# Ant-P-solveur : un solveur de contraintes à base de fourmis artificielles

**Christine Solnon**

*Laboratoire LISI (Université Lyon 1 - INSA de Lyon)*
*Bâtiment 710*
*43 bd du 11 novembre*
*69 622 Villeurbanne cedex*
*csolnon@bat710.univ-lyon1.fr*

RÉSUMÉ. *On décrit dans ce papier Ant-P-solveur, un solveur de contraintes inspiré du comportement collectif des fourmis. L'idée est de représenter le problème à résoudre sous la forme de la recherche d'un "meilleur" chemin dans un graphe. Des fourmis artificielles circulent dans ce graphe de façon aléatoire et incomplète, à la recherche de "bons" chemins. Elles communiquent entre elles, à travers l'environnement, en déposant sur les arcs du graphe une trace volatile.*
*Ant-P-solveur permet de résoudre un ensemble de problèmes de satisfaction de contraintes, appelés PermutCSPs. Ces problèmes consistent à chercher une permutation d'un tuple de valeurs satisfaisant un certain nombre de contraintes. On évalue les performances de Ant-P-solveur sur trois de ces problèmes: le problème des reines, le problème des intervalles et le problème d'ordonnancement de voitures.*

ABSTRACT. *We describe in this paper Ant-P-solver, a generic constraint solver based on the Ant Colony Optimization (ACO) metaheuristic. The ACO metaheuristic takes inspiration on the observation of real ants collective foraging behaviour. The idea is to model the problem as the search of a best path in a graph. Artificial ants walk trough this graph, in a stochastic and incomplete way, searching for good paths. Artificial ants communicate in a local and indirect way, by laying a pheromone trail on the edges of the graph.*
*Ant-P-solver has been designed to solve a general class of combinatorial problems: permutation constraint satisfaction problems, the goal of which is to find a permutation of $n$ known values, to be assigned to $n$ variables, under some constraints. Many constraint satisfaction problems involve such global permutation constraints. Ant-P-solver capabilities are illustrated, and compared with other approaches, on three of these problems, i.e., the n-queens, the all-interval series and the car sequencing problems.*

MOTS-CLÉS : *Algorithmes à base de fourmis artificielles, Satisfaction de contraintes, Contrainte globale de permutation.*

KEYWORDS: *Ant Colony Optimization (ACO) metaheuristic, Constraint Satisfaction, Global permutation constraint.*

# 1. Introduction

Les fourmis sont capables de résoudre collectivement des problèmes complexes, comme trouver le plus court chemin entre deux points dans un environnement accidenté. Pour cela, elles communiquent entre elles de façon locale et indirecte, grâce à une hormone volatile, appelée phéromone : au cours de leur progression, les fourmis déposent une trace de phéromone ; elles choisissent ensuite leur chemin de façon aléatoire, selon une probabilité dépendant de la quantité de phéromone précédemment déposée. La capacité collective des fourmis à trouver un plus court chemin s'explique essentiellement par le fait que plus un chemin est court, plus vite la fourmi retourne à la fourmilière par ce chemin, en redéposant de la phéromone, et plus de nouvelles fourmis sont attirées sur ce chemin et le renforcent. Un point clé de ce mécanisme, qui permet aux fourmis de s'adapter aux changements de l'environnement, provient du fait que la phéromone est volatile: les traces de phéromone s'évaporent progressivement, permettant ainsi "d'oublier" certains chemins, mauvais ou obsolètes, pour en découvrir de nouveaux.

## 1.1. *Algorithmes à base de fourmis artificielles*

Ce mécanisme, qui permet aux fourmis de résoudre collectivement des problèmes complexes, est à l'origine des algorithmes à base de fourmis artificielles. Ces algorithmes ont été initialement proposés dans [DOR 92, DOR 96], comme une approche multi-agents pour résoudre des problèmes d'optimisation combinatoire. L'idée est de représenter le problème à résoudre sous la forme de la recherche d'un meilleur chemin dans un graphe, puis de lancer des fourmis artificielles dans ce graphe, à la recherche de bons chemins.

Certaines caractéristiques du comportement des fourmis artificielles sont inspirées de la nature. Ainsi, les fourmis artificielles déposent de la phéromone sur les arcs du graphe qu'elles empruntent, et elles choisissent leur chemin de façon aléatoire, en fonction de la quantité de phéromone déposée sur les arcs incidants. Par ailleurs, la quantité de phéromone est progressivement diminuée, simulant ainsi un phénomène d'évaporation qui permet d'éviter une convergence prématurée.

Les fourmis artificielles ont aussi d'autres caractéristiques, qui ne trouvent pas leur équivalent dans la nature. En particulier, elles peuvent avoir une mémoire, qui leur permet de garder une trace de leurs actions passées. Dans la plupart des cas, les fourmis ne déposent une trace de phéromone qu'après avoir effectué un chemin complet, et non de façon incrémentale au fur et à mesure de leur progression. De plus, la quantité de phéromone déposée par une fourmi artificielle est généralement fonction de la qualité du chemin, tandis que les fourmis réelles déposent à peu près toujours la même quantité de phéromone. Enfin, la probabilité pour une fourmi artificielle de choisir un arc ne dépend généralement pas uniquement des traces de phéromone, mais aussi d'une certaine visibilité locale permettant d'évaluer la qualité du chemin.

