ECOLE NATIONALE
SUPÉRIEURE
D'INFORMATIQUE

الجمهوريـة الجزائريـة الديمقراطيـة الشعبيـة
وزارة التعليم العالي والبحث العلمي
People's Democratic République of Algeria
Ministry of Higher Education and Scientific Research

École Nationale Supérieure d'Informatique (ESI)
Département d'Informatique
2ème année, Spécialité SIQ – Groupe 02

Optimisation Combinatoire

Métaheuristique manipulant une population

Réalisé par :

- Bayadh Hadjer
- Rahal Nour Elhouda
- Medfouni Khitem
- Nakib Ibtihel
- Yazi Lynda Mellissa
- Benmachiche Khaled

Encadré par :
Amine KECHID

Année Académique
2024 – 2025

1. Introduction

Dans le cadre de l'optimisation combinatoire, les métaheuristiques basées sur des populations représentent une catégorie d'algorithmes qui exploitent l'interaction entre plusieurs solutions pour explorer efficacement l'espace de recherche. Contrairement aux méthodes de recherche locale, qui manipulent une seule solution à la fois, ces techniques travaillent avec un ensemble de solutions (une population), favorisant ainsi une exploration plus diversifiée et une meilleure capacité à éviter les minima locaux.

Parmi les principales méthodes à population, on distingue notamment :

Les Algorithmes Génétiques (AG), inspirés des mécanismes de l'évolution naturelle tels que la sélection, le croisement et la mutation.

La Méthode de la Colonie de Fourmis (ACO - Ant Colony Optimization), inspirée du comportement collectif des fourmis pour trouver des chemins optimaux.

2. Algorithmes Génétiques (AG)

Les Algorithmes Génétiques (AG) sont des algorithmes itératifs conçus pour optimiser une fonction appelée fonction d'adaptation (ou fonction objectif), qui associe un coût ou une qualité à chaque solution candidate.

À chaque itération, l'AG prend en entrée une population de solutions et fait évoluer cette population pour produire une nouvelle génération, en passant par des populations intermédiaires.

L'évolution de la population s'effectue à travers quatre opérations principales :

Sélection : Une population intermédiaire est construite en sélectionnant des individus selon un critère donné (sélection aléatoire, par rang, par tournoi, etc.). Le but est de privilégier les solutions de meilleure qualité tout en conservant une certaine diversité. La taille de la population intermédiaire peut être différente de la taille de la population initiale.

Croisement : Des couples d'individus sont sélectionnés dans la population intermédiaire pour produire de nouveaux descendants. Le croisement est effectué avec une probabilité Pc selon une méthode donnée (croisement à un point, à deux points, uniforme, etc.), afin de combiner les caractéristiques des parents.

Mutation : Chaque descendant peut subir une mutation avec une probabilité Pm. La mutation consiste à modifier aléatoirement un ou plusieurs gènes de l'individu, dans le but d'introduire de la diversité et d'éviter le piégeage dans des solutions locales.

Mise à jour de la population : Après les opérations de croisement et mutation, une nouvelle génération est formée en respectant la taille initiale de la population N selon un mécanisme de sélection.

Pour implémenter un Algorithme Génétique, il est nécessaire de définir plusieurs paramètres fondamentaux :

- La taille de la population initiale N.
- La taille de la population intermédiaire N'.
- La probabilité de croisement P_c.
- La probabilité de mutation P_m.
- Le critère d'arrêt.

Dans le cadre de ce travail, nous avons implémenté deux variantes des algorithmes génétiques une approche classique (ou naïve), basée uniquement sur les mécanismes standards de l'AG , et une approche à deux phases, qui combine l'algorithme génétique avec une recherche locale afin d'améliorer la qualité des solutions.

Nous présentons ci-dessous le principe de chacune de ces approches.

2.1. Approche classique (naïve)

Dans cette approche, nous avons exploré deux variantes différentes de l'algorithme génétique, qui se distinguent principalement par la manière dont les solutions sont générées et évaluées :

Première variante : Autorisation de solutions conflictuelles

Dans cette première version, la génération de la population initiale se fait de manière totalement aléatoire, sans contrainte particulière, ce qui peut entraîner la présence de solutions comportant des conflits.

La fonction d'adaptation utilisée combine deux objectifs :

- Minimiser le nombre de conflits dans une solution (pénalisé fortement).
- Réduire le nombre de couleurs utilisées.

