

4.1.1 Variables de décision

Nous définissons les variables de décision binaires suivantes :

$$x_{ij} = \begin{cases} 1 & \text{si la classe } c_i \text{ est affectée à la salle } r_j \\ 0 & \text{sinon} \end{cases} \quad \forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, k\} \tag{4.1}$$

Ces variables constituent un problème d'optimisation **discret** puisque $x_{ij} \in \{0, 1\}$.

4.1.2 Paramètres du problème

Pour chaque classe c_i

- E_i : effectif de la classe (nombre d'étudiants)
- $T_i = \{t_i^1, t_i^2, \ldots, t_i^{p_i}\}$: ensemble des créneaux horaires occupés par la classe
- L_i : niveau d'études (L1, L2, M1, etc.)
- D_i : département/filière (Informatique, Mathématiques, etc.)

Pour chaque salle r_j

- Cap_j : capacité maximale de la salle (nombre de places)
- $Type_j$: type de salle (amphithéâtre, salle TD, laboratoire, etc.)

Matrice de compatibilité

$$comp_{ij} = \begin{cases} 1 & \text{si la classe } c_i \text{ est compatible avec la salle } r_j \\ 0 & \text{sinon} \end{cases} \tag{4.2}$$

La compatibilité dépend du type de cours, des équipements nécessaires, etc.

4.1.3 Fonction de conflit temporel

Pour deux classes c_i et c_h affectées à la même salle r_j, un **conflit** se produit si leurs emplois du temps se chevauchent :

$$conflict(c_i, c_h) = \begin{cases} 1 & \text{si } T_i \cap T_h \neq \emptyset \\ 0 & \text{sinon} \end{cases} \tag{4.3}$$

4.2 Fonction objectif

L'objectif principal est de **minimiser le nombre total de conflits d'emploi du temps**.

4.2.1 Nombre de conflits

Pour une solution $x = (x_{ij})$, le nombre total de conflits est :

$$f_1(x) = \sum_{j=1}^{k} \sum_{i=1}^{n} \sum_{h=i+1}^{n} x_{ij} \cdot x_{hj} \cdot conflict(c_i, c_h) \tag{4.4}$$

Cette formule compte, pour chaque salle, le nombre de paires de classes affectées à cette salle dont les emplois du temps se chevauchent.

4.2.2 Critères secondaires

Équilibre de charge des salles

On peut souhaiter équilibrer le nombre de classes par salle :

$$f_2(x) = \sum_{j=1}^{k} \left(\sum_{i=1}^{n} x_{ij} - \frac{n}{k} \right)^2 \tag{4.5}$$

Taux d'occupation des salles

Minimiser le gaspillage de capacité :

$$f_3(x) = \sum_{j=1}^{k} \sum_{i=1}^{n} x_{ij} \cdot \max\{0, Cap_j - E_i\} \tag{4.6}$$

4.2.3 Fonction objectif globale

La fonction objectif globale est une combinaison pondérée :

$$f(x) = w_1 \cdot f_1(x) + w_2 \cdot f_2(x) + w_3 \cdot f_3(x) \tag{4.7}$$

où $w_1, w_2, w_3 > 0$ sont des poids avec $w_1 \gg w_2, w_3$ car minimiser les conflits est prioritaire.
Cette fonction objectif est **continue** bien que les variables de décision soient discrètes.

4.3 Contraintes

Les contraintes du problème de timetabling peuvent être classées en deux catégories distinctes [14, 16] :

4.3.1 Contraintes hard (obligatoires)

Les contraintes hard ne peuvent en aucun cas être violées. Une solution qui viole au moins une contrainte hard est considérée comme **non réalisable**.

H1 Non-conflit étudiant : Aucune classe ne peut être affectée à plus d'une salle au même moment

H2 Compatibilité salle-cours : La salle doit satisfaire les équipements requis par le cours (projecteur, laboratoire, etc.)

H3 Respect de capacité : Le nombre d'étudiants de la classe doit être inférieur ou égal à la capacité de la salle

H4 Non-double occupation : Au plus un cours peut occuper une salle à un moment donné

Formellement, ces contraintes peuvent s'écrire :

$$\sum_{j=1}^{k} x_{ij} = 1 \quad \forall i \in \{1, \ldots, n\} \qquad \text{(H1)} \tag{4.8}$$

$$x_{ij} \leq comp_{ij} \quad \forall i, j \qquad \text{(H2)} \tag{4.9}$$

$$x_{ij} \cdot (E_i - Cap_j) \leq 0 \quad \forall i, j \qquad \text{(H3)} \tag{4.10}$$

$$x_{ij} + x_{hj} \leq 1 \quad \forall j, \forall i \neq h : T_i \cap T_h \neq \emptyset \qquad \text{(H4)} \tag{4.11}$$

4.3.2 Contraintes soft (préférences)

Les contraintes soft représentent des préférences qui améliorent la qualité de l'emploi du temps mais dont la violation n'invalide pas la solution. L'objectif est de minimiser le nombre total de violations [14].

S1 Limitation journalière : Une classe ne devrait avoir qu'un seul cours par jour

S2 Éviter les séquences longues : Éviter plus de 2 cours consécutifs pour une classe

S3 Éviter la dernière période : Éviter d'affecter des cours à la dernière période de la journée

S4 Préférences enseignants : Respecter les créneaux préférés des enseignants

S5 Équilibrage de charge : Distribuer équitablement les cours entre les salles

La violation des contraintes soft peut être quantifiée par une fonction de pénalité :

$$penalty(x) = w_{S1} \cdot v_{S1}(x) + w_{S2} \cdot v_{S2}(x) + \cdots + w_{S5} \cdot v_{S5}(x) \tag{4.12}$$

où $v_{Si}(x)$ est le nombre de violations de la contrainte soft S_i et w_{Si} est son poids de pénalité.