Ces différentes caractéristiques des fourmis artificielles définissent la "méta-heuristique d'optimisation par une colonie de fourmis" (ACO metaheuristic) [DOR 99a, DOR 99b]. Cette approche a permis de résoudre différents problèmes d'optimisation combinatoire, comme le problème du voyageur de commerce [DOR 97], le problème d'affectation quadratique [GAM 99] ou le problème de routage de véhicules [BUL 99].

### 1.2. *Motivations*

Pour résoudre un nouveau problème d'optimisation à l'aide de fourmis artificielles, il s'agit de formuler le problème sous la forme de la recherche d'un meilleur chemin dans un graphe. Un point clé de cette phase de modélisation consiste à définir le graphe support et la règle de transition probabiliste qui est utilisée localement par les fourmis pour choisir leur chemin dans ce graphe. Il s'agit notamment de prendre en compte les différentes contraintes du problème, de telle sorte que les fourmis ne puissent effectuer que des chemins valides, i.e., ne violant aucune contrainte. De fait, pour certains problèmes plus fortement contraints, cette phase de modélisation peut devenir problématique.

Dans ce papier, on étudie les capacités des algorithmes à base de fourmis artificielles pour la résolution de ces problèmes fortement contraints : le but n'est plus d'optimiser une fonction, sous certaines contraintes, mais de trouver une affectation qui satisfait effectivement toutes les contraintes. Une motivation fondamentale, sous-jacente à notre travail, est de fournir un outil générique qui puisse résoudre un ensemble de problèmes. On s'intéresse plus particulièrement dans ce papier aux problèmes de recherche de permutations sous contraintes, appelés PermutCSPs.

Ce travail peut être vu comme une généralisation de la "ACO metaheuritic" à une nouvelle classe de problèmes. Sa contribution originale concerne plus particulièrement la définition d'un graphe, associé de manière générique à n'importe quel PermutCSP, sur lequel les fourmis évoluent, et la définition d'une règle de transition qui spécifie les probabilités de transition des fourmis lorsqu'elles construisent leurs chemins.

### 1.3. *Définition d'un PermutCSP*

Un PermutCSP est un problème de satisfaction de contraintes (CSP) particulier, consistant à chercher une permutation de $n$ valeurs, devant être affectées à $n$ variables, sous un certain nombre de contraintes. De nombreux CSPs contiennent de telles contraintes de permutation, aussi appelées "cycling" ou "sequencing" contraintes.

**Définition d'un CSP:** un problème de satisfaction de contraintes [TSA 93] est défini par un triplet $(X, D, C)$ tel que

– $X = \{X_1, X_2, ..., X_n\}$ est un ensemble fini de $n$ variables,

– $D$ est une fonction qui associe à chaque variable $X_i \in X$ son domaine de valeurs $D(X_i)$, i.e., l'ensemble fini de valeurs que $X_i$ peut prendre,

– $C$ est un ensemble de contraintes, i.e., de relations qui restreignent les valeurs que peuvent prendre simultanément les variables.

Une solution d'un CSP $(X, D, C)$ est une affectation de toutes les variables de $X$ qui satisfait toutes les contraintes de $C$.

**Définition d'un PermutCSP:** un PermutCSP est un CSP particulier, tel que toutes les solutions du CSP sont des permutations d'un tuple donné. Un tel problème sera défini par un quadruplet $(X, D, C, P)$ tel que

– $(X, D, C)$ est un CSP,

– $P =< v_1, v_2, \ldots, v_n >$ est un tuple de $|X| = n$ valeurs.

Une solution d'un PermutCSP est une affectation de toutes les variables

$$A = \{X_1 \leftarrow r_1, X_2 \leftarrow r_2, ..., X_n \leftarrow r_n\}$$

qui est solution de $(X, D, C)$ et telle que $< r_1, r_2, \ldots, r_n >$ est une permutation du tuple $P$.

**Par exemple,** le problème des 4 reines, consistant à placer 4 reines sur un échiquier $4 \times 4$, de telle sorte qu'aucune reine ne soit en prise, peut être défini par le PermutCSP suivant:

$$\begin{array}{lcl} X & = & \{X_1, X_2, X_3, X_4\} \\ D(X_i) & = & \{1, 2, 3, 4\} \ \forall X_i \in X \\ C & = & \forall (X_i, X_j) \in X^2, \ i \neq j \Rightarrow \mid X_i - X_j \mid \neq \mid i - j \mid \\ P & = & < 1, 2, 3, 4 > \end{array}$$

Une solution de ce problème est la permutation $< \ 2, 4, 1, 3 \ >$ de $P$, telle que la première reine, à la ligne 1, est placée à la colonne 2, la reine à la ligne 2 est placée à la colonne 4, la reine à la ligne 3 est placée à la colonne 1, et la reine à la ligne 4 est placée à la colonne 3.

## 2. Description de Ant-P-solveur

Ant-P-solveur est un algorithme incomplet et stochastique à base de fourmis artificielles, permettant de résoudre de façon générique n'importe quel PermutCSP. L'idée est d'associer un graphe —appelé le graphe des permutations— au PermutCSP à résoudre. Une colonie de fourmis artificielles circule dans ce graphe, à la recherche de "bons" chemins, correspondant à des solutions du PermutCSP.