Les différentes étapes de l'algorithme sont définies comme suit :

- **Sélection :** Nous utilisons la sélection par rang : les individus sont triés selon leur valeur de fonction d'adaptation, et les N' meilleurs sont choisis pour former la population intermédiaire.
- **Croisement :** Réalisé selon un schéma à un point : un point de coupure est choisi aléatoirement et deux parents échangent leurs segments pour produire deux descendants, sans vérification des conflits générés.

- **Mutation :** La mutation agit sur un seul gène choisi aléatoirement dans une solution, également sans contrôle des conflits pouvant être introduits.
- **Mise à jour de la population :** Une stratégie élitiste est adoptée : les meilleurs individus sont conservés afin d'assurer une amélioration progressive.
- **Critères d'arrêt :** L'algorithme s'arrête soit lorsqu'un nombre maximal d'itérations est atteint, soit lorsqu'une stagnation du meilleur résultat est observée pendant un certain nombre d'itérations consécutives.

La mise à jour de la meilleure solution trouvée est réalisée à chaque génération : une solution valide (sans conflit) qui améliore le nombre de couleurs est immédiatement adoptée. Le nombre de couleurs initial est fixé à une borne supérieure déterminée au départ.

Deuxième variante : Solutions strictement réalisables (sans conflit)

Dans cette seconde version, l'objectif est de limiter l'espace de recherche uniquement aux solutions réalisables, c'est-à-dire aux colorations sans conflit dès leur création.

Les spécificités de cette approche sont :

- **Population initiale :** Les solutions initiales sont générées aléatoirement, mais de façon à garantir l'absence totale de conflits.
- **Fonction d'adaptation :** Elle ne prend en compte que le nombre de couleurs utilisées, sans intégrer de pénalisation pour des conflits (puisque ceux-ci sont inexistants).
- **Croisement :** Lors du croisement de deux parents, plusieurs points de coupure sont testés pour tenter de produire des descendants sans conflit. Si aucun point ne permet d'obtenir une solution valide, la paire de parents est abandonnée.
- **Mutation :** Lorsqu'une mutation est tentée sur un sommet, elle n'est acceptée que si elle n'introduit pas de conflit. Si ce n'est pas possible, un autre sommet est choisi. Si aucune mutation sans conflit n'est réalisable pour l'individu, la mutation est abandonnée pour cet individu.
- **Mise à jour de la population :** Une stratégie élitiste est également utilisée pour conserver les meilleures solutions en termes de nombre de couleurs.

Cette approche permet de restreindre l'exploration à l'ensemble des solutions réalisables, au prix d'une complexification des opérations de croisement et mutation.

2.2. Approche à deux phases :

Pourquoi une approche à deux phases ?

Les algorithmes génétiques traditionnels pour la coloration de graphes tentent souvent d'optimiser simultanément deux objectifs contradictoires :

1. Minimiser le nombre de conflits (sommets adjacents de même couleur)
2. Minimiser le nombre total de couleurs utilisées

Cette double optimisation simultanée pose problème car :

- Pour réduire les conflits, l'algorithme a tendance à introduire de nouvelles couleurs
- Pour réduire le nombre de couleurs, l'algorithme risque d'augmenter les conflits

Notre approche à deux phases sépare clairement ces deux objectifs :

- **Phase 1** : Trouver une coloration valide (sans conflits) sans se préoccuper du nombre de couleurs
- **Phase 2** : Minimiser le nombre de couleurs tout en maintenant une coloration valide

Description de l'algorithme

Phase 1 : Recherche d'une coloration valide

Fonction d'adaptation (fitness) En phase 1, la fonction d'adaptation est centrée uniquement sur la minimisation des conflits :

```
fitness = conflicts * 1000
```

Le facteur 1000 assure une forte pénalisation des conflits.

Opérateurs génétiques spécialisés

1. **Sélection par tournoi** : Sélectionne les individus ayant le moins de conflits
2. **Croisement à un point** : Combine deux solutions parentales en échangeant une partie de leurs gènes
3. **Mutation intelligente** :
    - Identifie les sommets impliqués dans des conflits
    - Pour ces sommets, tente de choisir une couleur non utilisée par les voisins
    - Si nécessaire, introduit une nouvelle couleur pour résoudre le conflit
4. **Recherche locale** :
    - Pour chaque sommet en conflit, identifie les couleurs utilisées par ses voisins
    - Attribue une couleur non conflictuelle au sommet, en préférant les couleurs déjà existantes

La phase 1 se termine dès qu'une coloration valide est trouvée, c'est-à-dire sans aucun conflit.

