Des fourmis pour le problème d'ordonnancement de voitures

Christine Solnon

LIRIS CNRS UMR 5205, Université Lyon I
Nautibus, 43 Bd du 11 novembre, 69622 Villeurbanne cedex, France
christine.solnon@liris.cnrs.fr

Résumé

Le problème d'ordonnancement de voitures consiste à séquencer des voitures sur une chaine de montage en satisfaisant des contraintes liées à la capacité des différents ateliers de montage positionnés le long de la chaine. On décrit dans cet article un algorithme à base de fourmis pour résoudre ce problème, et on propose deux structures phéromonales complémentaires pour cet algorithme : la première vise à apprendre les bonnes sous-séquences de voitures ; la seconde vise à apprendre les voitures « critiques ». On compare expérimentalement ces deux structures phéromonales, qui ont des performances complémentaires, et on montre que leur combinaison améliore les performances de chacune. On compare finalement l'algorithme ACO combinant les deux structures phéromonales à d'autres approches, et on montre qu'il permet de résoudre plus rapidement une majorité d'instances.

Abstract

The car sequencing problem involves scheduling cars along an assembly line while satisfying capacity constraints. In this paper, we describe an Ant Colony Optimization (ACO) algorithm for solving this problem, and we propose two complementary pheromone structures for this algorithm : the first one aims at learning good sub-sequences of cars, whereas the second one aims at learning "critical" cars. We experimentally compare these two pheromone structures, that have complementary performances, and show that their combination improves performances of each of them. We finally compare the ACO algorithm that combines the two pheromone structures with state-of-the-art approaches for this problem, and show that it allows to solve quicker most instances.

1 Introduction

Le problème d'ordonnancement de voitures consiste à séquencer des voitures sur une chaine de montage pour leur installer des options (e.g., toit ouvrant ou climatisation). Chaque option est montée par une station différente conçue pour traiter au plus un certain pourcentage de voitures passant le long de la chaîne. Par conséquent, les voitures demandant une même option doivent être espacées de telle sorte que la capacité des stations ne soit jamais dépassée.

Ce problème est NP-difficile [16]. On peut facilement le formuler en un problème de satisfaction de contraintes (CSP) et il s'agit d'un benchmark classique pour les solveurs de contraintes [3, 13, 30]. La plupart de ces solveurs sont basés sur une approche complète qui explore l'espace de recherche de façon systématique jusqu'à ce qu'une solution soit trouvée, ou le problème soit montré inconsistant. Afin de réduire l'espace de recherche, cette aproche est combinée avec des algorithmes de filtrage qui propagent les contraintes de capacité pour réduire les domaines des variables. En particulier, Régin et Puget ont proposé un algorithme de filtrage dédié à ce type de contraintes [24], permettant ainsi de résoudre plus efficacement certaines instances. Cependant, cet algorithme ne réduit pas toujours suffisamment les domaines des variables, et ne permet pas de résoudre certaines instances en un temps « raisonnable ». Plus récemment, Gravel et al. ont proposé une modélisation linéaire en nombres entiers [15]. Cette approche complète a permis de résoudre à l'optimal certaines instances restées ouvertes jusque là, mais ne permet toujours pas de résoudre d'autres instances.

Ainsi, différentes approches incomplètes ont été proposées, qui n'explorent pas de façon exhaustive l'espace de recherche, mais se dirigent de façon opportuniste en essayant de trouver de bonnes solutions, e.g., la recherche

locale [2, 17, 23, 19, 14, 10], la recherche à grand voisinage [22], IDWalk [20], les algorithmes génétiques [31], ou l'optimisation par colonies de fourmis [25, 15]. Une synthèse des différentes approches, complètes et incomplètes, proposées pour ce problème peut être trouvée dans [27].

1.1 Optimisation par colonies de fourmis

Le principe de base de l'optimisation par colonies de fourmis (Ant Colony Optimization/ACO) [6, 5, 9] est de modéliser le problème à résoudre sous la forme d'un chemin de coût minimal dans un graphe, et d'utiliser des fourmis artificielles pour rechercher des chemins. Le comportement des fourmis artificielles est inspiré de celui des fourmis réelles : (i) elles déposent des traces de phéromone sur les composants du graphe ; (ii) chaque fourmi choisit son chemin en fonction de probabilités qui dépendent des traces précédemment déposées ; (iii) ces traces sont progressivement diminuées par évaporation. Intuitivement, cette communication indirecte via l'environnement —connue sous le nom de stigmergie— fournit une information sur la qualité des composants empruntés afin d'attirer les fourmis et d'intensifier la recherche dans les itérations futures vers les zones correspondantes de l'espace de recherche. Ce mécanisme d'intensification est combiné à un mécanisme de diversification, basé sur la nature aléatoire des décisions prises par les fourmis, qui garantit une bonne exploration de l'espace de recherche. Le compromis entre intensification et diversification peut être obtenu en modifiant les valeurs des paramètres déterminant l'importance de la phéromone.

Les fourmis artificielles ont aussi des capacités supplémentaires qui ne trouvent pas d'équivalent dans la nature. En particulier, elles peuvent généralement mémoriser leurs actions passées et appliquer des procédures d'optimisation, comme la recherche locale, pour améliorer la qualité des chemins construits. Souvent, les traces de phéromone ne sont mises à jour que lorsqu'un chemin complet a été calculé, et non lors de la progression des fourmis, et la quantité de phéromone déposée dépend de la qualité du chemin construit. Enfin, la probabilité pour une fourmi de choisir un composant du graphe dépend généralement non seulement des traces de phéromone, mais aussi d'heuristiques locales dépendantes du problème.

Le premier algorithme à base de fourmis a été proposé par Dorigo en 1992 [4]. Le problème choisi pour ces premières expérimentations est celui du voyageur de commerce, problème qui présente de fortes similitudes avec celui résolu par les fourmis dans la nature. Ce problème a été l'objet de nombreuses études concernant les capacités de résolution des fourmis artificielles [8, 7]. La métaheuristique ACO, décrite dans [6, 5, 9], est une généralisation de ces premiers algorithmes à base de fourmis, et a été appliquée avec succès à de nombreux autres problèmes combinatoires comme par exemple les problèmes d'affectation quadratique [11, 18], de routage de véhicules [1, 12], de satisfaction de contraintes [26], ou de recherche de cliques maximums [28].

1.2 Motivations et plan de l'article

Nous avons proposé dans [25] un premier algorithme ACO dédié aux problèmes de recherche de permutations sous contraintes, dont le but est de rechercher une permutation d'un ensemble de valeurs satisfaisant certaines contraintes. Les performances de cet algorithme ont été notamment illustrées sur le problème d'ordonnancement de voitures. Dans ce premier algorithme ACO pour le problème d'ordonnancement de voitures, la phéromone est déposée sur les couples de voitures séquencées consécutivement afin d'apprendre les sous-séquences de voitures prometteuses.

Dans [14], nous avons ensuite proposé et comparé différentes heuristiques pour résoudre le problème d'ordonnancement de voitures de façon constructive gloutonne. Ces heuristiques visent à favoriser le séquencement des voitures « critiques » (i.e., difficiles à séquencer sans violer de contraintes de capacité) lors de la construction gloutonne. Nous avons montré que cette approche gloutonne très simple permet de résoudre de nombreuses instances très rapidement. Nous avons également montré que ces heuristiques gloutonnes peuvent être facilement intégrées à l'algorithme ACO de [25], améliorant ainsi ses performances.