**Graphe des permutations associé à un PermutCSP:** Le graphe des permutations associe un sommet à chaque valeur du tuple $P$ de valeurs à permuter. Il y a en plus un sommet particulier correspondant à la fourmilière, noté $f$, à partir duquel les fourmis commenceront leurs recherches. Ainsi, le graphe des permutations associé à un PermutCSP $(X, D, C, P)$ où $P =< v_1, v_2, \ldots, v_n >$ est le graphe complet orienté $G = (V, E)$ tel que

$$\begin{array}{rcl} V & = & \{f, v_1, v_2, \ldots, v_n\} \\ E & = & \{(v_i, v_j) \in V^2\} \end{array}$$

Les fourmis déposent de la phéromone sur les arcs du graphe. La quantité de phéromone déposée sur un arc $(v_i, v_j)$ est notée $\tau(v_i, v_j)$.

**Définition d'un chemin:** un chemin dans un graphe des permutations $G = (V, E)$ est une suite de sommets de $V$. On ne considère que des chemins élémentaires, ne contenant pas de cycle. Un chemin partant d'un sommet $v_i$ et arrivant à un sommet $v_j$ est noté $\pi = v_i \rightsquigarrow v_j$.

Les chemins sont construits par les fourmis: elles partent de la fourmilière $f$, et visitent ensuite chacun des $n$ autres sommets du graphe, construisant ainsi un chemin hamiltonien. Le chemin construit par une fourmi correspond à une affectation complète: la valeur associée au $\text{i}^{\text{eme}}$ sommet du chemin est affectée à la $\text{i}^{\text{eme}}$ variable du PermutCSP.

**Evaluation d'un chemin:** l'objectif des fourmis est de trouver une affectation complète qui ne viole aucune contrainte. Par conséquent, un chemin est d'autant meilleur qu'il viole peu de contraintes, et l'évaluation d'un chemin $\pi$, dénotée par *eval*$(\pi)$, est le nombre de contraintes qui sont violées dans l'affectation correspondant au chemin $\pi$. Un chemin complet évalué à 0 est une solution du PermutCSP.

**Par exemple,** considérons de nouveau le problème des 4 reines. Les sommets du graphe des permutations associé sont $V = \{f, 1, 2, 3, 4\}$. Considérons maintenant une fourmi qui, partant de la fourmilière $f$, va successivement sur les sommets 2, 3, 1 puis 4. Le chemin correspondant est noté $\pi =< f, 2, 3, 1, 4 >$. Ce chemin correspond à l'affectation complète $A = \{X_1 \leftarrow 2, X_2 \leftarrow 3, X_3 \leftarrow 1, X_4 \leftarrow 4\}$. Cette affectation viole une contrainte. Par conséquent, le chemin $\pi$ est évalué à *eval*$(\pi) = 1$.

**Définition des probabilités de transition:** lors de la recherche d'un chemin, une fourmi artificielle choisit le prochain sommet à visiter, parmi l'ensemble des sommets qu'elle n'a pas encore visités, de façon aléatoire en fonction de probabilités de transition. La probabilité pour une fourmi ayant déjà parcouru un chemin $\pi$ d'aller sur un sommet $v_j$ est définie proportionnellement à la capacité d'attraction $\mathcal{A}_\pi(v_j)$ du sommet $v_j$ :

$$p_\pi(v_j) = \frac{\mathcal{A}_\pi(v_j)}{\sum_{v_k \in V} \mathcal{A}_\pi(v_k)}$$

La capacité d'attraction d'un sommet dépend à la fois de la phéromone précédemment déposée sur l'arc menant à ce sommet, et d'une certaine "visibilité" locale. Ainsi, la capacité d'attraction $\mathcal{A}_\pi(v_j)$ d'un sommet $v_j$ pour une fourmi ayant déjà parcouru le chemin $\pi = f \rightsquigarrow v_i$, est définie par

$$\begin{array}{rcll} \mathcal{A}_\pi(v_j) & = & 0 & \text{si } v_j \in \pi \\ \mathcal{A}_\pi(v_j) & = & \tau(v_i, v_j) * (local\_eval(\pi, v_j))^\beta & \text{si } v_j \notin \pi \end{array}$$

où $\beta$ est un paramètre qui contrôle le poids de la visibilité locale par rapport à la trace de phéromone. La visibilité locale évalue la qualité du prochain sommet $v_j$, et est inversement proportionnelle au nombre de nouvelles contraintes violées lorsqu'on affecte la valeur $v_j$ à la prochaine variable, i.e.,

$$local\_eval(\pi, v_j) = \frac{1}{1 + eval(\pi. < v_j >) - eval(\pi)}$$

**Fonction chemin:** La fonction "chemin($G$)" construit un chemin complet partant de la fourmilière $f$ dans un graphe des permutations $G$:

<u>fonction</u> chemin($G = (V, E)$) <u>retourne</u> un chemin complet
    $\pi \leftarrow < f >$
    *Candidats* $\leftarrow V - \{f\}$
    <u>Tant que</u> $Candidats \neq \emptyset$ <u>faire</u>
        choisir $v_j \in$ *Candidats* avec la probabilité $p_\pi(v_j)$
        ajouter $v_j$ à la fin du chemin $\pi$
        enlever $v_j$ de *Candidats*
    <u>fin tant que</u>
    <u>retourne</u>($\pi$)