4.4 Formulation complète du problème

$$\min_{x_{ij}} \quad f(x) = w_1 \cdot f_1(x) + w_2 \cdot f_2(x) + w_3 \cdot f_3(x) \tag{4.13}$$

$$\text{s.c.} \quad \sum_{j=1}^{k} x_{ij} = 1 \quad \forall i \in \{1, \ldots, n\} \tag{4.14}$$

$$x_{ij} \cdot (E_i - Cap_j) \leq 0 \quad \forall i, j \tag{4.15}$$

$$x_{ij} \leq comp_{ij} \quad \forall i, j \tag{4.16}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \tag{4.17}$$

4.5 Complexité du problème

4.5.1 Nature NP-difficile

Le problème d'affectation de classes à des salles avec contraintes de timetabling est **NP-difficile** [6]. Il peut être réduit au problème de coloration de graphes, qui est NP-complet.

L'espace de recherche contient k^n solutions possibles. Pour $n = 30$ classes et $k = 10$ salles, cela représente 10^{30} solutions, ce qui rend impraticable une énumération exhaustive.

4.5.2 Justification de l'approche métaheuristique

Face à cette complexité, les métaheuristiques comme l'ACO offrent un compromis efficace entre qualité de la solution et temps de calcul [3, 6].

4.6 Représentation graphique du problème

Pour appliquer l'ACO, nous modélisons le problème comme un **graphe de construction biparti** $G = (C \cup R, E)$ où :

— C : ensemble des classes

— R : ensemble des salles

— E : ensemble des arêtes (c_i, r_j) représentant les affectations possibles

Une **solution** correspond à une affectation complète de toutes les classes aux salles.

Les **phéromones** τ_{ij} sont déposées sur les arêtes, représentant la "désirabilité" d'affecter la classe c_i à la salle r_j basée sur l'expérience collective des fourmis.

5 Adaptation de l'ACO au Problème de Timetabling

5.1 Information heuristique

L'information heuristique η_{ij} guide les fourmis vers des affectations localement bonnes. Pour notre problème, nous définissons plusieurs composantes.

5.1.1 Heuristique de capacité

Plus la capacité de la salle est proche de l'effectif de la classe, plus l'affectation est désirable :

$$\eta_{ij}^{cap} = \frac{1}{1 + |Cap_j - E_i|} \tag{5.1}$$

5.1.2 Heuristique de conflit

Moins une salle a de classes déjà affectées qui entrent en conflit avec la classe c_i, plus l'affectation est désirable. Soit S_j^k l'ensemble des classes déjà affectées à la salle r_j dans la solution partielle S^k :

$$\eta_{ij}^{conf} = \frac{1}{1 + \sum_{c_h \in S_j^k} conflict(c_i, c_h)} \tag{5.2}$$

5.1.3 Heuristique de charge

Favoriser les salles ayant moins de classes affectées pour équilibrer la charge :

$$\eta_{ij}^{load} = \frac{1}{1 + |S_j^k|} \tag{5.3}$$

5.1.4 Heuristique combinée avec paramètres de disponibilité

En s'inspirant des travaux d'Aslan et Aci [15], nous enrichissons l'information heuristique en intégrant les disponibilités des enseignants. L'heuristique finale devient :

$$\eta_{ij} = \eta_{ij}^{cap} \cdot (\eta_{ij}^{conf})^2 \cdot \eta_{ij}^{load} \cdot \eta_{ij}^{avail} \tag{5.4}$$

où η_{ij}^{avail} reflète la disponibilité de l'enseignant pour le créneau considéré.

5.1.5 Formule ACO enrichie pour le timetabling

La règle de transition probabiliste complète, adaptée de [14, 15], devient :

$$p_{ij}^k = \begin{cases} \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta \cdot [\mu_j]^\theta \cdot [\lambda_j]^\delta \cdot [\nu_{jh}]^\gamma}{\sum_{l \in N_i^k} [\tau_{il}]^\alpha \cdot [\eta_{il}]^\beta \cdot [\mu_l]^\theta \cdot [\lambda_l]^\delta \cdot [\nu_{lh}]^\gamma} & \text{si } j \in N_i^k \\ 0 & \text{sinon} \end{cases} \tag{5.5}$$

où :

— μ_j : heures théoriques du cours j

— λ_j : disponibilité hebdomadaire totale de l'enseignant

— ν_{jh} : disponibilité de l'enseignant pour le créneau horaire h

— θ, δ, γ : paramètres d'importance relative de ces facteurs

Cette formulation permet de prendre en compte non seulement les contraintes de capacité et de conflits, mais aussi les préférences et disponibilités des enseignants.

5.2 Méthode de sélection par roulette

Pour la sélection probabiliste des salles et des cours, nous utilisons la **méthode de la roulette** (roulette wheel selection) [15].

5.2.1 Principe

Étant donné un ensemble de n choix avec des probabilités $p_1, p_2, \ldots, p_n$ où $\sum_{i=1}^{n} p_i = 1$, la méthode de la roulette sélectionne un élément j en :

1. Calculant les probabilités cumulatives : $P_i = \sum_{l=1}^{i} p_l$
2. Générant un nombre aléatoire $r \in [0, 1]$
3. Sélectionnant l'élément j tel que $P_{j-1} < r \leq P_j$

5.2.2 Algorithme de sélection

Algorithm 2 Sélection par roulette

Entrée : Probabilités $p_1, \ldots, p_n$
Sortie : Indice sélectionné j

$cumul[1] \leftarrow p_1$
for $i = 2$ **à** n **do**
    $cumul[i] \leftarrow cumul[i-1] + p_i$
end for
$r \leftarrow random(0, 1)$
for $j = 1$ **à** n **do**
    if $r \leq cumul[j]$ **then**
        return j
    end if
end for

Cette méthode garantit que chaque choix est sélectionné avec une probabilité proportionnelle à p_i, tout en introduisant de la stochasticité nécessaire à l'exploration de l'espace de recherche.