Ces heuristiques gloutonnes sont très efficaces et facilitent grandement la résolution du problème par les fourmis. Cependant, leur conception demande une bonne connaissance du problème à résoudre. Ainsi, l'objectif principal de cet article est de répondre à la question suivante :

> *Serait-il possible d'utiliser ACO pour identifier ces voitures critiques, et résoudre ainsi le problème d'ordonnancement de voitures sans avoir à intégrer d'heuristiques locales ?*

Afin de répondre à cette question, nous introduisons une nouvelle structure phéromonale, visant à apprendre les voitures critiques, et nous montrons que ce nouvel algorithme ACO est plus efficace que l'approche gloutonne de [14] pour identifier ces voitures critiques. Nous montrons également que cette nouvelle structure phéromonale peut être combinée avec la structure phéromonale proposée dans [25].

L'article est organisé de la façon suivante. On définit en 2 le problème d'ordonnancement de voitures. On rappelle en 3 le principe de base de l'approche gloutonne aléatoire proposée dans [14] (et sur laquelle notre algorithme ACO est basé), et en 4 la structure phéromonale introduite dans

[25] et visant à identifier les bonnes sous-séquences de voitures. On introduit en 5 la nouvelle structure phéromonale visant à identifier les voitures critiques, et on montre en 6 comment combiner ces deux structures phéromonales. On compare expérimentalement les différentes structures phéromonales en 7, et on les compare avec d'autres approches en 8.

2 Le problème d'ordonnancement de voitures

Nous considérons ici le problème d'ordonnancement de voitures introduit dans [3], et dont une description complète peut être trouvée dans [13]. Une variante de ce problème, faisant intervenir, en plus des contraintes de capacité liées aux options, des contraintes liées à la couleur des voitures, a fait l'objet du challenge ROADEF en 2005 [21, 27].

Définition du problème. Une instance du problème d'ordonnancement de voitures est définie par un tuple (C, O, p, q, r) tel que

- $C = \{c_1, .., c_n\}$ est l'ensemble des voitures,
- $O = \{o_1, .., o_m\}$ est l'ensemble des options,
- $p : O \rightarrow \mathbb{N}$ et $q : O \rightarrow \mathbb{N}$ sont deux fonctions qui définissent les contraintes de capacité associées aux options, i.e., pour chaque option $o_i \in O$, chaque sous-séquence de $q(o_i)$ voitures consécutives sur la chaine d'assemblage ne doit pas contenir plus de $p(o_i)$ voitures demandant l'option o_i,
- $r : C \times O \rightarrow \{0, 1\}$ est la fonction qui définit les options requises, i.e., pour chaque voiture $c_i \in C$ et pour chaque option $o_j \in O$, si o_j doit être installée sur c_i, alors $r(c_i, o_j) = 1$ sinon $r(c_i, o_j) = 0$.

Solution d'une instance. Résoudre une instance (C, O, p, q, r) consiste à chercher un ordonnancement des voitures en une séquence satisfaisant toutes les contraintes de capacité. Nous utiliserons les notations suivantes pour manipuler les séquences :

- une *séquence*, notée $\pi =< c_{i_1}, c_{i_2}, \ldots, c_{i_k} >$, est une suite de voitures appartenant à C ;
- l'*ensemble de toutes les séquences* que l'on peut construire à partir d'un ensemble de voitures C est noté Π_C ;
- la *longueur* d'une séquence π, notée $|\pi|$, est le nombre de voitures qu'elle contient ;
- la *concatenation* de 2 séquences π_1 et π_2, noté $\pi_1 \cdot \pi_2$, est la séquence composée des voitures de π_1 suivies des voitures de π_2 ;
- une séquence π_1 est une *sous-séquence* d'une autre séquence π_2, noté $\pi_1 \sqsubseteq \pi_2$, s'il existe deux autres séquences éventuellement vides π_3 et π_4 telles que $\pi_2 = \pi_3 \cdot \pi_1 \cdot \pi_4$;
- le *coût* d'une séquence π est le nombre de contraintes de capacité violées, i.e.,

$$cost(\pi) = \sum_{o_i \in O} \sum_{\substack{\pi' \sqsubseteq \pi \text{ tel que} \\ |\pi'| = q(o_i)}} violation(\pi', o_i)$$

où $violation(\pi', o_i) = 0$ si le nombre de voitures demandant l'option o_i dans π' est inférieur ou égal à $p(o_i)$, et $violation(\pi', o_i) = 1$ sinon.

On peut maintenant définir le processus de résolution d'une instance (C, O, p, q, r) par la recherche de la sequence $\pi \in \Pi_C$ telle que $cost(\pi)$ soit minimal.

Classes de voitures. Deux voitures peuvent demander une même configuration d'options. Ainsi, on définit la classe d'une voiture $c_i \in C$ par l'ensemble des options qu'elle demande, i.e.,

$$classOf(c_i) = \{o_j \in O / r(c_i, o_j) = 1\}$$

et on note $classes(C)$ l'ensemble de toutes les classes de voitures différentes, i.e.,

$$classes(C) = \{classOf(c_i) | c_i \in C\}$$

3 Construction gloutonne aléatoire de séquences

La figure 1 décrit un algorithme glouton aléatoire pour construire une séquence : partant d'une séquence vide, des voitures sont itérativement ajoutées en fin de séquence jusqu'à ce que toutes les voitures aient été séquencées. A chaque itération, l'ensemble des voitures candidates (*cand*) est restreint à l'ensemble des voitures introduisant le plus petit nombre de nouvelles violations (ligne 4). Afin de supprimer les symétries entre solutions, on restreint également l'ensemble des candidats aux voitures appartenant à des classes différentes (ligne 5). Etant donné cet ensemble de voitures candidates, la prochaine voiture c_i est choisie en fonction d'une probabilité de transition p. Cette probabilité peut être définie de différentes façons, et nous proposons dans les sections 4, 5 et 6 trois définitions différentes, basées sur des structures phéromonales différentes.

Dans cette section, on définit la probabilité de transition proportionnellement à une fonction heuristique η qui évalue localement la « difficulté » d'une voiture candidate c_i, i.e.,

$$p(c_i, cand, \pi) = \frac{[\eta(c_i, \pi)]^\beta}{\sum_{c_k \in cand} [\eta(c_k, \pi)]^\beta}$$

où β est un paramètre qui permet de pondérer l'importance de l'heuristique dans la politique de transition : plus β augmente, et plus la politique est gloutonne.

Entrée : une instance (C, O, p, q, r) du problème d'ordonnancement de voitures
une probabilité de transition $p : C \times \mathcal{P}(C) \times \Pi_C \rightarrow]0; 1]$
Sortie : une séquence π contenant chaque voiture de C une fois
1- $\pi \leftarrow <>$
2- **tant que** $\mid \pi \mid \leq \mid C \mid$ **faire**
3- Soit $C - \pi$ l'ensemble des voitures de C qui ne sont pas encore séquencées dans π
4- $cand \leftarrow \{c_k \in C - \pi | \forall c_j \in C - \pi, cost(\pi. < c_k >) \leq cost(\pi. < c_j >)$ et
5- $(classOf(c_k) = classOf(c_j)) \Rightarrow (k \leq j)$ }
6- choisir $c_i \in cand$ en fonction de la probabilité $p(c_i, cand, \pi)$
7- $\pi \leftarrow \pi \cdot < c_i >$
8- **retourner** π

FIG. 1 – Construction aléatoire gloutonne d'une séquence de voitures.