Phase 2 : Minimisation du nombre de couleurs

Initialisation La population initiale de la phase 2 est créée à partir de la meilleure solution trouvée en phase 1, avec de légères mutations pour assurer la diversité.

Fonction d'adaptation En phase 2, la fonction d'adaptation change pour se concentrer sur la minimisation des couleurs :

```
if conflicts > 0:
    fitness = conflicts * 10000  # Pénaliser fortement l'apparition de conflits
else:
    fitness = num_colors_used    # Minimiser le nombre de couleurs
```

Opérateurs génétiques spécialisés

1. **Mutation orientée vers la réduction des couleurs** :
   - Identifie les couleurs peu fréquentes
   - Tente de réattribuer à ces sommets des couleurs plus fréquemment utilisées
   - Vérifie toujours qu'aucun conflit n'est créé avec les voisins
2. **Recherche locale pour la consolidation des couleurs** :
   - Trie les couleurs par fréquence d'utilisation
   - Essaie de réattribuer les couleurs les moins utilisées vers les plus utilisées
   - Maintient toujours la validité de la coloration
3. **Optimisation des couleurs** :
   - Réindexe les couleurs pour qu'elles soient consécutives
   - Assure que les couleurs commencent à 0 et sont utilisées dans l'ordre croissant

Caractéristiques importantes de l'algorithme

1. **Élitisme** : Le meilleur individu est toujours conservé d'une génération à l'autre
2. **Traitement différent selon la phase** :
   - Phase 1 : Favorise la diversité des solutions pour résoudre les conflits
   - Phase 2 : Favorise la convergence vers des solutions avec moins de couleurs
3. **Équilibre entre exploration et exploitation** :
   - Les opérateurs génétiques (croisement, mutation) assurent l'exploration globale
   - La recherche locale assure l'exploitation locale et l'amélioration des solutions
4. **Adaptation dynamique** :
   - La mutation s'adapte aux caractéristiques des individus et aux problèmes spécifiques
   - Plus agressive en phase 1, plus conservatrice en phase 2

2.3. Comparaison entre les deux approches:

Graphe	Méthode	K trouvé	K optimal	Temps (s)
dsjc250.5	Approche 1	100	28	449.37
	Approche 2	34	28	965.57
r250.5	Approche 1	116	65	218.33
	Approche 2	69	65	1018.78

Table 1: Comparaison des performances des deux approches AG.

3. Colonie de Fourmis (ACO)

Les algorithmes d'optimisation par colonies de fourmis (ACO - Ant Colony Optimization) s'inspirent du comportement des fourmis dans la nature, en particulier leur capacité à résoudre des problèmes complexes de manière collective. En effet, chaque fourmi effectue des tâches simples, mais c'est la coopération de l'ensemble des fourmis qui permet de trouver des solutions efficaces. De manière similaire, ACO permet de résoudre des problèmes d'optimisation en utilisant des solutions simples mais collaboratives.

3.1. Principe général

Inspirations des fourmis

Les fourmis utilisent un système de phéromones pour communiquer entre elles. Lorsqu'une fourmi trouve une solution intéressante, comme un chemin vers une source de nourriture, elle laisse une trace chimique (phéromone) sur le chemin emprunté. D'autres fourmis détectent cette phéromone et ont plus de chances de suivre le même chemin, renforçant ainsi cette solution. Ce processus de renforcement et de rétroaction entre les fourmis permet à l'ensemble de la colonie de converger progressivement vers une solution optimale.

L'algorithme ACO reprend ce principe en utilisant des phéromones numériques pour guider la recherche de solutions. Les solutions de chaque itération sont stockées sous forme de "traces" numériques, et les solutions qui sont plus efficaces se voient attribuer plus de poids dans les itérations suivantes, attirant ainsi plus d'« exploration » vers ces solutions favorables.

Particularités de l'ACO

L'un des aspects les plus intéressants de l'ACO est son parallélisme naturel. Contrairement à d'autres méthodes d'optimisation qui cherchent à améliorer une seule solution à la fois, l'ACO permet à plusieurs "agents" (les fourmis) de travailler simultanément sur différentes parties du problème. Ces agents peuvent explorer différentes parties de l'espace de solution, ce qui rend l'algorithme particulièrement efficace pour résoudre des problèmes complexes, où les solutions ne sont pas évidentes ou sont difficiles à trouver.