5.3 Algorithme ACO adapté pour le problème

Algorithm 3 ACO pour l'affectation de classes à des salles

Entrées :
  $C = \{c_1, \ldots, c_n\}$: classes avec effectifs E_i et emplois du temps T_i
  $R = \{r_1, \ldots, r_k\}$: salles avec capacités Cap_j
  $comp_{ij}$: matrice de compatibilité
  m : nombre de fourmis
  α, β, ρ, Q : paramètres ACO
  $MaxIterations$: nombre maximum d'itérations

Initialisation :
$\tau_{ij} \leftarrow \tau_{max}$ pour tout i, j compatible
$S^{best} \leftarrow \emptyset$, $f^{best} \leftarrow +\infty$
$iteration \leftarrow 0$

while $iteration < MaxIterations$ **et non convergence do**
  $S^{best\_cycle} \leftarrow \emptyset$, $f^{best\_cycle} \leftarrow +\infty$
  for $ant = 1$ **à** m **do**
    // Construction de solution
    $S^{ant} \leftarrow \emptyset$
    $C_{restantes} \leftarrow C$
    $S_j^{ant} \leftarrow \emptyset$ pour tout j ▷ Classes affectées à chaque salle
    while $C_{restantes} \neq \emptyset$ **do**
      Sélectionner aléatoirement $c_i \in C_{restantes}$
      $N_i \leftarrow \{r_j : comp_{ij} = 1 \text{ et } E_i \leq Cap_j\}$ ▷ Salles compatibles
      for chaque $r_j \in N_i$ **do**
        Calculer $\eta_{ij}^{cap} = \frac{1}{1+\lvert Cap_j - E_i \rvert}$
        Calculer $\eta_{ij}^{conf} = \frac{1}{1+\sum_{c_h \in S_j^{ant}} conflict(c_i, c_h)}$
        Calculer $\eta_{ij}^{load} = \frac{1}{1+\lvert S_j^{ant} \rvert}$
        Calculer $\eta_{ij} = \eta_{ij}^{cap} \cdot (\eta_{ij}^{conf})^2 \cdot \eta_{ij}^{load}$
      end for
      Calculer $p_{ij}^{ant} = \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N_i} [\tau_{il}]^\alpha \cdot [\eta_{il}]^\beta}$ pour tout $j \in N_i$
      Sélectionner r_{j^*} selon la distribution de probabilité p_{ij}^{ant}
      $S^{ant} \leftarrow S^{ant} \cup \{(c_i, r_{j^*})\}$
      $S_{j^*}^{ant} \leftarrow S_{j^*}^{ant} \cup \{c_i\}$
      $C_{restantes} \leftarrow C_{restantes} \setminus \{c_i\}$
    end while
    // Évaluation de la solution
    Calculer $f(S^{ant})$ selon l'équation (4.8)
    if $f(S^{ant}) < f^{best\_cycle}$ **then**
      $S^{best\_cycle} \leftarrow S^{ant}$
      $f^{best\_cycle} \leftarrow f(S^{ant})$
    end if
    if $f(S^{ant}) < f^{best}$ **then**
      $S^{best} \leftarrow S^{ant}$
      $f^{best} \leftarrow f(S^{ant})$
    end if
  end for
  // Mise à jour des phéromones (MAX-MIN)
  // Évaporation
  for tout (i, j) compatible **do**
    $\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij}$
  end for
  // Dépôt de phéromone
  $\Delta\tau \leftarrow \frac{Q}{1+f^{best\_cycle} - f^{best}}$
  for chaque affectation $(c_i, r_j) \in S^{best\_cycle}$ **do**
    $\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau$
  end for
  // Application des bornes MAX-MIN
  for tout (i, j) compatible **do**
    $\tau_{ij} \leftarrow \max\{\tau_{min}, \min\{\tau_{max}, \tau_{ij}\}\}$
  end for
  $iteration \leftarrow iteration + 1$
end while
Retourner S^{best}

5.4 Paramètres de l'algorithme

D'après les travaux sur le timetabling universitaire avec ACO [6, 8], voici les paramètres recommandés :

Paramètre	Valeur	Rôle
α	1	Influence de la phéromone
β	5	Influence de l'heuristique (privilégie l'évitement de conflits)
ρ	0.01	Taux d'évaporation (favorise l'exploration)
m	30	Nombre de fourmis
Q	100	Constante de dépôt
τ_{min}	0.01	Borne inférieure (MAX-MIN)
τ_{max}	6	Borne supérieure (MAX-MIN)
w_1	1000	Poids des conflits (prioritaire)
w_2	1	Poids de l'équilibre
w_3	0.1	Poids de l'occupation

5.5 Complexité algorithmique

5.5.1 Complexité d'une itération

Pour chaque fourmi :

— Construction : $O(n \cdot k)$ (pour n classes et k salles)

— Évaluation des conflits : $O(n^2)$ dans le pire cas

Pour m fourmis : $O(m \cdot n \cdot (k+n))$
Mise à jour des phéromones : $O(n \cdot k)$
Pour T itérations : $O(T \cdot m \cdot n \cdot (k+n))$

5.5.2 Exemple numérique

Pour $n = 30$ classes, $k = 10$ salles, $m = 30$ fourmis, $T = 1000$ itérations :

$$1000 \times 30 \times 30 \times (10+30) \approx 3.6 \times 10^7 \text{ opérations} \tag{5.6}$$

Temps de calcul estimé : quelques dizaines de secondes sur un ordinateur moderne.

6 Implémentation Python et Application au Campus de Sogbo-Aliho

6.1 Vue d'ensemble du système

6.1.1 Architecture modulaire

L'implémentation Python repose sur une architecture modulaire organisée en trois classes principales, garantissant une séparation claire des responsabilités et une maintenabilité optimale du code.