Nous avons introduit et comparé dans [14] cinq définitions différentes pour la fonction heuristique η. Ces définitions sont basées sur les taux d'utilisation des options requises et visent à favoriser le choix de voitures demandant des options dont la demande est importante par rapport à la capacité. La fonction heuristique ayant obtenu les meilleurs résultats en moyenne est définie par la somme des taux d'utilisation des options requises par la voiture, i.e.,

$$\eta(c_i, \pi) = \sum_{o_j \in O} r(c_i, o_j) \cdot \mathrm{utilRate}(o_j, C - \pi)$$

où $\mathrm{utilRate}(o_j, C - \pi)$ est le taux d'utilisation de l'option o_j par rapport à l'ensemble $C - \pi$ des voitures qui ne sont pas encore séquencées dans π. Ce taux d'utilisation est le ratio de voitures de $C - \pi$ demandant l'option o_i par rapport au nombre maximum de voitures dans une séquence de longueur $|C - \pi|$ qui pourraient avoir l'option o_i tout en satisfaisant les contraintes de capacité, i.e.,

$$utilRate(o_i, C - \pi) = \frac{q(o_i) \cdot \sum_{c_j \in C - \pi} r(c_j, o_i)}{p(o_i) \cdot \mid C - \pi \mid}$$

Un taux d'utilisation supérieur à 1 indique que la capacité de la station sera nécessairement excédée ; un taux proche de 0 indique que la demande est très basse au vu de la capacité de la station.

4 Une première structure phéromonale pour identifier les bonnes sous-séquences de voitures

Résoudre une instance (C, O, p, q, r) revient à trouver une permutation de C satisfaisant les contraintes de capacité. Ce problème est facilement modélisable sous la forme de la recherche d'un meilleur chemin hamiltonien dans un graphe complet associant un sommet à chaque voiture. De tels problèmes de recherche de chemins hamiltoniens sont des applications classiques pour la métaheuristique ACO : pour ces problèmes, les fourmis déposent de la phéromone sur les arcs du graphe afin d'apprendre les sous-séquences de sommets prometteuses. Suivant ce principe, nous avons proposé dans [25] un premier algorithme ACO pour le problème d'ordonnancement de voitures dont nous rappelons ici le principe : tout d'abord, la structure phéromonale est initialisée ; ensuite, à chaque cycle, chaque fourmi construit une séquence et les traces de phéromone sont mises à jour ; l'algorithme s'arrête quand une fourmi a trouvé une solution ou quand un nombre maximal de cycle a été atteint.

Structure phéromonale. Les fourmis communiquent en déposant de la phéromone sur les couples de voitures. Etant donné un couple de voitures $(c_i, c_j) \in C \times C$, la quantité de phéromone sur ce couple est notée $\tau_1(c_i, c_j)$ et représente l'expérience passée de la colonie concernant la « désirabilité » de séquencer c_j juste après c_i.

Notons que pour cette première structure phéromonale, nous suivons le principe du *MAX-MIN* Ant System [29], i.e., les traces de phéromone sont bornées entre deux valeurs τ_{min_1} et τ_{max_1} telles que $0 < \tau_{min_1} < \tau_{max_1}$. L'objectif est d'éviter une stagnation prématurée de la recherche, en garantissant que la différence relative entre deux traces ne peut devenir trop importante, et donc que la probabilité de choisir un sommet ne peut devenir trop petite. De plus, les traces de phéromone sont initialisées à la borne maximale τ_{max} au début de la résolution. Par conséquent, lors des premiers cycles de la recherche la différence relative entre deux traces est modérée de sorte que l'exploration est accentuée.

Construction de séquences par les fourmis. A chaque cycle, chaque fourmi construit une séquence selon l'algorithme glouton aléatoire de la figure 1. Pour choisir la prochaine voiture c_i à ajouter à la fin de la séquence en cours de construction π, la probabilité de transition dépend de deux facteurs : un facteur phéromonal qui correspond à l'expérience passée de la colonie concernant le fait d'ajouter c_i à la fin de π, et le facteur heuristique η introduit en 3,

i.e.,
si $\pi =<>$ (i.e., quand on choisit la première voiture) :

$$p(c_i, cand, \pi) = \frac{[\eta(c_i, \pi)]^{\beta_1}}{\sum_{c_k \in cand}[\eta(c_k, \pi)]^{\beta_1}}$$

sinon, si la dernière voiture séquencée dans π est c_j :

$$p(c_i, cand, \pi) = \frac{[\tau_1(c_j, c_i)]^{\alpha_1} \cdot [\eta(c_i, \pi)]^{\beta_1}}{\sum_{c_k \in cand}[\tau_1(c_j, c_k)]^{\alpha_1} \cdot [\eta(c_k, \pi)]^{\beta_1}}$$

où α_1 et β_1 sont des paramètres qui déterminent l'importance respective de la phéromone et de l'heuristique.

Mise-à-jour de la phéromone. Une fois que toutes les fourmis ont construit une séquence, les traces de phéromone sont mises à jour. Elles sont tout d'abord « évaporées » en multipliant chaque trace $\tau_1(c_i, c_j)$ par un taux de persistance ρ_1 tel que $0 \leq \rho_1 \leq 1$.

Ensuite, les meilleures fourmis du cycle déposent une trace de phéromone sur les arcs qu'elles ont empruntés, en quantité inversement proportionnelle au nombre de contraintes violées, i.e., pour chaque séquence π construite durant le cycle, si le coût de π est minimal pour le cycle alors, pour chaque couple de voitures consécutives $< c_j, c_k > \subseteq \pi$, on incrémente $\tau_1(c_j, c_k)$ de $1/cost(\pi)$.

5 Une seconde structure phéromonale pour identifier les voitures critiques

La fonction heuristique η introduite en 3 et utilisée dans la probabilité de transition définie en 4 a pour but de favoriser la sélection de voitures « critiques », i.e., de voitures sur lesquelles doivent être installées des options ayant de forts taux d'utilisation de sorte qu'elles peuvent difficilement être séquencées sans violer des contraintes de capacité. Nous proposons maintenant d'utiliser une deuxième structure phéromonale pour identifier ces voitures critiques en fonction des expériences passées.

Structure phéromonale. Les traces de phéromone sont déposées sur les classes de voitures, une classe de voitures regroupant l'ensemble des voitures demandant un même sous-ensemble d'options (cf section 2). Etant donnée une classe de voitures $cc \in classes(C)$, on note $\tau_2(cc)$ la quantité de phéromone déposée sur elle. Intuitivement, cette quantité de phéromone représente l'expérience passée de la colonie concernant la difficulté de séquencer les voitures de cette classe.

La structure phéromonale décrite ici n'entre pas dans le cadre du *MAX-MIN* Ant System comme pour la première structure phéromonale introduite en 4 : nous n'introduisons qu'une borne minimale τ_{min_2} (garantissant que la probabilité de choisir une voiture ne sera jamais nulle), et pas de borne maximale. De plus, les traces de phéromone sont initialisées à la borne minimale τ_{min_2} au début de la recherche, l'objectif étant d'identifier le plus rapidement possible les classes de voiture critiques.

Construction d'une séquence par une fourmi. Les fourmis construisent incrémentalement des séquences, suivant en cela l'algorithme de la figure 1, mais en considérant une autre définition en ce qui concerne la probabilité de choisir une voiture c_j parmi l'ensemble des voitures candidates *cand*. Cette probabilité dépend maintenant de la quantité de phéromone déposée sur la classe de la voiture c_j, l'objectif étant de favoriser la sélection des voitures les plus critiques :

$$p(c_i, cand, \pi) = \frac{[\tau_2(classOf(c_i))]^{\alpha_2}}{\sum_{c_k \in cand}[\tau_2(classOf(c_k))]^{\alpha_2}}$$

Le paramètre α_2 est introduit pour régler le degré d'influence de la phéromone sur le choix des voitures.