**Fonction Ant-P-solveur:** La fonction "Ant-P-solveur($G, \rho$)" décrite page suivante retourne le meilleur chemin trouvé par des fourmis dans le graphe des permutations $G$. Le paramètre $\rho$ contrôle le taux d'évaporation de la phéromone. A chaque cycle de cet algorithme, chaque fourmi $a_i$ construit un chemin $\pi_i$. Si ce chemin est meilleur que $best\pi$, le meilleur chemin trouvé depuis le début de la recherche, alors $best\pi$ est mis à jour. A la fin de chaque cycle, une trace de phéromone est déposée sur les arcs du meilleur chemin $\pi$ trouvé pendant ce cycle. La quantité de phéromone déposée est proportionnelle à la qualité du chemin, i.e., elle est inversement proportionnelle à *eval*($\pi$). Enfin, la quantité de phéromone est uniformément diminuée sur tous les arcs du graphe. L'algorithme s'arrête soit quand une fourmi a trouvé une solution, soit quand un nombre maximal de cycles ont été effectués.

```
fonction Ant-P-solveur(G = (V, E), ρ) retourne le meilleur chemin complet trouvé
    nbCycles ← 0
    min ← ∞
    pour toute paire de sommets (v_i, v_j) ∈ V^2 faire: τ(v_i, v_j) ← 1
    tant que nbCycles < nbMaxCycles et min > 0 faire
        nbCycles ← nbCycles + 1
        /* Chaque fourmi a_i construit un chemin π_i */
        pour chaque fourmi a_i faire
            π_i ← chemin(G)
            si eval(π_i) <min alors min ←eval(π_i)
                                    bestπ ← π_i
        /* La meilleure fourmi du cycle courant dépose de la phéromone */
        soit π le meilleur chemin trouvé durant le cycle courant
        pour chaque arc (v_i, v_j) du chemin π faire
            τ(v_i, v_j) ← τ(v_i, v_j)+min/eval(π)
        /* Evaporation de la phéromone */
        pour toute paire (v_i, v_j) ∈ V^2 faire τ(v_i, v_j) ← τ(v_i, v_j) * ρ
    fin tant que
    retourne(bestπ)
```

**Valeurs des paramètres:** Ant-P-solveur est paramétré par le poids de la visibilité locale $\beta$, le taux d'évaporation $\rho$ et le nombre de fourmis. La valeur de ces paramètres a été fixée de façon expérimentale, en exécutant Ant-P-solveur avec différentes valeurs de paramètres, sur différents problèmes, et en sélectionnant les valeurs qui ont donné les meilleurs résultats en moyenne. Il faut remarquer que, dans un souci de généricité, nous avons utilisé les mêmes valeurs de paramètres pour résoudre tous les problèmes détaillés dans la section suivante.

– La valeur de $\beta$ a été fixée à 10: avec des valeurs plus petites, les fourmis trouvent globalement des chemins plus mauvais, de telle sorte que cela prend plus de temps pour converger vers la solution optimale; avec des valeurs plus grandes, les fourmis ne sont plus assez sensibles aux traces de phéromones, et ne sont donc plus capables de bénéficier des expériences passées.

– Le nombre de fourmis a été fixé à 8: avec des valeurs plus petites, chaque cycle dure moins longtemps, de telle sorte que l'algorithme converge plus rapidement, mais le meilleur chemin trouvé à chaque cycle est globalement moins bon de telle sorte que, pour les problèmes difficiles, l'algorithme converge vers des solutions non optimales. D'un autre coté, avec un plus grand nombre de fourmis, le meilleur chemin trouvé à chaque cycle n'est pas tellement meilleur que celui trouvé avec 8 fourmis, de sorte que la résolution collective n'est pas améliorée, tandis que chaque cycle prend plus de temps.

– Le taux d'évaporation a été fixé à $\rho = 0.99$ (soit $1\%$ d'évaporation à chaque cycle), de telle sorte que si la quantité de phéromone sur un arc est de 1, et si plus aucune "meilleure" fourmi ne passe sur cet arc, alors la quantité de phéromone devient insignifiante au bout de quelques centaines de cycles, et l'arc est "oublié".

## 3. Résolution de PermutCSPs avec Ant-P-solveur

Ant-P-solveur a été implémenté en C++. Pour résoudre un nouveau problème avec Ant-P-solveur, il suffit d'implémenter une classe C++, qui décrit effectivement le problème à résoudre. Cette classe spécifie essentiellement le tuple initial $P$ de valeurs à permuter, et la fonction "local_eval$(\pi, v_j)$" qui retourne le nombre de nouvelles contraintes violées lorsqu'on affecte la valeur $v_j$ à la prochaine variable, par rapport à l'affectation partielle en cours correspondant au chemin $\pi$.

On illustre maintenant notre approche sur trois problèmes classiques, et on la compare expérimentalement avec Ilog solveur[ILO 99], un solveur de référence basé sur une recherche complète. Dans la prochaine partie, on compare Ant-P-solveur avec d'autres approches.

Toutes les expérimentations, à la fois avec Ilog solveur et Ant-P-solveur, ont été effectuées sur la même machine, à savoir une station HP 715/100. Dans la mesure où Ant-P-solveur est non déterministe, il a été exécuté dix fois sur chaque instance considérée, et les résultats affichés sont une moyenne sur ces dix essais. D'une façon générale, on remarque que les dix exécutions donnent des résultats relativement proches, avec un faible écart type.