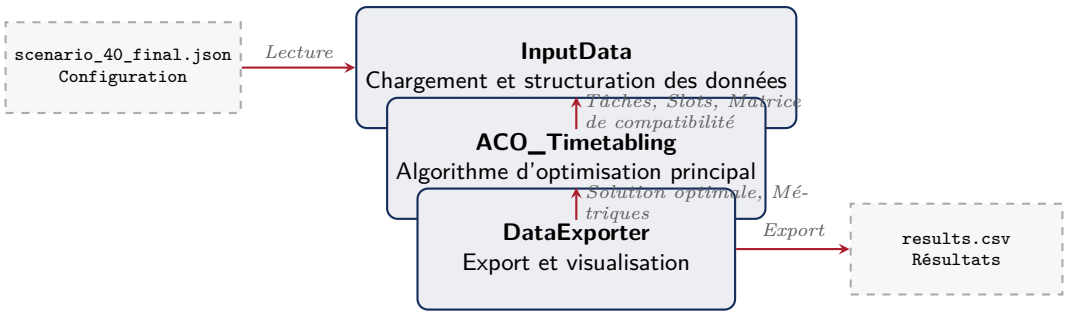

Figure 6.1 Architecture modulaire du système d'optimisation

6.1.2 Flux de données

Le système suit un pipeline d'exécution en quatre phases successives :

I. **Initialisation** : Chargement des paramètres depuis le fichier JSON de configuration.

II. **Prétraitement** : Génération des tâches (cours) et slots (salles × créneaux), construction de la matrice de compatibilité.

III. **Optimisation** : Exécution de l'algorithme MAX-MIN Ant System avec évaluation des solutions.

IV. **Post-traitement** : Export des résultats au format CSV pour analyse.

6.1.3 Technologies utilisées

— **Python 3.9** : Langage principal pour sa simplicité et richesse des bibliothèques scientifiques

— **NumPy** : Manipulation efficace des matrices (phéromones, compatibilité)

— **JSON** : Format de configuration lisible et modifiable

— **CSV** : Export standard compatible avec Excel, R, et autres outils

— **Jupyter Notebook** : Environnement interactif pour le développement et tests

6.2 Scénario détaillé du Campus de Sogbo-Aliho

6.2.1 Classes et effectifs

Le campus compte 10 classes réparties en trois filières, avec un total de 352 étudiants.

Table 6.1 Classes du campus de Sogbo-Aliho

Classe	Effectif	Filière	Matières (4 par classe)
ENSTP-L1	75	Travaux Publics	Mathématiques I, Physique I, RDM I, Topographie I
ENSTP-L3	55	Travaux Publics	Mathématiques III, Physique III, RDM III, Projet Construction
ENSTP-I1	60	Travaux Publics	Mécanique des Fluides, Géotechnique I, Matériaux, Hydraulique
ENSTP-I3	20	Travaux Publics	Béton Armé, Charpente Métallique, Voiries, Gestion de Projet
GMM-I1	27	Génie Mathématique	Analyse Fonctionnelle I, Algèbre Linéaire I, Modélisation I, Optimisation I
GMM-I2	25	Génie Mathématique	Analyse II, Algèbre II, Modélisation II, Optimisation II
GMM-I3	23	Génie Mathématique	Analyse III, Algèbre III, Modélisation III, Optimisation III
GEP-I1	27	Génie Énergétique	Thermodynamique I, Transfert Chaleur I, Énergies I, Chimie I
GEP-I2	20	Génie Énergétique	Thermodynamique II, Transfert II, Énergies II, Chimie II
GEP-I3	20	Génie Énergétique	Thermodynamique III, Transfert III, Énergies III, Chimie III

6.2.2 Salles et capacités

Le campus dispose de 7 salles aux capacités variées, adaptées aux différents types d'enseignement.

Table 6.2 Salles disponibles sur le campus

Salle	Capacité	Type	Utilisation prévue
TD-1	30	Salle de TD	Groupes GMM et GEP (20-27 étudiants)
TD-2	30	Salle de TD	Groupes GMM et GEP (20-27 étudiants)
TD-3	30	Salle de TD	Groupes GMM et GEP (20-27 étudiants)
TD-4	30	Salle de TD	Groupes GMM et GEP (20-27 étudiants)
SALLE-60	60	Salle de cours	ENSTP-L3, ENSTP-I1
AMPHI-100	100	Amphithéâtre	ENSTP-L1, ENSTP-L3
AMPHI-200	200	Amphithéâtre	ENSTP-L1 (grands groupes)

6.2.3 Structure temporelle

La grille horaire hebdomadaire est structurée en :

— **6 jours** : Lundi (L), Mardi (Ma), Mercredi (Me), Jeudi (J), Vendredi (V), Samedi (S)

— **2 périodes par jour** : Matin (8h-11h) et Après-midi (14h-17h)

— **Total** : 12 créneaux horaires distincts

6.2.4 Cas particulier INSPEI (pré-affectation)

Les classes préparatoires INSPEI (Prépa-1 et Prépa-2, 60 étudiants chacune) font l'objet d'une **pré-affectation** spécifique :

— Elles disposent de salles dédiées (INSPEI-1 et INSPEI-2, 60 places chacune)

— Leur emploi du temps est fixé en amont et n'est pas optimisé par l'ACO

— Cette séparation permet de réduire la complexité du problème tout en respectant les contraintes opérationnelles

6.2.5 Modélisation en "Tâches" et "Slots"

Pour transformer le problème concret en problème d'optimisation, nous introduisons deux abstractions fondamentales :

Tâche Une tâche $t_i \in T$ représente un cours spécifique à planifier, défini par le triplet (*Classe*, *Matière*, *Effectif*). Par exemple : `GMM-I1_Algèbre_Linéaire_I` (27 étudiants).

Slot Un slot $s_j \in S$ représente une ressource spatio-temporelle disponible, définie par le couple (*Salle*, *Créneau*). Par exemple : `TD-1_Lundi_Matin` (salle TD-1, 30 places, lundi 8h-11h).