Paramètres dans l'ACO

L'algorithme ACO comporte plusieurs paramètres clés qu'il est important de fixer correctement pour obtenir de bonnes performances :

- **Le taux d'évaporation des phéromones :** Ce paramètre détermine la vitesse à laquelle les phéromones disparaissent au fil du temps. Si l'évaporation est trop rapide, les solutions sub-optimales peuvent rester trop longtemps en compétition avec les bonnes solutions. Si elle est trop lente, l'algorithme risque de converger trop tôt vers une solution non optimale.
- **Le facteur de renforcement des phéromones :** Ce paramètre indique dans quelle mesure les fourmis renforcent les traces de phéromones sur un chemin qu'elles ont emprunté. Un facteur trop élevé peut entraîner une convergence prématurée vers une solution locale, tandis qu'un facteur trop bas ralentit l'apprentissage de la colonie.
- **Le nombre de fourmis :** Ce paramètre représente le nombre d'agents qui exploreront simultanément l'espace de solution. Un nombre trop faible peut rendre l'algorithme moins efficace, tandis qu'un nombre trop élevé peut entraîner des calculs plus coûteux sans améliorer significativement les résultats

3.2. Présentation de l'algorithme 1

L'algorithme que nous allons décrire est une approche de colonie de fourmis pour résoudre le problème de coloration de graphes. Il combine plusieurs éléments, notamment l'initialisation des phéromones, la construction de solutions guidée par les phéromones et l'heuristique, et une mise à jour des phéromones après chaque itération.

Initialisation des phéromones

L'algorithme commence par initialiser les traces de phéromones pour chaque arête du graphe. Une phéromone est une valeur numérique associée à chaque arête, et elle est utilisée pour guider la construction des solutions. Les phéromones sont initialisées à une valeur fixe τ_0, dans notre cas c'est 1.0.

Construction de la solution

Chaque fourmi construit une solution en affectant des couleurs aux sommets du graphe, en respectant les contraintes de coloration. L'affectation est guidée par les phéromones et par une heuristique, qui prend en compte les conflits de couleur (sommet adjacent ayant la même couleur). Pour cela, les étapes suivantes sont suivies :

- **Calcul du degré de saturation (DSATUR) :** Cette heuristique permet de colorier en priorité les sommets les plus contraints (celles avec le plus grand nombre de couleurs voisines différentes) pour minimiser le nombre de couleurs. Le degré de saturation est calculé pour chaque sommet à chaque étape de construction de la solution.

- **Exploration via les phéromones :** Plus une couleur est bonne (réduit les conflits), plus elle est renforcée par l'ajout de phéromones. Les phéromones guident les fourmis vers des solutions efficaces en favorisant les arêtes ayant des traces de phéromones fortes.
- **Exploitation via l'heuristique :** L'heuristique favorise l'utilisation de couleurs moins élevées, de manière à éviter de créer de nouvelles couleurs inutiles. Elle est calculée en fonction du nombre de conflits qu'une couleur pourrait générer. Moins il y a de conflits, plus la couleur sera favorisée.

Mécanisme de recherche locale

Après la construction de la solution par une fourmi, une recherche locale est appliquée avec une probabilité de 30% pour améliorer la solution. Cette recherche locale fonctionne comme suit:

- Identification des sommets en conflit (ayant la même couleur que leurs voisins)
- Sélection aléatoire d'un de ces sommets
- Tentative de réaffectation d'une couleur qui n'est pas utilisée par ses voisins
- Acceptation du changement si le nombre de conflits diminue ou reste stable avec un nombre de couleurs inférieur ou égal

Cette recherche locale permet d'affiner les solutions construites par les fourmis et d'explorer plus efficacement l'espace des solutions.

Mise à jour des phéromones

Après chaque itération, les phéromones sont mises à jour :

- Une partie des phéromones s'évapore selon la formule:

  $$\tau_{uv} = (1 - \rho)\tau_{uv}$$

  où ρ est le taux d'évaporation (paramètre `rho` dans le code, fixé à 0.1 par défaut).
- Les meilleures solutions déposent des phéromones supplémentaires sur les arêtes qu'elles ont empruntées. La quantité de phéromones déposée est calculée selon la formule:

  $$qualit = \begin{cases} \frac{2.0}{nb\_couleurs} & \text{si } conflits = 0 \\ \frac{1.0}{10 conflits nb\_couleurs} & \text{sinon} \end{cases}$$

  Cette formule favorise les solutions avec moins de conflits et moins de couleurs.