### 3.1. *Le problème des reines*

Les résultats obtenus pour résoudre le problème des reines, avec des instances de 50 à 200 reines, sont décrits dans le tableau suivant:

| | Ant-P-solveur | | Ilog solveur | |
|---|---|---|---|---|
| Nb var | Nb cycles | Temps | Nb pt de choix | Temps |
| 50 | 45 | 2.7s | 553 | 0.6s |
| 75 | 48 | 6.8s | 2 278 | 3.1s |
| 100 | 74 | 18.1s | 115 | 0.4s |
| 125 | 227 | 87.6s | 232 | 0.8s |
| 150 | 262 | 144.5s | 319 425 | 555.1s |
| 175 | 342 | 256.0s | - | $>$ 10 000.0s |
| 200 | 289 | 282.6s | 147 013 | 256.6s |

Ce premier problème illustre la faisabilité de notre approche, mais ces résultats ne sont pas vraiment intéressants: sur les petites instances, Ilog solveur est bien plus rapide que Ant-P-solveur, tandis que sur les plus grandes instances, les deux solveurs ont des résultats comparables. En fait, le problème des reines est très souvent utilisé pour illustrer des solveurs de CSPs, essentiellement parce qu'il est facile à spécifier et à implémenter. Cependant, comme cela est noté dans [TSA 93], les benchmarks sur ce problème doivent être interprétés avec prudence dans la mesure où il a des caractéristiques très particulières et atypiques. En particulier, chaque valeur pour chaque variable est en conflit avec au plus trois valeurs de chaque autre variable, et cela quel que soit le nombre de reines. Par conséquent, plus il y a de reines, et moins le problème est contraint.

### 3.2. *Le problème des intervalles*

Ce problème est décrit dans la bibliothèque CSPLIB [GEN 99] (cf prob007). Il s'agit de trouver une permutation des $n$ premiers nombres entiers, de telle sorte que les différences absolues de tous les couples consécutifs de nombres soient toutes différentes. Les solveurs de contraintes complets, comme Ilog solveur ou CHIP, utilisant des contraintes globales et des algorithmes de filtrages pour réduire les domaines des variables lors de la recherche, peuvent trouver sans retour-arrière une première solution à ce problème [SIM 98]. Cette solution correspond à la séquence régulière suivante: $< 0,(n-1),1,(n-2),2,(n-3),\ldots >$. Cependant, trouver une autre solution que celle-ci est un problème difficile.

Le tableau suivant donne les résultats obtenus pour résoudre des instances du problème des intervalles, avec un nombre de variables allant de 10 à 24. Pour le programme Ilog, nous avons utilisé deux contraintes globales `IlcDistribute` pour exprimer le fait qu'à la fois les variables et les intervalles doivent avoir des valeurs toutes différentes, et on a cherché une deuxième solution, dans la mesure où Ilog trouve la première solution sans retour arrière.

| | Ant-P-solveur | | Ilog solveur | |
|---|---|---|---|---|
| Nb var | Nb cycles | Temps | Nb pt de choix | Temps |
| 10 | 14 | 0.0s | 93 | 0.1s |
| 12 | 46 | 0.1s | 939 | 1.1s |
| 14 | 101 | 0.5s | 20 718 | 29.3s |
| 16 | 309 | 2.0s | 556 978 | 882.4s |
| 18 | 480 | 3.7s | - | $>$ 3 600.0s |
| 20 | 1 091 | 10.4s | - | $>$ 3 600.0s |
| 22 | 1 796 | 20.6s | - | $>$ 3 600.0s |
| 24 | 4 725 | 65.0s | - | $>$ 3 600.0s |

Sur ce problème, Ant-P-solveur est clairement plus efficace. Ces résultats montrent qu'une approche incomplète, dans la mesure où elle prend en compte les contraintes globales, est mieux adaptée qu'une approche complète pour résoudre ce genre de problème.

### 3.3. *Le problème d'ordonnancement de voitures*

Ce problème est décrit dans [GEN 99] (cf prob001). Il s'agit d'ordonnancer des voitures le long d'une chaine de montage. Les voitures sont groupées par classes de voitures. Toutes les voitures appartenant à une même classe doivent recevoir un même ensemble d'options. A chaque option $i$, est associée une contrainte de capacité, représentée par un ratio $p_i/q_i$, qui spécifie que pour n'importe quelle séquence de $q_i$ voitures consécutives dans la chaine de montage, au plus $p_i$ d'entre elles pourront recevoir l'option $i$. Le but est de trouver une permutation des $n$ voitures à produire qui satisfait toutes les contraintes de capacité.

Pourcentage d'utilisation et heuristiques d'ordre :

La difficulté d'une instance de ce problème dépend du nombre de voitures et du nombre de classes de voitures, mais aussi et surtout du pourcentage d'utilisation des différentes options. Le pourcentage d'utilisation d'une option $i$ correspond à la proportion de voitures nécessitant l'option $i$ par rapport à la capacité maximale pour cette option (i.e., le nombre total de voitures qui pourraient recevoir cette option sans violer de contraintes de capacité) [DAV 99]. Un fort pourcentage d'utilisation, proche de 100%, indique que la demande est très proche de la capacité maximale, et donc que le problème est fortement contraint. Par conséquent, [SMI 96] et [REG 97] ont introduit des heuristiques d'ordre: l'idée est d'affecter en premier les voitures demandant des options ayant un fort pourcentage d'utilisation, correspondant à des options critiques. De telles heuristiques d'ordre sont très facilement incorporables dans Ant-P-solveur: il suffit d'augmenter les capacités d'attraction des sommets correspondant à des voitures qui nécessitent des options critiques, i.e., proportionnellement à leur pourcentage d'utilisation.