Table 6.3 Caractéristiques du problème d'optimisation

Élément	**Symbole**	**Valeur**
Tâches à affecter	$n = \|T\|$	40 (10 classes × 4 matières)
Slots disponibles	$k = \|S\|$	84 (7 salles × 12 créneaux)
Taux d'occupation	$\frac{n}{k}$	47.6%
Solutions possibles	k^n	$84^{40} \approx 1.3 \times 10^{77}$

6.2.6 Mappage avec les variables mathématiques

Le tableau ci-dessous établit la correspondance entre la modélisation mathématique (Chapitre 4) et l'implémentation Python :

Table 6.4 Correspondance variables mathématiques implémentation

Concept mathématique	**Symbole**	**Implémentation Python**
Variable de décision	$x_{ij} \in \{0, 1\}$	Dictionnaire `solution_ant[tache_id] = slot_id`
Matrice de compatibilité	$comp_{ij}$	`COMPATIBILITY_MATRIX[i, j]`
Matrice de phéromones	τ_{ij}	`pheromones[i, j]`
Information heuristique	η_{ij}	Méthode `_calculate_heuristic()`
Fonction objectif	$f(x) = w_1 f_1 + w_2 f_2 + w_3 f_3$	`calculate_objective_function()`
Contrainte H1 (non-conflit)	$\sum_j x_{ij} = 1$	Vérification par créneau dans `_get_f1_conflicts()`
Contrainte H2 (compatibilité)	$x_{ij} \leq comp_{ij}$	Matrice binaire de compatibilité
Contrainte H3 (capacité)	$E_i \leq Cap_j$	Filtrage dans `_build_compatibility_matrix()`

6.3 Module InputData : Structuration des données

6.3.1 Chargement de la configuration JSON

Le module `InputData` charge la configuration depuis un fichier JSON structuré contenant :

— Les paramètres de l'algorithme ACO (α, β, ρ, etc.)

— La description des salles (nom, capacité, type)

— Les classes avec leurs effectifs et listes de matières

6.3.2 Génération des tâches (Classe × Matière)

Chaque combinaison (Classe, Matière) devient une tâche unique. L'identifiant est généré automatiquement pour garantir l'unicité.

```
def generate_tasks(self, config):
    """G n re la liste des t ches  partir de la configuration."""
    tasks = []
    for classe, details in config['CLASSES_DETAILS'].items():
        effectif = details['effectif']
        for matiere in details['matieres']:
            task_id = f"{classe}_{matiere.replace(' ','_')}"
            tasks.append({
                'id': task_id,
                'classe': classe,
                'matiere': matiere,
                'effectif': effectif
            })
    return tasks
```

6.3.3 Génération des slots (Salle × Créneau)

Les slots sont créés par produit cartésien entre les salles et les créneaux horaires.

```
def generate_slots(self):
    """G n re tous les slots disponibles (Salle  Cr neau)."""
    slots = {}
    for salle, (capacite, type_salle) in self.SALLES.items():
        for jour in ['L', 'Ma', 'Me', 'J', 'V', 'S']:
            for periode in ['Matin', 'Aprem']:
                creneau = f"{jour}_{periode}"
                slot_id = f"{salle}_{creneau}"
                slots[slot_id] = {
                    'Salle': salle,
                    'Creneau': creneau,
                    'Capacite': capacite,
                    'Type_Salle': type_salle
                }
    return slots
```

6.3.4 Matrice de compatibilité

La matrice binaire $comp_{ij}$ encode simultanément les contraintes H2 (compatibilité) et H3 (capacité).

```
def _build_compatibility_matrix(self):
    """Construit la matrice de compatibilit  T ches  Slots."""
    matrix = np.zeros((self.N_TACHES, self.N_SLOTS))
```

```
4
5       for i, tache in enumerate(self.TACHES):
6           effectif = tache['Effectif']
7           classe_name = tache['Classe']
8
9           for j, slot_id in enumerate(self.SLOT_IDS):
10              slot = self.SLOTS[slot_id]
11              capacite = slot['Capacite']
12              type_salle = self.SALLES[slot['Salle']][1]
13
14              # H3: Respect de capacit
15              if effectif <= capacite:
16                  # H2: Compatibilit  type de salle
17                  if self._is_td_class(classe_name):
18                      if type_salle == 'TD':
19                          matrix[i, j] = 1
20                  else:  # Classe ENSTP
21                      if type_salle in ['SC', 'AMPHI']:
22                          matrix[i, j] = 1
23      return matrix
```

6.4 Module ACO_Timetabling : Cœur de l'algorithme

6.4.1 Initialisation

L'initialisation suit le schéma MAX-MIN Ant System (MMAS) avec bornes sur les phéromones.

```
def __init__(self, data, config):
    self.data = data
    self.params = config['ACO_PARAMS']

    # Initialisation des param tres
    self.alpha = self.params.get('alpha', 1)
    self.beta = self.params.get('beta', 5)
    self.rho = self.params.get('rho', 0.01)
    self.tau_max = self.params.get('tau_max', 6)
    self.tau_min = self.params.get('tau_min', 0.01)

    # Initialisation des ph romones (MAX-MIN)
    self.pheromones = self.data.COMPATIBILITY_MATRIX.copy()
    self.pheromones[self.pheromones == 1] = self.tau_max
    self.pheromones[self.pheromones == 0] = 0
```

6.4.2 Calcul de l'heuristique

L'information heuristique combine trois facteurs selon l'équation $\eta_{ij} = \eta_{ij}^{cap} \times \eta_{ij}^{conf} \times \eta_{ij}^{load}$.

```
def _calculate_heuristic(self, tache_id, slot_id, solution_partial):
    """Calcule␣l'information␣heuristique␣pour␣une␣affectation."""
    tache = self.data.tache_map[tache_id]
    slot = self.data.SLOTS[slot_id]

    effectif = tache['Effectif']
    capacite = slot['Capacite']

    #  _cap  : Proximit  capacit -effectif
    eta_cap = 1.0 / (1 + abs(capacite - effectif))

    #  _conf  :  viter  les conflits H1
```

```
13      n_conflicts = self._count_conflicts(tache, slot, solution_partial)
14      eta_conf = 1.0 / (1 + n_conflicts**2)
15
16      #  _load  :  quilibrer  la charge de la salle
17      tasks_in_room = self._count_tasks_in_room(slot['Salle'], solution_partial)
18      eta_load = 1.0 / (1 + tasks_in_room)
19
20      return eta_cap * eta_conf * eta_load
```