Mécanisme d'élitisme

Pour accélérer la convergence vers de bonnes solutions, un mécanisme d'élitisme est implémenté. Après chaque itération, les solutions sont triées par nombre de conflits et de couleurs, et seules les 30% meilleures solutions (élites) sont utilisées pour la mise à jour des phéromones. Cela permet de concentrer l'exploration sur les régions les plus prometteuses de l'espace de recherche.

Paramètres Alpha et Beta

Les paramètres α et β sont cruciaux pour contrôler l'équilibre entre exploration et exploitation dans l'algorithme :

- α contrôle l'importance des phéromones. Une valeur élevée favorise les solutions précédentes avec des traces de phéromones fortes.
- β contrôle l'importance de l'heuristique. Une valeur élevée favorise les couleurs avec moins de conflits.

Ces paramètres influencent la probabilité qu'une fourmi choisisse une couleur en fonction des phéromones et de l'heuristique selon la formule:

$$\text{probabilité} = \frac{(\tau^{\alpha})(\eta^{\beta})}{\sum_{j \in \text{couleurs disponibles}} (\tau_j^{\alpha})(\eta_j^{\beta})}$$

où τ représente la somme des phéromones sur les arêtes reliant le sommet courant aux sommets déjà colorés, et η est la valeur heuristique calculée comme l'inverse du nombre de conflits plus un:

$$\eta = \begin{cases} \frac{1.0}{1+conflits} & \text{si } conflits > 0 \\ 2.0 & \text{sinon} \end{cases}$$

Choix probabiliste

À chaque étape, chaque couleur possible est choisie avec une probabilité basée sur les phéromones et l'heuristique. Cela permet à l'algorithme de s'aventurer dans de nouvelles solutions tout en privilégiant les couleurs qui ont montré de bons résultats.

Le choix de la couleur est effectué avec la méthode `random.choices()`, où les couleurs sont choisies en fonction des probabilités calculées à partir des phéromones et de l'heuristique:

couleur_choisie = random.choices(couleurs, weights=probas_norm)[0]

Cette méthode permet une exploration équilibrée des différentes couleurs, avec un biais pour les plus prometteuses.

Critères d'arrêt

L'algorithme s'arrête lorsqu'un des critères suivants est atteint :

- Le nombre de conflits atteint zéro, ce qui signifie qu'une solution optimale a été trouvée.

- Un nombre maximal d'itérations est atteint, ce qui permet de limiter le temps d'exécution de l'algorithme.

Les solutions les plus prometteuses sont conservées à chaque itération, et l'algorithme cherche à minimiser les conflits tout en utilisant un nombre minimal de couleurs.

L'historique des solutions est également conservé pour permettre une analyse plus approfondie de la convergence de l'algorithme. Plusieurs instances de graphes DIMACS standard pourraient être utilisées pour comparer les performances:

- Graphes de petite taille: pour valider la correctitude de l'algorithme
- Graphes de taille moyenne: pour évaluer les performances dans des conditions réalistes
- Graphes de grande taille et densité élevée: pour tester les limites de l'algorithme

Des ajustements des paramètres (α, β, ρ, nombre de fourmis) peuvent être nécessaires selon les caractéristiques du graphe à traiter.

3.3. Présentation de l'algorithme 2

Représentation et initialisation

L'algorithme utilise une représentation orientée objet où :

- Le graphe est représenté via la bibliothèque NetworkX
- Les phéromones sont stockées dans une structure de dictionnaire associant une valeur à chaque paire de nœuds (u, v)
- Les phéromones sont initialisées à une valeur τ_0 (paramètre `initial_pheromone`)

Listes de candidats étendues

Une innovation importante de cette implémentation est l'utilisation de listes de candidats étendues :

- Pour chaque nœud, on maintient une liste de candidats incluant :
  - Les voisins directs (nœuds adjacents)
  - Les voisins de second ordre (nœuds à distance 2)
- Cette approche permet de concentrer la recherche sur les nœuds les plus pertinents, tout en réduisant l'espace de recherche

3.4. Construction des solutions

Sélection du nœud initial

Le processus de construction d'une solution commence par le choix d'un nœud initial :

- La sélection est biaisée vers les nœuds de haut degré
- La probabilité de sélection d'un nœud i est proportionnelle à son degré d_i :

$$P(i) = \frac{d_i}{\sum_{j \in V} d_j} \tag{1}$$

Règle de transition ACS

La sélection du prochain nœud à colorer utilise la règle de transition pseudo-proportionnelle d'ACS :