Mise-à-jour de la phéromone. Les fourmis déposent des traces de phéromone pendant la construction des séquences : à chaque fois que plus aucune voiture ne peut être séquencée sans introduire de nouvelles violations de contraintes, une trace de phéromone est ajoutée sur chaque classe de voiture dont au moins une voiture reste à séquencer, indiquant ainsi que les voitures de ces classes devraient être séquencées en priorité. La quantité de phéromone déposée est égale au nombre de nouvelles violations de contraintes introduites par les voitures de la classe. Plus précisément, on modifie l'algorithme de la figure 1 en ajoutant entre les lignes 5 et 6 les trois lignes suivantes :

si $\forall c_i \in cand,\ cost(\pi. < c_i >) > cost(\pi)$ **alors**
  pour chaque classe $cc \in \{classOf(c_i) \mid c_i \in C - \pi\}$:
    $\tau_2(cc) \leftarrow \tau_2(cc) + cost(\pi. < c_i >) - cost(\pi)$
    (où c_i est une voiture de la classe cc)

Notons que ces dépôts phéromonaux ont lieu pendant la construction d'une séquence, et non une fois que toutes les fourmis ont construit une séquence comme cela est souvent le cas dans les algorithmes ACO. En effet, les quantités de phéromone déposées ne dépendent pas de la qualité globale des séquences construites durant le cycle, mais de l'évaluation locale de la voiture par rapport à la séquence en cours de construction. Notons également que toutes les fourmis déposent de la phéromone, et non uniquement les meilleures du cycle.

Enfin, les dépots phéromonaux ayant lieu à chaque construction d'une séquence, l'évaporation a également lieu à la fin de la construction de chaque séquence.

6 Combinaison des deux structures phéromonales

Les deux structures phéromonales introduites en 4 et 5 sont complémentaires : la première vise à identifier des sous-séquences de voitures prometteuses, la seconde vise à identifier des classes de voitures critiques. On peut donc les combiner très facilement. L'algorithme résultant utilise deux structures phéromonales différentes :

- Les fourmis déposent de la phéromone sur les couples de voitures $(c_i, c_j) \in C \times C$, la quantité de phéromone $\tau_1(c_i, c_j)$ représentant l'expérience passée de la colonie concernant le fait de séquencer la voiture c_j juste derrière la voiture c_i. Pour cette structure phéromonale, les quantités de phéromone sont mises à jour à la fin de chaque cycle, une fois que chaque fourmi a construit une séquence, et seules les meilleures fourmis déposent de la phéromone.
- Les fourmis déposent également de la phéromone sur les classes de voiture $cc \in Classes(C)$, la quantité de phéromone $\tau_2(cc)$ représentant l'expérience passée de la colonie concernant la difficulté de séquencer les voitures de cette classe sans violer de contraintes. Pour cette structure phéromonale, la phéromone est déposée par chaque fourmi pendant qu'elle construit une séquence, et la phase d'évaporation a lieu à la fin de chaque construction d'une séquence par une fourmi.

L'algorithme de construction d'une séquence par une fourmi est celui de la figure 1, mais en considérant une autre définition en ce qui concerne la probabilité de choisir une voiture c_i parmi l'ensemble des voitures candidates *cand*. Cette probabilité dépend maintenant des deux structures phéromonales, i.e.,
Si $\pi =<>$ (i.e., quand on choisit la première voiture) :

$$p(c_i, cand, \pi) = \frac{[\tau_2(classOf(c_i))]^{\alpha_2}}{\sum_{c_k \in cand} [\tau_2(classOf(c_k))]^{\alpha_2}}$$

sinon, si la dernière voiture séquencée dans π est c_j :

$$p(c_i, cand, \pi) = \frac{[\tau_1(c_j, c_i)]^{\alpha_1} \cdot [\tau_2(classOf(c_i))]^{\alpha_2}}{\sum_{c_k \in cand} [\tau_1(c_j, c_k)]^{\alpha_1} \cdot [\tau_2(classOf(c_k))]^{\alpha_2}}$$

où α_1 et α_2 sont deux paramètres qui déterminent les poids relatifs des deux facteurs phéromonaux.

7 Comparaison expérimentale des différents algorithmes ACO

Jeux d'essais. Les instances du jeu d'essai généré par Lee [17] sont trivialement résolues par les différents algorithmes ACO que nous venons de décrire (en moins d'un centième de secondes en moyenne). Nous considérons donc ici un nouveau jeu d'essai contenant des instances plus difficiles générées par Perron et Shaw [22]. Les instances de ce jeu d'essai ont toutes $|O| = 8$ options et $|Classes(C)| = 20$ classes de voitures différentes par instance ; les contraintes de capacité sur les options, définies par les fonctions p et q, sont générées aléatoirement en respectant les contraintes suivantes : $\forall o_i \in O, 1 \leq p(o_i) \leq 3$ et $p(o_i) < q(o_i) \leq p(o_i) + 2$. Toutes ces instances admettent une solution ne violant aucune contrainte.

Ces instances sont regroupées en trois groupes en fonction du nombre de voitures $|C|$ à séquencer : le premier groupe contient 32 instances de 100 voitures, le deuxième 21 instances de 300 voitures et le troisième 29 instances de 500 voitures. Pour chacun de ces trois groupes, nous avons séparé l'ensemble des instances en deux sous-groupes en fonction de leur « difficulté » : lorsque tous les algorithmes considérés ont réussi à résoudre l'instance, on considère qu'elle est « facile », sinon on considère qu'elle est « difficile ». Ainsi, il y a 23 (resp. 14 et 13) instances à 100 (resp. 300 et 500) voitures qui sont faciles et 9 (resp. 7 et 16) instances à 100 (resp. 300 et 500) voitures qui sont difficiles.

Conditions expérimentales. Les différents algorithmes considérés et leur paramétrage sont les suivants.

Greedy(η) est l'algorithme glouton aléatoire décrit en 3. Pour cet algorithme, nous avons fixé le poids du facteur heuristique β à 6.

ACO(τ_1, η) est l'algorithme ACO basé sur la première structure phéromonale décrite en 4, combinée avec l'heuristique de l'algorithme glouton η. Pour cet algorithme, nous avons fixé le poids du facteur phéromonal α_1 à 2, le poids du facteur heuristique β_1 à 6, le taux de persistance phéromonale ρ_1 à 0.99, et les bornes phéromonales minimale et maximale τ_{min_1} et τ_{max_1} à 0.01 et 4 respectivement.

ACO(τ_2) est l'algorithme ACO basé sur la seconde structure phéromonale décrite en 5. Pour cet algorithme, nous avons fixé le poids du facteur phéromonal α_2 à 6, le taux de persistance phéromonale ρ_2 à 0.99, et la borne phéromonale minimale τ_{min_2} à 1.

ACO(τ_1, τ_2) est l'algorithme ACO combinant les deux structures phéromonales. Pour la première structure phéromonale, nous avons fixé le poids du facteur phéromonal α_1 à 2, le taux de persistance phéromonale ρ_1 à 0.99, et les bornes phéromonales minimale et maximale τ_{min_1} et τ_{max_1} à 0.01 et 4 respectivement. Pour la seconde structure phéromonale, nous avons fixé le poids du facteur phéromonal α_2 à 6, le taux de persistance phéromonale ρ_2 à 0.99, et la borne phéromonale minimale τ_{min_2} à 1.