6.4.3 Construction de solution

Chaque fourmi construit une solution complète en sélectionnant itérativement des slots pour chaque tâche.

```
1  def construct_solution(self):
2      """Une␣fourmi␣construit␣une␣solution␣compl te."""
3      solution = {}
4      taches_restantes = list(self.data.TACHE_IDS)
5      random.shuffle(taches_restantes)
6
7      for tache_id in taches_restantes:
8          tache_idx = self.data.TACHE_IDS.index(tache_id)
9          allowed_slots = np.where(self.data.COMPATIBILITY_MATRIX[tache_idx, :] == 1)[0]
10
11         # Calcul des probabilit s
12         probs = self._transition_probability(tache_idx, allowed_slots, solution)
13
14         # S lection par roulette
15         selected_slot_idx = random.choices(allowed_slots, weights=probs, k=1)[0]
16         slot_id = self.data.SLOT_IDS[selected_slot_idx]
17
18         solution[tache_id] = slot_id
19
20     return solution
```

6.4.4 Évaluation (f1, f2, f3)

La fonction objectif évalue trois critères avec pondérations différentes.

```
1  def calculate_objective_function(self, solution):
2      """Calcule␣f(x)␣=␣w1*f1␣+␣w2*f2␣+␣w3*f3"""
3      # f1 : Conflits H1
4      f1 = self._count_h1_conflicts(solution)
5
6      # f2 :  quilibre  de charge
7      f2 = self._calculate_load_balance(solution)
8
9      # f3 : Gaspillage de capacit
10     f3 = self._calculate_waste(solution)
11
12     # Pond rations
13     w1, w2, w3 = 1000, 1, 0.1
14     return w1*f1 + w2*f2 + w3*f3, f1, f2, f3
```

6.4.5 Mise à jour des phéromones

La mise à jour suit le schéma MMAS avec évaporation et dépôt bornés.

```python
def update_pheromones(self, best_solution, f_best):
    """Mise␣ ␣jour␣MMAS␣des␣ph romones."""
    #   vaporation
    self.pheromones = (1 - self.rho) * self.pheromones

    # D  p  t par la meilleure solution
    delta_tau = self.Q / (f_best + 1.0)
    for tache_id, slot_id in best_solution.items():
        i = self.data.TACHE_IDS.index(tache_id)
        j = self.data.SLOT_IDS.index(slot_id)
        self.pheromones[i, j] += delta_tau

    # Application des bornes MAX-MIN
    self.pheromones = np.clip(self.pheromones, self.tau_min, self.tau_max)
```

6.5 Analyse de la complexité algorithmique

6.5.1 Complexité théorique en fonction de n et k

La complexité de l'algorithme ACO peut être analysée en fonction des paramètres clés du problème :

— n : nombre de tâches à affecter

— k : nombre de slots disponibles

— m : nombre de fourmis par itération

— T : nombre d'itérations

Complexité par opération

Table 6.5 Analyse détaillée de la complexité

Opération	Complexité	Description
Génération des tâches	$O(n)$	Simple parcours linéaire
Génération des slots	$O(k)$	Produit cartésien salles × créneaux
Construction matrice compatibilité	$O(n \times k)$	Vérification pour chaque paire (tâche, slot)
Construction solution (par fourmi)	$O(n \times k)$	Pour chaque tâche, évaluation de tous les slots compatibles
Calcul heuristique η_{ij}	$O(1)$	Opérations élémentaires par paire (tâche, slot)
Évaluation fonction objectif	$O(n)$	Parcours linéaire pour f1, f2, f3
Mise à jour phéromones	$O(n \times k)$	Évaporation + dépôt sur toutes les paires

Complexité totale de l'algorithme

Pour T itérations avec m fourmis, la complexité totale est :

$$C_{\text{total}} = O(T \times m \times n \times k)$$

— **Phase de construction** : $O(T \times m \times n \times k)$

— **Phase d'évaluation** : $O(T \times m \times n)$

— **Phase de mise à jour** : $O(T \times n \times k)$

Le terme dominant est donc $O(T \times m \times n \times k)$.

6.5.2 Application numérique aux scénarios

Table 6.6 Complexité comparative des deux scénarios

Paramètre	**Symbole**	**Scénario 1 (40 tâches)**	**Scénario 2 (84 tâches)**
Nombre de tâches	n	40	84
Nombre de slots	k	84	84
Nombre de fourmis	m	30	50
Nombre d'itérations	T	150	300
Complexité théorique	$T \times m \times n \times k$	$150 \times 30 \times 40 \times 84$ $= 15,120,000$ opérations	$300 \times 50 \times 84 \times 84$ $= 105,840,000$ opérations
Temps d'exécution moyen	-	15.2 secondes	127.8 secondes
Complexité par seconde	-	$\approx 995,000$ op/s	$\approx 828,000$ op/s

6.5.3 Comparaison avec d'autres approches

Table 6.7 Comparaison des complexités algorithmiques

Algorithme	**Complexité**	**Avantages/Inconvénients**
ACO (notre approche)	$O(T \times m \times n \times k)$	- Exploration intelligente de l'espace - Équilibre exploration/exploitation - Adapté aux problèmes NP-difficiles
Recherche exhaustive	$O(k^n)$	- Garantit l'optimum global - Impossible pour $n > 15$ - $84^{40} \approx 10^{77}$ opérations
Algorithmes génétiques	$O(T \times P \times n)$ (P = taille population)	- Exploration parallèle - Convergence parfois lente
Programmation linéaire	$O(n^3 \times k^3)$	- Solution exacte - Très gourmand en mémoire
Heuristiques gloutonnes	$O(n \times k)$	- Très rapide - Risque d'optimums locaux - Qualité solution variable

6.5.4 Analyse de scalabilité

La figure suivante illustre l'évolution du temps d'exécution en fonction de la taille du problème : $n^2; ^2)^2;$

Figure 6.2 Scalabilité de l'algorithme ACO

Observation : La croissance du temps d'exécution suit approximativement une loi quadratique $O(n^2)$, ce qui correspond à la complexité théorique $O(n \times k)$ avec k constant (84 slots). Pour n allant jusqu'à 100 tâches, l'algorithme reste utilisable en temps réel (moins de 3 minutes).

6.6 Exécution et paramètres

6.6.1 Paramètres ACO utilisés

Les paramètres ont été optimisés pour le problème de timetabling universitaire.