- Soit q un nombre aléatoire uniforme dans $[0, 1]$ et q_0 un paramètre dans $[0, 1]$
- Si $q \leq q_0$ (exploitation) : choisir le nœud j qui maximise $\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}$
- Si $q > q_0$ (exploration) : choisir le nœud j avec une probabilité :

$$p_{ij} = \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{k \in N_i} \tau_{ik}^{\alpha} \cdot \eta_{ik}^{\beta}} \tag{2}$$

  où N_i est l'ensemble des nœuds candidats non encore colorés

Heuristique dynamique

L'information heuristique η_{ij} combine plusieurs facteurs :

- Le degré de saturation du nœud j (nombre de couleurs différentes parmi ses voisins)
- Le degré du nœud j (nombre de voisins)
- Formellement :

$$\eta_{ij} = (1 + \text{saturation}(j)) \cdot (1 + 0.1 \cdot \text{degré}(j)) \tag{3}$$

Coloration glouton

Une fois l'ordre des nœuds déterminé, la coloration est effectuée de façon gloutonne :

- Chaque nœud reçoit la plus petite couleur (entier non négatif) qui n'est utilisée par aucun de ses voisins
- Ce processus garantit une coloration valide, mais la qualité dépend fortement de l'ordre des nœuds

3.5. Mécanismes de mise à jour des phéromones

L'algorithme implémente deux mécanismes de mise à jour des phéromones :

Mise à jour locale

Lorsqu'une fourmi se déplace du nœud i au nœud j :

$$\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij} + \rho \cdot \tau_0 \tag{4}$$

Cette mise à jour a pour effet de diminuer l'attractivité des arêtes récemment utilisées, favorisant ainsi la diversification de la recherche.

Mise à jour globale

À la fin de chaque itération, les phéromones sont mises à jour en fonction de la meilleure solution trouvée :

$$\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij} + \rho \cdot \Delta\tau_{ij} \tag{5}$$

où :

- $\Delta\tau_{ij} = \frac{1}{\text{nombre\_couleurs}}$ pour les arêtes (i, j) de la meilleure solution
- La qualité de la solution est inversement proportionnelle au nombre de couleurs utilisées

Renforcement des ensembles indépendants

Une caractéristique innovante de l'algorithme est le renforcement des ensembles indépendants :

- Les nœuds de même couleur qui ne sont pas adjacents mais partagent des voisins communs reçoivent un renforcement supplémentaire
- Cela encourage la formation de grands ensembles indépendants, réduisant ainsi le nombre de couleurs nécessaires

3.6. Paramètres d'optimisation

L'algorithme comporte plusieurs paramètres ajustables :

- α : importance des phéromones (valeur par défaut : 2.0)
- β : importance de l'information heuristique (valeur par défaut : 1.0)
- ρ : taux d'évaporation des phéromones (valeur par défaut : 0.4)

- q_0 : paramètre d'équilibre exploration/exploitation (valeur par défaut : 0.5)
- τ_0 : niveau initial de phéromones (valeur par défaut : 0.1)
- Nombre de fourmis : généralement adapté à la taille du graphe
- Nombre maximum d'itérations : critère d'arrêt de l'algorithme

3.7. Validation et analyse des solutions

L'algorithme intègre des mécanismes pour :

- Vérifier la validité de la coloration (absence de conflits)
- Analyser la distribution des couleurs
- Suivre l'évolution des solutions au cours des itérations

3.8. Complexité algorithmique

La complexité temporelle de l'algorithme est de l'ordre de :

$$O(I \cdot A \cdot |V|^2) \tag{6}$$

où I est le nombre d'itérations, A est le nombre de fourmis, et $|V|$ est le nombre de sommets du graphe.

Exemple d'executions dans des banchmarks

Graphe	**Méthode**	**K trouvé**	**K optimal**	**Temps (s)**
dsjc250.5	Algorithme 1	35	28	6.2467
	Algorithme 2	36	28	171.16
r250.5	Algorithme 1	66	66	5.1653
	Algorithme 2	69	66	168.00

Table 2: Comparaison des performances des deux algorithmes sur différents graphes.