Description des résultats obtenus :

On considère cinquante problèmes (produits par J. Lee [LEE 98]), groupés en cinq ensembles de dix instances chacun (appelés respectivement car-60, car-70, car-80, car-85 et car-90), en fonction de leurs pourcentages d'utilisation (respectivement de 60, 70, 80, 85 et 90%). Toutes ces instances sont satisfiables, et ont 200 voitures à ordonnancer, 5 options, et de 17 à 30 classes de voitures. Le tableau suivant donne les résultats obtenus avec Ilog-solveur et Ant-P-solveur. Ant-P-solveur a été limité à 5000 cycles tandis que Ilog-solver a été limité à 3600 secondes d'exécution. Dans ces limites, les solveurs n'ont pas toujours été capables de trouver une solution. Ainsi, on donne, en plus du temps cpu et du nombre de cycles ou points de choix, le pourcentage d'exécutions qui ont effectivement réussi. Pour le programme Ilog, on a utilisé la contrainte globale `IlcSequence`, qui a été conçue pour ce genre de problème, et on a utilisé les heuristiques d'ordre, comme préconisé dans [REG 97]. Pour Ant-P-solveur, on a utilisé des heuristiques d'ordre similaires.

| | Ant-P-solveur | | | Ilog solveur | | |
|---|---|---|---|---|---|---|
| Pb | Réussite | Nb cycles | Temps | Réussite | Nb pt choix | Temps |
| car-60 | 100% | 88 | 1.7s | 40% | 235 | 17.7s |
| car-70 | 100% | 99 | 2.5s | 20% | 234 | 17.8s |
| car-80 | 100% | 140 | 4.4s | 60% | 289 | 19.0s |
| car-85 | 98% | 942 | 33.1s | 50% | 315 | 23.5s |
| car-90 | 57% | 3 187 | 119.8s | 80% | 2 783 | 229.7s |

Sur les instances “faciles” de ce problème, avec des pourcentages d’utilisation variant de 60% à 80%, on remarque que Ant-P-solveur est nettement plus efficace et a un plus fort taux de réussite: Ant-P-solveur n’échoue jamais, tandis que Ilog-solver échoue sur plus de la moitié des instances. En effet, ces instances étant relativement peu contraintes, l’algorithme de filtrage utilisé par Ilog ne réduit pas suffisamment l’espace de recherche. Dans ce cas, un solveur incomplet est clairement plus approprié. En contrepartie, pour les instances les plus difficiles, avec un pourcentage d’utilisation de 90 %, l’algorithme de filtrage devient efficace, et permet donc de résoudre plus de problèmes que Ant-P-solveur, tandis que les temps d’exécution deviennent comparables.

Problèmes surcontraints :

Ant-P-solveur explore l’espace de recherche de façon stochastique et incomplète, et ne permet donc pas de prouver l’inconsistance de problèmes insatisfiables. En revanche, dans le contexte général de problèmes surcontraints, appelés “max-CSPs”, Ant-P-solveur peut être utilisé pour chercher une affectation complète qui maximise le nombre de contraintes satisfaites. Considérons par exemple l’instance `pb19/71` du problème d’ordonnancement de voitures, extraite de [GEN 99]. La consistance de cette instance est restée une question ouverte dans [REG 97], i.e., Ilog solveur ne peut ni trouver une solution, ni prouver l’inconsistance de cette instance en un temps raisonnable. Dans [GEN 98], I. Gent prouve que cette instance est effectivement inconsistante et que toute affectation complète viole au moins deux contraintes de capacité. Sur cette instance, Ant-P-solveur a trouvé une affectation qui viole deux contraintes de capacité, en 969 cycles et 81 secondes. Cette affectation est effectivement optimale, même si Ant-P-solveur ne peut le prouver.

## 4. Comparaison avec d’autres approches

### 4.1. *Contraintes globales vs contraintes locales*

D’une façon générale, pour résoudre un CSP contenant des contraintes de permutation, il vaut mieux utiliser des contraintes globales, qui permettent de les traiter plus efficacement. En effet, [DAV 99] montre que l’utilisation de contraintes globales réduit toujours l’espace de recherche (de $n^n$ à $n!$), même si cela peut parfois prendre plus de temps pour effectivement résoudre le problème.

L’intérêt d’utiliser des contraintes globales est bien illustré par le problème des intervalles. En effet, [HOO 98] décrit la formulation de ce problème sous la forme d’un problème SAT, et l’utilise pour comparer les principaux algorithmes de recherche locale. Ces résultats montrent que l’instance SAT encodant le problème des intervalles à 12 variables cause les plus grandes difficultés, même aux meilleurs algorithmes de recherche locale pour SAT. En fait, la difficulté ne provient probablement pas du type d’approche considéré, mais du fait que les contraintes globales de permutation sont perdues par l'encodage SAT, de telle sorte que l'espace de recherche à explorer devient bien plus important.

### 4.2. *Traitement des contraintes globales*

De nombreux solveurs de contraintes ont introduit des contraintes globales. La façon de prendre en compte ces contraintes globales dépend du type d'approche considérée, i.e., approches complètes, par réparations locales ou génétiques.