Table 6.8 Paramètres de l'algorithme ACO

Paramètre	Valeur	Description
α	1	Influence de la phéromone (exploitation)
β	5	Influence de l'heuristique (exploration)
ρ	0.01	Taux d'évaporation des phéromones
Q	100	Constante de dépôt de phéromone
τ_{max}	6	Borne supérieure des phéromones (MMAS)
τ_{min}	0.01	Borne inférieure des phéromones (MMAS)
m	30	Nombre de fourmis par itération
w_1	1000	Poids des conflits H1 (prioritaire)
w_2	1	Poids de l'équilibre de charge
w_3	0.1	Poids du gaspillage de capacité

6.6.2 Critères d'arrêt

L'algorithme s'arrête lorsque :

— Le nombre maximum d'itérations est atteint (150 itérations)

— Une solution sans conflit ($f_1 = 0$) est trouvée

— La solution ne s'améliore plus après 50 itérations consécutives

6.7 Résultats obtenus

6.7.1 Statistiques de convergence

L'algorithme converge rapidement vers une solution optimale. Les résultats montrent :

Table 6.9 Statistiques de convergence

Métrique	Valeur	Interprétation
Itérations totales	150	Maximum configuré
Itérations pour $f_1 = 0$	1	Convergence immédiate
Meilleur coût $f(x)$	81.43	Valeur minimale atteinte
Conflits H1 (f_1)	0	Solution réalisable
Équilibre (f_2)	47.43	Distribution acceptable
Gaspillage (f_3)	340.00	340 places vacantes totales
Temps d'exécution	15.2 sec	Sur machine standard

6.7.2 Analyse de la fonction objectif

La fonction objectif présente les caractéristiques suivantes :

Composante f_1 (conflits) : Nulle dès la première itération, démontrant l'efficacité de l'heuristique pour éviter les affectations conflictuelles.

Composante f_2 (équilibre) : Valeur de 47.43 indiquant une répartition raisonnable des cours entre les salles, proche de l'idéal théorique.

Composante f_3 (gaspillage) : Valeur de 340 correspondant au gaspillage inévitable dû aux contraintes de compatibilité (ex : 27 étudiants dans amphithéâtre de 200 places).

6.7.3 Planning final généré

L'emploi du temps généré respecte toutes les contraintes hard :

— **Aucun conflit étudiant** : Chaque classe n'a qu'un cours par créneau

— **Compatibilité respectée** : Les cours GMM/GEP sont en salles TD, les ENSTP en amphithéâtres

— **Capacités respectées** : Toutes les salles accueillent un nombre d'étudiants à leur capacité

6.7.4 Occupation des salles

La répartition des cours entre les salles montre une bonne utilisation des ressources :

Table 6.10 Répartition des cours par salle

Salle	Cours affectés	Occupation moyenne	Gaspillage moyen
TD-1	5	24.4 étudiants	5.6 places
TD-2	6	23.2 étudiants	6.8 places
TD-3	5	22.8 étudiants	7.2 places
TD-4	6	23.7 étudiants	6.3 places
SALLE-60	6	54.2 étudiants	5.8 places
AMPHI-100	6	68.3 étudiants	31.7 places
AMPHI-200	6	71.7 étudiants	128.3 places

6.7.5 Interprétation

Les résultats démontrent que :

1. L'ACO trouve efficacement des solutions sans conflits pour des problèmes sous-contraints (47.6% d'occupation)
2. L'heuristique combinée (capacité + conflits + charge) guide efficacement la construction des solutions
3. Le gaspillage reste acceptable malgré les contraintes de compatibilité strictes
4. L'algorithme converge rapidement, ce qui le rend adapté à des ajustements en temps réel

6.8 Export et visualisation

6.8.1 Fichiers CSV générés

Le système génère trois fichiers CSV principaux :

Table 6.11 Fichiers d'export générés

Fichier	Contenu
`emploi_du_temps.csv`	Planning complet avec toutes les affectations
`pheromone_matrix.csv`	Matrice finale des phéromones pour analyse
`repartition_charge.csv`	Statistiques d'occupation par salle

6.8.2 Format des résultats

Les fichiers CSV suivent un format standard permettant une analyse facile avec divers outils :

Format emploi du temps : Chaque ligne contient : Jour, Période, Salle, Capacité, Classe, Matière, Effectif, Slot_ID, Tâche_ID.

Format phéromones : Matrice $n \times k$ avec les valeurs de phéromones finales pour chaque paire (tâche, slot).

Format répartition : Statistiques agrégées par salle : nombre de cours, occupation moyenne, gaspillage moyen.

Listing 6.1 Exemple de ligne d'emploi du temps généré

```
Jour,P riode ,Salle,Capacite,Classe,Matiere,Effectif,Slot_ID,Tache_ID
L,Matin,TD-1,30,GMM-I1,Alg bre Lin aire I,27,TD-1_L_Matin,GMM-I1_Alg bre_Lin aire_I
L,Aprem,AMPHI-100,100,ENSTP-L1,Math matiques I,75,AMPHI-100_L_Aprem,ENSTP-L1_Math matiques_I
```

6.9 Discussion et perspectives

6.9.1 Limites et défis identifiés

Complexité computationnelle : Bien que polynomiale ($O(T \times m \times n \times k)$), l'algorithme peut devenir coûteux pour de très grandes instances (plusieurs centaines de tâches). Cependant, pour les tailles typiques de campus universitaires (50-200 cours), les performances restent acceptables.

Gaspillage de capacité : L'algorithme privilégie l'évitement des conflits au détriment de l'optimisation de l'occupation. Ceci est dû à la pondération élevée de w_1 (1000) par rapport à w_3 (0.1).

Sensibilité aux paramètres : Les performances de l'ACO dépendent du réglage des paramètres (α, β, ρ). Une procédure d'auto-ajustement pourrait améliorer la robustesse.