Chaque exécution de chacun de ces algorithmes est limitée à 150000 constructions de séquences : pour *ACO(τ_1, η)* et *ACO(τ_1, τ_2)* nous avons fixé le nombre maximum de cycles à 5000 et le nombre de fourmis à 30 ; pour *Greedy(η)*

et *ACO(τ_2)* nous avons fixé le nombre de cycles à 150000 étant donné qu'une seule séquence est construite à chaque cycle.

Les algorithmes ont été implémentés en C et ont été exécutés sur un Pentium 4 cadencé à 2GHz. Comme ils sont non déterministes, nous les avons exécutés 50 fois pour chaque instance considérée.

Comparaison sur les instances faciles. Pour les instances faciles, toutes les exécutions de chacun des quatre algorithmes considérés ont trouvé une solution. Ces instances ne sont donc pas très discriminantes pour nos algorithmes. On constate cependant des différences en ce qui concerne le nombre moyen de séquences construites et le temps CPU correspondant pour trouver une solution : *Greedy(η)* construit en moyenne 2696 séquences en 0.96 secondes ; *ACO(τ_1, η)* en construit 602 en 0.24 secondes ; *ACO(τ_2)* en construit 593 en 0.13 secondes et *ACO(τ_1, τ_2)* en construit 206 en 0.07 secondes.

Comparaison sur les instances difficiles. Le tableau 1 donne les résultats obtenus par chacun des quatre algorithmes pour les instances difficiles. Comparons tout d'abord les performances de *Greedy(η)* et *ACO(τ_2)*, qui construisent tous deux les séquences en utilisant une heuristique gloutonne estimant la « difficulté » des classes de voitures, mais qui diffèrent sur la façon de faire cette estimation : *Greedy(η)* se base sur la somme des taux d'utilisation des options demandées par les classes, tandis que *ACO(τ_2)* « apprend » cela en fonction de l'expérience passée. On constate qu'à l'exception (notable) de l'instance 500-52, *ACO(τ_2)* a un pourcentage de réussite nettement supérieur à *Greedy(η)*, et est également bien plus rapide en général : les deux algorithmes mettent à peu près le même temps pour construire une séquence, mais *ACO(τ_2)* en construit généralement sensiblement moins que *Greedy(η)* pour trouver une solution.

Lorsque l'on compare les performances de *Greedy(η)* et *ACO(τ_2)* avec celles de *ACO(τ_1, η)* et *ACO(τ_1, τ_2)* respectivement, on remarque que l'intégration de la structure phéromonale pour « apprendre » les bonnes séquences de voitures améliore toujours les performances : *ACO(τ_1, η)* (resp. *ACO(τ_1, τ_2)*) a été capable de résoudre 10 (resp. 14) instances de plus que *Greedy(η)* (resp. *ACO(τ_2)*) ; le nombre de séquences construites pour trouver une solution, et donc les temps d'exécutions, étant également nettement plus petits. Ainsi, l'algorithme combinant les deux structures phéromonales —*ACO(τ_1, τ_2)*— a été capable de résoudre chaque instance au moins une fois sur les 50 exécutions considérées.

Notons par ailleurs que pour des instances de même taille, les temps d'exécution d'*ACO(τ_1, τ_2)* peuvent varier considérablement : certaines instances (comme 500-64 ou 500-65) sont en fait résolues par la deuxième structure phéromonale (visant à « apprendre » les classes critiques) en quelques cycles seulement, tandis que d'autres (comme 500-13 ou 500-27) ont besoin de plusieurs centaines de cycles avant que la première structure phéromonale (visant à « apprendre » les bonnes séquences) ne permette à l'algorithme de converger.

8 Comparaison avec d'autres approches

Approches considérées. On compare maintenant le meilleur de nos quatre algorithmes, *ACO(τ_1, τ_2)*, avec deux approches par recherche locale qui se sont avérées particulièrement performantes, i.e., *IDWalk* et *PV*.

IDWalk [20] est une nouvelle méta-heuristique basée sur la recherche locale qui s'est avérée plus performante que d'autres méta-heuristiques basées sur la recherche locale —comme la recherche taboue et le recuit simulé— notamment pour le problème d'ordonnancement de voitures. En plus du nombre maximum S de mouvements autorisés, *IDWalk* a un seul paramètre Max qui donne le nombre maximum de voisins considérés à chaque mouvement. A chaque itération, l'algorithme choisit le premier voisin non détériorant, et si tous les Max voisins détériorent la configuration courante, l'algorithme choisit celui qui détériore le moins la configuration. Le paramètre Max est réglé automatiquement en effectuant au début de la recherche quelques marches courtes (de 20000 mouvements) avec différentes valeurs possibles, puis en lançant une marche longue avec la valeur du paramètre réglé. Ce paramétrage est également ajusté à plusieurs reprises au cours de la résolution lorsque la longueur de la marche atteint certains seuils. Dans le cas du problème d'ordonnancement de voitures, le voisinage considéré est un échange entre deux voitures appartenant à des classes différentes dont une participe à un conflit ; la voiture participant à un conflit est choisie selon une probabilité proportionnelle au nombre de contraintes qu'elle viole. La séquence initiale de voitures à partir de laquelle la recherche locale est effectuée est une permutation aléatoire de l'ensemble des voitures à produire.

PV (pour « Petit Voisinage ») [10] est l'algorithme qui a remporté le challenge ROADEF [21], challenge qui a vu s'affronter 27 équipes de chercheurs du monde entier pour la résolution d'un problème d'ordonnancement de voitures posé par Renault. Ce problème intègre, en plus des contraintes de ratio liées aux capacités des stations de montage, des contraintes liées aux couleurs visant à minimiser la consommation de solvant. Le problème d'ordonnancement de voitures « classique » auquel nous nous sommes intéressés est donc un cas particulier du problème du challenge, et les algorithmes conçus pour le challenge peuvent être utilisés pour résoudre les instances classiques. L'algorithme qui a remporté le challenge s'inspire de l'algorithme par recherche locale de [14] : partant d'une séquence ini-

TAB. 1 – Résultats expérimentaux de *Greedy(η)*, *ACO(τ_2)*, *ACO(τ_1, η)* et *ACO(τ_1, τ_2)* sur les instances difficiles. Chaque ligne donne le numéro de l'instance suivi des résultats obtenus par chacun des quatre algorithmes : le pourcentage d'exécutions ayant trouvé une solution, le temps CPU moyen en secondes (suivi de l'écart-type entre parenthèses) mis pour trouver une solution et le nombre de séquences construites divisé par 30 (ce nombre correspond au nombre de cycles effectués par *ACO(τ_1, η)* et *ACO(τ_1, τ_2)* puisque ces deux algorithmes construisent 30 séquences par cycle).