3.9. Comparaison et choix de l'algorithme

L'analyse des résultats présentés dans le tableau 2 révèle des différences significatives de performance entre les deux algorithmes. L'algorithme 1 se distingue par sa rapidité d'exécution, avec des temps de calcul environ 30 fois inférieurs à ceux de l'algorithme 2 (6,25 secondes contre 171,16 secondes pour le graphe dsjc250.5, et 5,17 secondes contre 168 secondes pour le graphe r250.5). En termes de qualité des solutions, l'algorithme 1 obtient des résultats légèrement meilleurs que l'algorithme 2, avec un nombre de couleurs plus proche de l'optimal (35 contre 36 pour dsjc250.5, et 66 contre 69 pour r250.5). Pour r250.5, l'algorithme 1 atteint

même la coloration optimale connue. Cette supériorité de l'algorithme 1 peut s'expliquer par son mécanisme de recherche locale et son approche élitiste, qui semblent plus efficaces que la règle de transition ACS et les listes de candidats étendues utilisées par l'algorithme 2. De plus, l'algorithme 1 semble mieux s'adapter aux caractéristiques spécifiques des graphes DIMACS. Pour des applications pratiques où le temps de calcul est une contrainte importante, l'algorithme 1 représente clairement le meilleur choix, offrant un excellent compromis entre qualité de solution et efficacité computationnelle.

3.10. Conclusion

L'algorithme ACO est une méthode puissante pour résoudre le problème de la coloration de graphes, en s'appuyant sur des principes biologiques inspirés des fourmis. Il est particulièrement efficace pour les problèmes combinatoires complexes et peut être adapté à d'autres types de problèmes d'optimisation.

4. Comparaison entre AG et ACO

4.1. Comparaison des métriques de performance AG vs ACO

Performance en termes de qualité de solution

Graphe	AG (Approche 2)	ACO (Algorithme 1)	Optimal
dsjc250.5	34	35	28
r250.5	69	66	65-66

Table 3: Comparaison de la qualité des solutions (nombre de couleurs) entre AG et ACO.

Pour dsjc250.5, l'AG offre une solution légèrement meilleure (34 vs 35 couleurs). Pour r250.5, l'ACO est clairement supérieur et atteint presque (ou exactement) la solution optimale (66 vs 69 couleurs).

Performance en termes de temps d'exécution

Graphe	AG (Approche 2)	ACO (Algorithme 1)
dsjc250.5	965.57 s	6.25 s
r250.5	1018.78 s	5.17 s

Table 4: Comparaison des temps d'exécution entre AG et ACO.

L'ACO est considérablement plus rapide que l'AG (environ 150-200 fois plus rapide).

4.2. Analyse comparative

Forces de l'Algorithme Génétique (Approche 2)

1. **Séparation des objectifs** : L'approche à deux phases sépare clairement l'objectif de minimisation des conflits et celui de minimisation du nombre de couleurs.
2. **Performance sur certains graphes** : Meilleure performance sur le graphe dsjc250.5.
3. **Adaptabilité** : Les opérateurs génétiques spécialisés dans chaque phase permettent une meilleure adaptation aux caractéristiques du problème.

Forces de l'Algorithme ACO (Algorithme 1)

1. **Efficacité computationnelle** : Temps d'exécution exceptionnellement court.
2. **Qualité des solutions** : Atteint l'optimal ou s'en approche très près pour certains graphes.
3. **Mécanisme élitiste** : La sélection des 30% meilleures solutions pour la mise à jour des phéromones accélère la convergence.
4. **Recherche locale intégrée** : Application d'une recherche locale avec une probabilité de 30% pour améliorer les solutions.

Différences fondamentales d'approche

1. **Construction de solution** :
   - AG : Travaille sur une population de solutions complètes qui évoluent
   - ACO : Construction progressive des solutions guidée par les phéromones
2. **Exploitation de l'historique** :
   - AG : Implicite via la sélection et le croisement
   - ACO : Explicite via les traces de phéromones
3. **Mécanisme d'amélioration** :
   - AG : Principalement par croisement et mutation
   - ACO : Principalement par le dépôt de phéromones et la recherche locale

4.3. Critères de sélection d'algorithme

1. **Pour des contraintes de temps** :
   - L'ACO est indiscutablement plus adapté quand le temps d'exécution est critique
2. **Pour une qualité maximale** :

- Dépend du type de graphe
- AG peut être préférable pour certains graphes (comme dsjc250.5)
- ACO pour d'autres (comme r250.5)

3. **Compromis temps/qualité** :
   - L'ACO offre généralement le meilleur compromis, avec des solutions proches de l'optimal en un temps très court

4.4. Conclusion

L'analyse comparative montre que l'ACO (Algorithme 1) présente un avantage significatif en termes d'efficacité computationnelle tout en maintenant une qualité de solution comparable ou supérieure à l'AG. Pour la plupart des applications pratiques de coloration de graphe où le temps est un facteur important, l'ACO serait le choix recommandé. Cependant, l'AG peut encore être utile dans des situations où la qualité de la solution prime absolument sur le temps de calcul, et pour certains types spécifiques de graphes.