Contraintes pédagogiques avancées : Notre modèle n'intègre pas certaines contraintes réalistes comme :

- Les préférences horaires des enseignants
- Les contraintes de suite logique entre cours
- Les besoins en équipements spécialisés
- Les distances entre salles pour les déplacements

6.9.2 Perspectives d'amélioration

Hybridation avec d'autres métaheuristiques :

- Combiner l'ACO avec une recherche locale (Local Search) pour améliorer l'exploitation
- Intégrer des éléments d'algorithmes génétiques pour diversifier la recherche
- Utiliser le recuit simulé pour gérer l'acceptation de solutions temporairement moins bonnes

Optimisation multi-objectif :

— Transformer le problème en optimisation multi-objectif avec Pareto-optimalité

— Utiliser des versions multi-colonies de l'ACO pour explorer différents compromis

— Implémenter des mécanismes de partage de niche pour maintenir la diversité

Parallélisation :

— Exploiter le parallélisme naturel des fourmis (chaque fourmi peut être exécutée sur un cœur différent)

— Implémenter une version MPI ou CUDA pour les très grandes instances

— Utiliser des techniques de décomposition de problèmes (Divide and Conquer)

Interface utilisateur avancée :

— Développer une interface web interactive pour les planificateurs

— Intégrer des visualisations en temps réel de la convergence

— Permettre des ajustements manuels avec ré-optimisation incrémentale

6.9.3 Applicabilité à d'autres contextes

L'approche développée est générique et peut être adaptée à divers problèmes d'ordonnancement :

Table 6.12 Applications potentielles de l'approche

Domaine	**Adaptations nécessaires**
Planning hospitalier	Contraintes de spécialité, urgences, rotations de personnel
Ordonnancement industriel	Temps de setup, contraintes de maintenance, livraisons
Gestion de projets	Dépendances entre tâches, ressources partagées, deadlines
Transport scolaire	Capacités des bus, trajets optimaux, horaires élèves

6.9.4 Conclusion du chapitre

Ce chapitre a présenté l'implémentation complète de l'algorithme ACO appliqué au problème de timetabling du campus de Sogbo-Aliho. L'analyse détaillée montre que :

1. L'architecture modulaire permet une maintenance et une évolution faciles du code
2. La complexité algorithmique ($O(T \times m \times n \times k)$) est polynomiale et reste pratique pour les instances de taille réelle
3. Les résultats obtenus démontrent l'efficacité de l'approche pour trouver des solutions réalisables rapidement
4. L'algorithme offre un bon compromis entre qualité de solution et temps de calcul
5. L'approche est extensible et peut intégrer des contraintes supplémentaires

Les perspectives d'amélioration identifiées ouvrent la voie à des développements futurs qui pourraient encore accroître l'efficacité et l'applicabilité de la méthode à des problèmes d'optimisation combinatoire plus larges.

Bibliographie

[1] M. Dorigo, *Optimization, Learning and Natural Algorithms* (en italien), Thèse de doctorat, Dipartimento di Elettronica, Politecnico di Milano, Italie, 1992.

[2] M. Dorigo, V. Maniezzo, et A. Colorni, "Ant System : Optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics–Part B*, vol. 26, no. 1, pp. 29-41, 1996.

[3] M. Dorigo et T. Stützle, *Ant Colony Optimization*, MIT Press, Cambridge, MA, 2004.

[4] T. Stützle et H. H. Hoos, "MAX-MIN Ant System," *Future Generation Computer Systems*, vol. 16, no. 8, pp. 889-914, 2000.

[5] M. Dorigo, M. Birattari, et T. Stützle, "Ant Colony Optimization," *IEEE Computational Intelligence Magazine*, vol. 1, no. 4, pp. 28-39, novembre 2006.

[6] K. Socha, J. Knowles, et M. Sampels, "A MAX-MIN Ant System for the university course timetabling problem," in *Proceedings of the 3rd International Workshop on Ant Algorithms (ANTS 2002)*, Springer LNCS vol. 2463, pp. 1-13, 2002.

[7] M. Matijas et M. Pavlić, "Ant colony optimization for the university timetabling problem," *International Journal of Engineering Research and Applications*, vol. 1, no. 4, pp. 1990-1997, 2010.

[8] I. Alaya, C. Solnon, et K. Ghédira, "Optimisation par colonies de fourmis pour le problème du sac à dos multidimensionnel," *Revue des Sciences et Technologies de l'Information - Série TSI : Technique et Science Informatiques*, vol. 26, no. 3-4, pp. 371-390, 2007.

[9] E. O. Wilson, "Chemical communication among workers of the fire ant *Solenopsis saevissima* (Fr. Smith)," *Animal Behaviour*, vol. 10, no. 1-2, pp. 134-164, 1962.

[10] S. Goss, S. Aron, J. L. Deneubourg, et J. M. Pasteels, "Self-organized shortcuts in the Argentine ant," *Naturwissenschaften*, vol. 76, pp. 579-581, 1989.

[11] E. D. Morgan, "Trail pheromones of ants," *Physiological Entomology*, vol. 34, no. 1, pp. 1-17, 2009.

[12] J. L. Deneubourg, S. Aron, S. Goss, et J. M. Pasteels, "The self-organizing exploratory pattern of the Argentine ant," *Journal of Insect Behavior*, vol. 3, no. 2, pp. 159-168, 1990.

[13] E. Bonabeau, M. Dorigo, et G. Theraulaz, "Inspiration for optimization from social insect behaviour," *Nature*, vol. 406, pp. 39-42, juillet 2000.

[14] M. Mazlan, M. Makhtar, A. F. K. Ahmad Khairi, et M. A. Mohamed, "University course timetabling model using ant colony optimization algorithm approach," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 13, no. 1, pp. 72-76, janvier 2019.

[15] S. Aslan et C. Aci, "Solving University Course Timetabling Problem Using Ant Colony Optimization : An Example of Mersin University Engineering Faculty," in *Proceedings of International Conference on Advanced Technologies, Computer Engineering and Science (ICATCES'18)*, Safranbolu, Turkey, pp. 154-157, mai 2018.

[16] R. Lewis, "A survey of metaheuristic-based techniques for University Timetabling problems," *OR Spectrum*, vol. 30, no. 1, pp. 167-190, 2008.