	Greedy(η)			*ACO(τ_2)*			*ACO(τ_1, η)*			*ACO(τ_1, τ_2)*		
Nom	%	Temps	$\frac{\#\text{ seq}}{30}$	%	Temps	$\frac{\#\text{ seq}}{30}$	%	Temps	$\frac{\#\text{ seq}}{30}$	%	Temps	$\frac{\#\text{ seq}}{30}$
100-11	16	9.1 (5.6)	2331	100	3.2 (3.2)	909	100	2.9 (1.2)	649	100	0.7 (0.3)	176
100-16	0			12	7.8 (3.6)	2347	0			100	1.5 (0.6)	420
100-19	0			0			0			84	8.9 (4.3)	2339
100-21	2	15.3 (0.0)	4178	6	11.9 (4.2)	3489	44	9.0 (4.8)	2097	98	2.7 (2.5)	730
100-60	0			86	4.3 (3.6)	1312	0			100	0.9 (0.3)	246
100-82	10	7.7 (4.6)	2163	54	7.2 (3.3)	2330	64	2.3 (3.8)	571	100	2.0 (2.0)	608
100-85	0			10	5.0 (4.1)	1649	0			24	9.5 (4.1)	2798
100-86	0			0			0			52	6.5 (4.8)	1795
100-98	0			0			4	17.6 (4.2)	4276	2	26.5 (0.0)	4809
300-08	38	20.5(12.0)	2133	100	0.2 (0.2)	20	100	1.8 (0.5)	152	100	0.1 (0.1)	12
300-25	78	19.8(15.0)	1940	100	0.6 (0.6)	65	100	2.0 (0.4)	163	100	0.3 (0.3)	30
300-37	0			100	4.0 (3.8)	447	100	4.2 (3.2)	349	100	1.5 (0.7)	142
300-40	0			0			0			82	16.0 (11.5)	1482
300-45	0			78	19.6 (13.1)	2181	72	8.6(10.0)	687	100	2.9 (1.1)	280
300-80	8	26.2(14.5)	2432	100	0.1 (0.1)	11	100	2.8 (0.5)	218	100	0.1 (0.1)	8
300-81	0			0			0			100	14.7 (10.3)	1388
500-07	0			0			0			100	15.3 (2.9)	593
500-13	0			0			0			54	38.2 (27.2)	1536
500-27	0			0			0			24	42.5 (21.2)	1614
500-28	0			0			0			100	15.7 (2.8)	593
500-30	0			100	0.0 (0.0)	2	90	13.4 (4.1)	549	100	0.0 (0.0)	2
500-34	0			0			0			24	43.7 (24.5)	1758
500-38	0			100	0.5 (0.6)	27	0			100	0.7 (0.7)	26
500-52	100	11.4(11.4)	698	0			100	0.8 (0.3)	36	100	11.3 (1.7)	484
500-53	0			100	5.6 (7.0)	356	96	14.5(20.9)	570	100	1.7 (1.3)	72
500-62	0			0			100	9.7 (1.8)	421	100	8.4 (0.7)	350
500-64	98	20.5(19.3)	1169	100	0.0 (0.0)	1	100	1.6 (0.4)	62	100	0.0 (0.0)	1
500-65	32	37.9(24.8)	2519	100	0.1 (0.1)	9	100	1.9 (0.5)	89	100	0.1 (0.1)	8
500-68	0			0			12	55.2(28.8)	2072	100	16.5 (5.9)	673
500-77	0			100	0.2 (0.2)	11	100	7.4 (0.6)	297	100	0.2 (0.2)	10
500-88	0			18	37.1 (15.0)	2570	22	55.5(36.9)	2097	22	18.5 (15.2)	891
500-93	0			0			38	25.5(24.8)	1067	100	9.1 (1.1)	367

tiale calculée à l'aide d'un algorithme glouton similaire à *Greedy(η)*, l'algorithme choisit à chaque itération le premier voisin de la configuration courante qui ne détériore pas son coût. Le voisinage considéré est défini par un ensemble de cinq transformations possibles (échange de 2 voitures, insertion d'une voiture en avant ou en arrière, miroir et mélange aléatoire), la probabilité de choisir chacune de ces transformations étant respectivement de 0.6, 0.13, 0.13, 0.13 et 0.01.

Jeux d'essais. Nous considérons les instances générées par Perron et Shaw [22] comme en 7, et nous séparons également ces instances en deux groupes en fonction de leur « difficulté », mais on considère pour cela les taux de réussite des trois algorithmes comparés *IDWalk*, *ACO(τ_1, τ_2)* et *PV* : on considère qu'une instance est facile lorsque les taux de réussite sont de 100% pour les trois algorithmes. Ainsi, il y a 25 (resp. 13 et 17) instances à 100 (resp. 300 et 500) voitures qui sont faciles et 7 (resp. 8 et 12) instances à 100 (resp. 300 et 500) voitures qui sont difficiles.

Conditions expérimentales. *ACO(τ_1, τ_2)* est exécuté sur un Pentium 4 cadencé à 2GHz. Il est paramétré comme en 7 de sorte qu'il construit au maximum $5000 * 30$ séquences, ce qui prend en moyenne 22, 53 et 123 secondes pour les instances ayant respectivement 100, 300 et 500 voitures. L'algorithme a été exécuté 50 fois sur chaque instance.

IDWalk [20] est exécuté sur une machine cadencée à 2.2GHz et chaque exécution a été limitée à 30, 60 et 130 secondes pour les instances ayant respectivement 100, 300 et 500 voitures. L'algorithme a été exécuté 13 fois sur chaque instance.

PV [10] est exécuté sur une machine cadencée à 3GHz et chaque exécution a été limitée à 20, 40 et 90 secondes pour les instances ayant respectivement 100, 300 et 500 voitures. L'algorithme a été exécuté 13 fois sur chaque instance.

Les résultats obtenus par *IDWalk* et *PV* ont été fournis par les auteurs respectifs de ces approches (et je les en remercie très chaleureusement), ce qui explique les différences de conditions expérimentales.

Comparaison sur les instances faciles. Le tableau 2 donne les résultats obtenus par chacun des trois algorithmes sur les instances faciles, i.e., les instances pour lesquelles toutes les exécutions de tous les algorithmes ont trouvé une solution. Pour ces instances, on constate que *ACO(τ_1, τ_2)* est plus rapide que *PV* qui est lui-même plus rapide que *IDWalk*, mais pour les instances à 500 voitures, les temps de *PV* se rapprochent de ceux de *ACO(τ_1, τ_2)*.

Quand on compare les algorithmes à base de recherche locale sur le nombre de mouvements effectués, on remarque que *PV* fait moins de mouvements que *IDWalk*, ce qui s'explique probablement d'une part par le fait que *PV* démarre sa recherche à partir d'une solution générée par un algorithme glouton alors que *IDWalk* la démarre d'une solution générée aléatoirement, et d'autre part par le fait que les mouvements de *PV* sont plus sophistiqués que ceux de *IDWalk* (qui ne considère que l'échange de deux voitures).

Comparaison sur les instances difficiles. Le tableau 3 donne les résultats obtenus par chacun des trois algorithmes sur les instances difficiles. Pour ces instances, on constate qu'il n'y a pas une méthode meilleure que les autres sur toutes les instances. En général, *ACO(τ_1, τ_2)* et *PV* sont meilleurs que *IDWalk* mais *IDWalk* est meilleur que *ACO(τ_1, τ_2)* pour trois instances à 500 voitures (500-27, 500-34 et 500-88). Lorsque l'on compare les performances de *ACO(τ_1, τ_2)* avec celles de *PV*, on constate que pour 7 (resp. 6) instances *ACO(τ_1, τ_2)* a un meilleur (resp. moins bon) taux de réussite que *PV*. En revanche, *ACO(τ_1, τ_2)* est souvent beaucoup plus rapide que *PV*, d'autant plus que la machine sur laquelle *PV* a été exécuté est plus puissante.

On constate par ailleurs que *ACO(τ_1, τ_2)* a des temps de calculs souvent plus stables que *PV* : à chaque fois que ces deux approches résolvent 100% des exécutions d'une même instance, l'écart-type de *ACO(τ_1, τ_2)* est nettement inférieur à celui de *PV*.