Approches complètes :

Les solveurs complets explorent l'espace de recherche de façon exhaustive, jusqu'à ce qu'une solution soit trouvée. Afin de réduire l'espace de recherche, des techniques de propagation sont généralement appliquées. L'idée est de vérifier, à chaque nœud de la recherche, que le problème satisfait une certaine consistance partielle (où certaines contraintes ont été relachées). Cette vérification est effectuée par un algorithme de filtrage qui supprime des domaines des variables les valeurs inconsistantes par rapport à la consistance partielle considérée [TSA 93].

Différents algorithmes de filtrage spécialisés ont été définis pour traiter les contraintes globales de permutation, e.g., les contraintes `IlcDistribute` et `IlcSequence` de Ilog solveur [REG 97], ou `cycle` de CHIP [BEL 94]. Ces algorithmes de filtrage permettent effectivement de résoudre certaines instances particulièrement difficiles du problème d'ordonnancement de voitures. En revanche, sur des problèmes moins fortement contraints, comme le problème des intervalles ou les instances moins contraintes du problème d'ordonnancement de voitures, l'espace de recherche n'est pas suffisamment réduit par ces algorithmes de filtrage, et reste trop grand pour pouvoir être exploré de façon complète. Dans ce cas, une approche incomplète, comme celle de Ant-P-solveur, est mieux appropriée.

Approches par réparation locale :

Les solveurs par réparation locale sont incomplets et (généralement) stochastiques. L'idée est de travailler sur des affectations complètes mais inconsistantes, et de les réparer graduellement jusqu'à les rendre consistantes. Sur les problèmes très fortement combinatoires, ces algorithmes trouvent souvent une solution proche de l'optimum assez rapidement. En contrepartie, ils ne peuvent garantir l'optimalité de la solution trouvée, ni prouver l'inconsistance d'un problème.

GENET [DAV 94] est une approche par réparation locale, basée sur une adaptation de l'heuristique "min-conflicts": l'idée est de s'échapper des optima locaux en augmentant le poids des contraintes violées, ce qui permet "d'apprendre" quelles sont les contraintes critiques, plus difficiles à satisfaire. Deux extensions différentes de GENET ont été proposées afin de prendre en compte des contraintes globales de permutation:

– Dans SWAPGENET [DAV 99], l'idée est de sélectionner les réparations possibles à l'intérieur d'un échange entre couples de valeurs, de telle sorte que les affectations successives satisfont toujours la contrainte de permutation. SWAPGENET a été évalué sur le problème d'ordonnancement de voitures. Les temps cpu obtenus semblent comparables à ceux obtenus avec Ant-P-solveur (ils n'ont pas été effectués sur la même machine ni sur les mêmes instances). Rien n'est dit sur les pourcentages de réussite.

– Dans E-GENET [LEE 98], l'idée est de travailler sur des tuples de valeurs, et d'associer des pénalités à ces tuples en fonction du nombre de contraintes satisfaites. Les expérimentations sur le problème d'ordonnancement de voitures montrent que E-GENET a un taux de réussite bien inférieur à celui de Ant-P-solveur, sur les mêmes instances du problème (73% de réussite pour car-80, rien n'est dit pour car-90). Les temps cpu ne sont pas donnés.

Algorithmes génétiques :

Les algorithmes génétiques sont des approches incomplètes et stochastiques inspirées du processus d'évolution naturelle des espèces. L'exploration de l'espace de recherche est effectuée à travers des opérations de sélection, mutation et croisement sur une population d'individus (représentant des affectations complètes candidates).

GAcSP [WAR 95] combine un algorithme génétique avec des techniques de réparation locales pour résoudre le problème d'ordonnancement de voitures. Dans cette approche, les contraintes globales de permutation sont prises en compte par une fonction de réparation, qui assure que les nouveaux individus créés à chaque génération, par croisement, vérifient effectivement les contraintes de permutation. Après cette phase de réparation, chaque individu est optimisé par une approche de "hill-climbing" (proche de celle utilisée dans [DAV 99]). Les expérimentations montrent que GAcSP permet effectivement de résoudre les instances faciles du problème d'ordonnancement de voitures, ayant de faibles pourcentages d'utilisation. Cependant, en augmentant le pourcentage d'utilisation, le taux de réussite chute très fortement: sur des instances avec 200 voitures et un pourcentage d'utilisation de 80%, seulement 9% des exécutions trouvent une solution, en 7289 secondes en moyenne.

## 5. Conclusion

On a décrit dans ce papier Ant-P-solveur, une approche stochastique et incomplète, inspirée par les algorithmes à base de fourmis, permettant de résoudre des problèmes de recherche de permutations sous contraintes.

Ce travail ouvre de nombreuses perspectives de recherche. En particulier, on envisage d'introduire des techniques de réparation, afin d'améliorer les solutions trouvées par les fourmis, et accélérer le processus de convergence. Par ailleurs, un point critique de ce genre d'approche stochastique réside dans le réglage des différents paramètres. Ainsi, nous étudions les possibilités d'intégration de techniques inspirées des algorithmes génétiques afin de permettre aux fourmis de faire évoluer les paramètres en fonction du problème à résoudre [BOT 98]. Enfin, Ant-P-solveur est tout naturellement parallélisable. L'idée est d'exécuter en parallèle plusieurs processus, chacun travaillant sur une matrice de phéromone différente, tandis que les meilleures solutions de chaque processus sont échangées régulièrement. Une approche similaire, faisant coopérer en parallèle des algorithmes génétiques et des algorithmes à base de fourmis, est proposée dans [CAL 99].