5. Comparaison entre les approches à recherche locale et à population

5.1. Méthodes de recherche locale

Les métaheuristiques à recherche locale utilisées incluent le recuit simulé (SA), la recherche tabou (Tabu Search) et la recherche à voisinage variable (VNS) sont caractérisées par une exploration intensive de l'espace de solutions à partir d'une solution initiale donnée, en modifiant progressivement celle-ci via des mouvements locaux.

- **Avantages** :
  - Simplicité d'implémentation et de contrôle.
  - Bonne capacité à affiner une solution existante.
- **Limites** :
  - Risque de stagnation dans des optima locaux.
  - Moins adaptée à l'exploration globale de l'espace de recherche.

5.2. Méthodes à population

Les métaheuristiques à population, notamment les algorithmes génétiques (AG) et l'optimisation par colonies de fourmis (ACO) manipulent un ensemble de solutions en parallèle, favorisant ainsi une exploration plus large de l'espace de recherche.

- **Avantages** :
  - Exploration plus globale, permettant d'éviter les optima locaux.
  - Diversité de solutions maintenue grâce aux opérateurs (mutation, croisement, phéromones).
  - Adaptée à des espaces complexes ou discontinus.
- **Limites** :
  - Plus coûteuse en temps de calcul.
  - Requiert un réglage précis de plusieurs paramètres (taille de population, taux de mutation, intensité de la sélection, etc.).

5.3. Comparaison des résultats expérimentaux

Afin d'évaluer les performances respectives des deux grandes familles de métaheuristiques étudiées, nous effectuons une comparaison entre les meilleures approches de recherche locale — Tabu2 pour la qualité de solution et SA2 pour le temps d'exécution — et la meilleure approche à population, à savoir l'algorithme de colonie de fourmis (ACO – Algorithme 1). La comparaison se fait selon deux critères : le nombre de couleurs utilisées et le temps d'exécution.

Graphe	Méthode	K trouvé	K optimal	Temps (s)
dsjc250.5	SA2	37	28	12.86
	Tabu2	32	28	234.40
	ACO	35	28	6.25
r250.5	SA2	68	65	10.82
	Tabu2	66	65	319.57
	ACO	66	66	5.17

Table 5: Comparaison des performances des trois algorithmes sur deux graphes.

Performance en termes de qualité de solution

Graphe	Tabu2	ACO (Algorithme 1)	Optimal
dsjc250.5	32	35	28
r250.5	66	66	65-66

Table 6: Comparaison de la qualité des solutions (nombre de couleurs) ACO et Tabu2.

Conclusion (qualité de solution) : Tabu2 parvient à générer des solutions légèrement plus proches de l'optimum que l'ACO, notamment sur le graphe `dsjc250.5`. Toutefois, sur `r250.5`, les deux approches donnent des résultats équivalents, montrant que l'ACO reste compétitif en qualité.

Graphe	SA2	ACO (Algorithme 1)
dsjc250.5	12.86 s	6.25 s
r250.5	10.82 s	5.17 s

Table 7: Comparaison des temps d'exécution entre ACO et SA2.

Performance en termes de temps d'exécution

Conclusion (temps d'exécution) : L'algorithme ACO se montre nettement plus rapide que SA2, avec des temps d'exécution réduits de plus de moitié pour les deux graphes testés. Cette efficacité temporelle renforce l'intérêt de l'ACO dans des contextes contraints en ressources ou en temps.

Conclusion

La comparaison montre que l'approche par colonies de fourmis (ACO) offre un excellent compromis entre qualité de solution et efficacité computationnelle. Bien que Tabu2 puisse fournir de meilleures solutions dans certains cas spécifiques, l'ACO reste très compétitif tout en étant significativement plus rapide. Ainsi, pour des applications où le temps de calcul est critique, l'ACO constitue une option privilégiée, tandis que les approches à recherche locale conservent leur intérêt pour maximiser la qualité dans un cadre moins contraint.