On constate également des différences entre ACO d'une part et les deux approches par recherche locale d'autre part en ce qui concerne le passage à l'échelle. En effet, pour *IDWalk* et *PV* la complexité en temps d'un mouvement ne dépend pas du nombre de voitures à séquencer : chaque mouvement est évalué « localement » en considérant à chaque fois les voitures adjacentes aux voitures concernées par le mouvement de sorte que la complexité d'un mouvement ne dépend que de la fonction q définissant le nombre de voitures sur lesquelles on doit vérifier la contrainte de capacité. Ainsi, *IDWalk* (resp. *PV*) font en moyenne 34481, 33757 et 33183 (resp. 3788, 7648 et 6587) mouvements par seconde pour les instances ayant respectivement 100, 300 et 500 voitures. En revanche, la complexité en temps de *ACO(τ_1, τ_2)* dépend à la fois du nombre de voitures à séquencer et du nombre de classes de voitures différentes : s'il y a $|C| = n$ voitures à séquencer et $|classes(C)| = k$ classes de voitures différentes, alors la complexité de la construction d'une séquence par une fourmi est en $\mathcal{O}(n \cdot k)$, la complexité du dépot phéromonal est en $\mathcal{O}(n)$ et la complexité de l'évaporation est en $\mathcal{O}(n^2)$. Ainsi, *ACO(τ_1, τ_2)* construit en moyenne 6944, 2816 et 1220 séquences par seconde pour les instances ayant respectivement 100, 300 et 500 voitures. Cette progression laisse penser que pour des instances ayant un plus grand nombre de voitures, *IDWalk* et *PV* pourraient devenir plus rapides qu'*ACO(τ_1, τ_2)*.

Ces résultats doivent également être modulés par le fait qu'*IDWalk* est une méta-heuristique généraliste qui n'a pas été spécialement conçue pour résoudre le problème d'ordonnancement de voitures et qui gagnerait probablement en performances si elle intégrait des heuristiques propres au problème considéré, notamment pour générer la configuration initiale à partir de laquelle la recherche locale est effectuée (ce que fait *PV*). De même, *PV* a été mis au point pour résoudre les instances du Challenge ROADEF et les résultats donnés ici ont été obtenus avec le même paramétrage (en ce qui concerne les probabilités de transformation notamment) que pour le challenge. Par ailleurs, la limite de temps de 20 secondes pour les instances à 100 voitures apparait trop courte pour cette approche, et si on augmente le temps maximum d'exécution à 180 secondes, alors *PV* a un taux de réussite de 100% pour les instances 100-21, 100-85 et 100-86 tandis que les taux de réussite de 100-19 et 100-60 passent à 85% et 23% respectivement. En revanche, l'instance 100-98 n'est toujours pas résolue, par aucune des 13 exécutions.

TAB. 2 – Résultats expérimentaux de *IDWalk*, *ACO(τ_1, τ_2)* et *PV* sur les instances faciles. Pour chaque algorithme, on donne le temps CPU moyen en secondes (suivi de l'écart-type entre parenthèses) pour trouver une solution ; pour *IDWalk* et *PV* on donne le nombre de centaines de mouvements effectués pour trouver une solution tandis que pour *ACO(τ_1, τ_2)* on donne le nombre de séquences construites.

	IDWalk		*ACO(τ_1, τ_2)*		*PV*	
	Temps	$\frac{\#\text{mv}}{100}$	Temps	# seq	Temps	$\frac{\#\text{mv}}{100}$
25 instances with 100 cars	2.2 (1.4)	282	0.2 (0.1)	1692	1.1 (0.7)	82
13 instances with 300 cars	11.2 (4.5)	2907	0.2 (0.1)	421	2.4 (0.8)	193
17 instances with 500 cars	21.4 (9.2)	5278	4.5 (1.0)	5482	5.3 (2.0)	350

TAB. 3 – Résultats expérimentaux de *IDWalk*, *ACO(τ_1, τ_2)* et *PV* sur les instances difficiles. Chaque ligne donne le numéro de l'instance, suivi pour chaque algorithme du pourcentage d'exécutions ayant trouvé une solution, du temps CPU moyen en secondes (suivi de l'écart-type entre parenthèses) pour trouver une solution ; pour *IDWalk* et *PV* on donne le nombre de centaines de mouvements effectués pour trouver une solution tandis que pour *ACO(τ_1, τ_2)* on donne le nombre de séquences construites.

	IDWalk			*ACO(τ_1, τ_2)*			*PV*		
Nom	%	Temps	$\frac{\#\text{mv}}{100}$	%	Temps	# seq	%	Temps	$\frac{\#\text{mv}}{100}$
100-19	0			84	8.9 (4.3)	70186	15	17.5 (0.7)	475
100-21	77	22.1 (8.5)	6146	98	2.7 (2.5)	21902	92	7.2 (5.3)	256
100-60	0			100	0.9 (0.3)	7395	15	14.1 (4.3)	315
100-85	0			24	9.5 (4.1)	83962	54	8.5 (4.9)	671
100-86	77	17.5 (8.9)	4093	52	6.5 (4.8)	53867	92	6.5 (4.4)	216
100-98	0			2	26.5 (0.0)	144280	0		
100-99	77	16.3 (5.7)	9037	100	0.0 (0.0)	345	100	3.3 (3.0)	230
300-08	0			100	0.1 (0.1)	380	100	21.0 (9.0)	1847
300-25	92	14.5 (3.7)	4813	100	0.3 (0.3)	924	100	3.6 (1.6)	370
300-36	23	39.2 (10.2)	17290	100	0.0 (0.0)	5	100	4.7 (2.9)	717
300-40	0			82	16.0 (11.5)	44475	38	30.1 (6.9)	1353
300-45	77	26.4 (16.0)	7657	100	2.9 (1.1)	8422	85	13.3 (9.1)	633
300-47	54	35.7 (21.9)	10800	100	0.0 (0.0)	12	100	6.8 (3.0)	880
300-78	92	20.3 (11.6)	5383	100	0.3 (0.2)	727	100	4.7 (1.8)	316
300-81	0			100	14.7 (10.3)	41643	69	24.2 (6.6)	2175
500-08	92	47.8 (27.9)	26520	100	0.1 (0.0)	66	100	6.6 (1.8)	420
500-13	38	97.5 (17.4)	35310	54	38.2 (27.2)	46089	100	13.5 (6.1)	818
500-27	100	26.3 (16.8)	9122	24	42.5 (21.2)	48438	100	11.0 (2.9)	1099
500-30	62	54.3 (37.8)	8346	100	0.0 (0.0)	64	100	10.7 (3.9)	528
500-34	100	18.0 (2.7)	2604	24	43.7 (24.5)	52770	100	9.3 (3.9)	470
500-44	8	66.9 (0.0)	23440	100	0.1 (0.1)	107	100	10.8 (7.2)	869
500-53	0			100	1.7 (1.3)	2174	100	30.2 (13.8)	2951
500-56	85	36.1 (23.8)	11660	100	1.1 (1.1)	1344	100	19.7 (5.5)	602
500-65	0			100	0.1 (0.1)	249	100	25.7 (9.3)	977
500-74	69	70.6 (31.2)	30240	100	0.4 (0.4)	432	100	8.2 (1.9)	669
500-77	92	69.5 (24.3)	16140	100	0.2 (0.2)	326	100	9.6 (3.4)	712
500-88	100	26.3 (10.5)	6945	22	18.5 (15.2)	26741	100	10.8 (3.5)	826

9 Conclusion

Nous avons introduit deux structures phéromonales différentes pour résoudre le problème d'ordonnancement de voitures avec un algorithme basé sur l'optimisation par colonies de fourmis. Ces deux structures phéromonales ont

des objectifs complémentaires : la première vise à identifier les bonnes sous-séquences de voitures tandis que la seconde vise à identifier les voitures critiques. Nous avons montré que la combinaison de ces deux structures complémentaires permet de résoudre très rapidement beaucoup d'instances, souvent plus rapidement que l'approche par recherche locale qui a gagné le challenge ROADEF 2005. Cependant sur certaines instances, ACO obtient de moins bons résultats que la recherche locale.