Nous proposons par ailleurs dans [PIM 00] une généralisation de cette approche à la résolution de n'importe quel CSP. Le graphe utilisé comme support de la recherche par les fourmis associe un sommet à chaque couple <variable/valeur> correspondant à l'affectation d'une valeur à une variable, tandis que la règle de transition utilise une fonction d'évaluation locale similaire à celle utilisée par Ant-P-solveur. Ce solveur, appelé Ant-solveur, peut être utilisé pour résoudre n'importe quel CSP, de façon générique. On l'illustre plus particulièrement sur le problème de satisfiabilité de formules booléennes (SAT).

## 6. Bibliographie


[BEL 94] BELDICEANU N., CONTEJEAN E., « Introducing Global Constraints in CHIP », *Journal of Mathematical and Computer Modelling*, vol. 20, n° 12, 1994, p. 97–123.

[BOT 98] BOTEE H., BONABEAU E., « Evolving Ant Colony Optimization », *Journal of Adv. Complex Systems - Hermes*, vol. 1, 1998, p. 149–159.

[BUL 99] BULLNHEIMER B., HARTL R., STRAUSS C., « An Improved Ant system Algorithm for the Vehicle Routing Problem », *Annals of Operations Research*, vol. 89, 1999, p. 319–328.

[CAL 99] CALÉGARI P., « Parallelization of population-based evolutionary algorithms for combinatorial optimization problems », PhD thesis, Ecole polytechnique fédérale de Lausanne, 1999.

[DAV 94] DAVENPORT A., TSANG E., ZHU K., WANG C., « GENET: a connectionist architecture for solving constraint satisfaction problems by iterative improvement », *Proceedings of AAAI'94*, 1994, p. 325–330.

[DAV 99] DAVENPORT A., TSANG E., « Solving constraint satisfaction sequencing problems by iterative repair », *Proceedings of the first international conference on the practical applications of constraint technologies and logic programming (PACLP)*, 1999, p. 345–357.

[DOR 92] DORIGO M., « Learning and Natural Algorithms (in italian) », PhD thesis, Politecnico di Milano, 1992.

[DOR 96] DORIGO M., MANIEZZO V., COLORNI A., « The Ant System: Optimization by a Colony of Cooperating Agents », *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, vol. 26, n° 1, 1996, p. 29–41.

[DOR 97] DORIGO M., GAMBARDELLA L., « Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem », *IEEE Transactions on Evolutionary Computation*, vol. 1, n° 1, 1997, p. 53–66.

[DOR 99a] DORIGO M., CARO G. D., GAMBARDELLA L. M., « Ant Algorithms for Discrete Optimization », *Artificial Life*, vol. 5, n° 2, 1999, p. 137–172.

[DOR 99b] DORIGO M., DI CARO G., « The Ant Colony Optimization Meta-Heuristic », CORNE D., DORIGO M., GLOVER F., Eds., *New Ideas in Optimization*, p. 11–32, McGraw Hill, UK, 1999.

[GAM 99] GAMBARDELLA L., TAILLARD E., DORIGO M., « Ant Colonies for the Quadratic Assignment Problem », *Journal of the Operational Research Society*, vol. 50, 1999, p. 167–176.

[GEN 98] GENT I., « Two Results on Car-sequencing Problems », Technical report (http://www.apes.cs.strath.ac.uk/apesreports.html), 1998, APES.

[GEN 99] GENT I., WALSH T., « CSPLib: a benchmark library for constraints », rapport, 1999, APES-09-1999, available from http://csplib.cs.strath.ac.uk/. A shorter version appears in CP99.

[HOO 98] HOOS H., « Stochastic Local Search - Methods, Models, Applications », PhD thesis, TU Darmstadt, 1998.

[ILO 99] ILOG, « ILOG Solver 4.4 User's Manual », rapport, 1999, Ilog S.A. 9 rue de Verdun, 94253 Gentilly Cedex.

[LEE 98] LEE J., LEUNG H., WON H., « Performance of a Comprehensive and Efficient Constraint Library using Local Search », *11th Australian Joint Conference on Artificial Intelligence*, LNAI, Springer-Verlag, 1998.

[PIM 00] PIMONT S., SOLNON C., « A Generic Ant Algorithm for Solving Constraint Satisfaction Problems », rapport, 2000, LISI.

[REG 97] REGIN J.-C., PUGET J.-F., « A Filtering Algorithm for Global Sequencing Constraints », SMOLKA G., Ed., *CP97*, vol. 1330 de *LNCS*, p. 32–46, Springer-Verlag, 1997.

[SIM 98] SIMONIS H., BELDICEANU N., « A note on CSPLIB prob007 », rapport, 1998, Cosytech.

[SMI 96] SMITH B., « Succeed-first or Fail-first: A Case Study in Variable and Value Ordering Heuristics », *third Conference on the Practical Applications of Constraint Technology PACT'97*, 1996, p. 321–330.

[TSA 93] TSANG E., *Foundations of Constraint Satisfaction*, Academic Press, 1993.

[WAR 95] WARWICK T., TSANG E., « Tackling car sequencing problems using a genetic algorithm », *Journal of Evolutionary Computation - MIT Press*, vol. 3, n° 3, 1995, p. 267–298.