Une conclusion naturelle de ces résultats expérimentaux est que, ACO et recherche locale ayant des performances complémentaires, il serait certainement intéressant de les hybrider : les fourmis construisent des solutions en exploitant la phéromone, et la recherche locale améliore la qualité des solutions construites par les fourmis. De fait, une telle hybridation entre ACO et recherche locale s'est avérée particulièrement fructueuse pour de nombreux problèmes d'optimisation combinatoire [7, 29, 26, 28].

Enfin, même si cela est rarement pris en compte pour comparer des approches, on peut souligner la simplicité de l'algorithme ACO, qui est très facile à implémenter : le code C implémentant *ACO(*τ_1, τ_2*)* comporte moins de 300 lignes tout compris. Cela n'est généralement pas le cas pour les approches par recherche locale qui doivent mettre en œuvre des structures de données évoluées afin d'évaluer le plus incrémentalement possible le coût de chaque mouvement.

Notons toutefois qu'*ACO* nécessite le réglage de 9 paramètres tandis que *PV* n'en demande que 5, et *IDWalk* 1. Nos travaux futurs viseront à simplifier et tenter d'automatiser cette étape de paramétrage.

Références

[1] B. Bullnheimer, R.F. Hartl, and C. Strauss. An improved ant system algorithm for the vehicle routing problem. *Annals of Operations Research*, 89 :319–328, 1999.

[2] A.J. Davenport and E.P.K. Tsang. Solving constraint satisfaction sequencing problems by iterative repair. In *Proceedings of the first international conference on the practical applications of constraint technologies and logic programming (PACLP)*, pages 345–357, 1999.

[3] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In Y. Kodratoff, editor, *Proceedings of ECAI-88*, pages 290–295, 1988.

[4] M. Dorigo. *Optimization, Learning and Natural Algorithms* (in Italian). PhD thesis, Dipartimento di Elettronica, Politecnico di Milano, Italy, 1992.

[5] M. Dorigo, G. Di Caro, and L.M. Gambardella. Ant algorithms for discrete optimization. *Artificial Life*, 5(2) :137–172, 1999.

[6] M. Dorigo and G. Di Caro. The Ant Colony Optimization meta-heuristic. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 11–32. McGraw Hill, UK, 1999.

[7] M. Dorigo and L.M. Gambardella. Ant colony system : A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1) :53–66, 1997.

[8] M. Dorigo, V. Maniezzo, and A. Colorni. Ant System : Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics – Part B : Cybernetics*, 26(1) :29–41, 1996.

[9] M. Dorigo and T. Stuetzle. *Ant Colony Optimization*. MIT Press, 2004.

[10] B. Estellon, F. Gardi, and K. Nouioua. Ordonnancement de véhicules : une approche par recherche locale à grand voisinage. In *Actes des premières Journées Francophones de Programmation par Contraintes (JFPC)*, pages 21–28, 2005.

[11] L.M. Gambardella, E. Taillard, and M. Dorigo. Ant colonies for the quadratic assignment problem. *Journal of the Operational Research Society*, 50 :167–176, 1999.

[12] L.M. Gambardella, E.D. Taillard, and G. Agazzi. MACS-VRPTW : A multiple ant colony system for vehicle routing problems with time windows. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 63–76. McGraw Hill, London, UK, 1999.

[13] I.P. Gent and T. Walsh. Csplib : a benchmark library for constraints. Technical report, APES-09-1999, 1999. available from http ://csplib.cs.strath.ac.uk/. A shorter version appears in CP99.

[14] J. Gottlieb, M. Puchta, and C. Solnon. A study of greedy, local search and ant colony optimization approaches for car sequencing problems. In *Applications of evolutionary computing*, volume 2611 of *LNCS*, pages 246–257. Springer, 2003.

[15] M. Gravel, C. Gagné, and W.L. Price. Review and comparison of three methods for the solution of the car-sequencing problem. *Journal of the Operational Research Society*, 2004.

[16] T. Kis. On the complexity of the car sequencing problem. *Operations Research Letters*, 32 :331–335, 2004.

[17] J.H.M. Lee, H.F. Leung, and H.W. Won. Performance of a comprehensive and efficient constraint library using local search. In *11th Australian JCAI*, LNAI. Springer-Verlag, 1998.

[18] V. Maniezzo and A. Colorni. The Ant System applied to the quadratic assignment problem. *IEEE Transactions on Data and Knowledge Engineering*, 11(5) :769–778, 1999.

[19] L. Michel and P. Van Hentenryck. A constraint-based architecture for local search. In *OOPSLA '02 : Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 83–100, New York, NY, USA, 2002. ACM Press.

[20] B. Neveu, G. Trombettoni, and F. Glover. Id walk : A candidate list strategy with a simple diversification device. In *Proceedings of CP'2004*, volume 3258 of *LNCS*, pages 423–437. Springer Verlag, 2004.

[21] A. Nguyen and V.-D. Cung. Le problème du car sequencing renault et le challenge roadef'2005. In *Premières Journées Francophones de Programmation par Contraintes (JFPC 2005)*, pages 3–10, 2005.

[22] L. Perron and P. Shaw. Combining forces to solve the car sequencing problem. In *Proceedings of CP-AI-OR'2004*, volume 3011 of *LNCS*, pages 225–239. Springer, 2004.

[23] M. Puchta and J. Gottlieb. Solving car sequencing problems by local optimization. In *Applications of Evolutionary Computing (EvoCOP 2002)*, volume 2279 of *LNCS*, pages 132–142. Springer, 2002.

[24] J.-C. Regin and J.-F. Puget. A filtering algorithm for global sequencing constraints. In *CP97*, volume 1330 of *LNCS*, pages 32–46. Springer-Verlag, 1997.

[25] C. Solnon. Solving permutation constraint satisfaction problems with artificial ants. In *Proceedings of ECAI'2000, IOS Press, Amsterdam, The Netherlands*, pages 118–122, 2000.

[26] C. Solnon. Ants can solve constraint satisfaction problems. *IEEE Transactions on Evolutionary Computation*, 6(4) :347–357, 2002.

[27] C. Solnon, V.D. Cung, A. Nguyen, and C. Artigues. The car sequencing problem : overview of state-of-the-art methods and industrial case-study of the roadef'2005 challenge problem. *European Journal of Operation Research (EJOR), to appear*, 2006.

[28] C. Solnon and S. Fenet. A study of aco capabilities for solving the maximum clique problem. *Journal of Heuristics*, 12(3) :155–180, 2006.

[29] T. Stützle and H.H. Hoos. MAX-MIN Ant System. *Journal of Future Generation Computer Systems, special isuue on Ant Algorithms*, 16 :889–914, 2000.

[30] E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, UK, 1993.

[31] T. Warwick and E. Tsang. Tackling car sequencing problems using a genetic algorithm. *Journal of Evolutionary Computation - MIT Press*, 3(3) :267–298, 1995.