Optimisation par colonie de fourmis pour la configuration

Patrick Albert, Laurent Henocque, Mathias Kleiner

Optimisation par colonie de fourmis pour la configuration

Patrick Albert **Laurent Henocque** **Mathias Kleiner**

Laboratoire LSIS
Université de Saint-Jérôme
13397 Marseille

ILOG S.A
9 rue de Verdun
94253 Gentilly

palbert@ilog.fr,{mathias.kleiner,laurent.henocque}@lsis.org

Résumé

Une des difficultés inhérentes à la recherche énumérative est l'explosion combinatoire. Parmi les algorithmes incomplets qui tentent de résoudre ce problème, l'optimisation par colonie de fourmis (ACO - Ant Colony Optmisation), qui combine des méthodes aléatoires et heuristiques avec l'apprentissage par renforcement, a prouvé son efficacité sur de nombreux problèmes de satisfaction de contraintes (CSP). Cet article présente une application d'un algorithme basé sur ACO pour la configuration, ce qui à notre connaissance n'avait pas encore été étudié. Nous décrivons comment la nature des problèmes non-bornés de configuration influe sur l'approche ACO, notamment à cause de la présence de variables ensemblistes et de domaines ouverts. Nous proposons un algorithme et un modèle phéromonal original permettant de traiter ces difficultés. Nous montrons également l'utilisation de l'optimisation par essaim de particules (PSO) pour converger vers des ensembles de paramètres optimaux. Enfin, nous fournissons des résultats expérimentaux, à la fois pour des instances aléatoires et pour le problème d'optimisation des racks.

Abstract

An inherent difficulty in enumerative search algorithms for optimisation is the combinatorial explosion that occurs when increasing the size of the input. Among incomplete algorithms that address this issue, Ant Colony Optimization (ACO) uses à combination of random and heuristic methods plus reinforcement learning, which proved efficient on à wide range of CSPs problems. This paper presents early results in applying an ACO-based algorithm to configuration, which to the best of our knowledge was never investigated before. We describe how the nature of unbounded configuration problems impacts the ACO approach due to the presence of set-variables with open domains. We propose an ACO framework able to deal with those issues through an original pheromone model and algorithm. We also present the use of Particle Swarm Optimization (PSO) to converge towards good parameter sets. Finally, we provide expérimental results, both for random problem instances and the "racks" optimisation problem.

1 Introduction

Ces recherches ont pour but d'augmenter la taille des problèmes de configuration pouvant être traités par une recherche énumérative. Des domaines d'application récents, comme la configuration de machines, le traitement du langage ou le web sémantique, ont mis à jour la nécessité pour les solveurs de configuration de traiter un grand nombre de composants.

L'espace de recherche de ces problèmes augmente très rapidement avec ce nombre. De plus, dans le cas général, des problèmes ayant un nombre infini de modèles finis peuvent facilement être construits, ce qui signale clairement les limites des méthodes de recherche exhaustives. Une alternative connue dans la communauté SAT/CSP pour juguler cette explosion dans le cas de problèmes satisfiables est l'utilisation de procédures incomplètes. Parmi celles-ci, les approches stochastiques ont mené à des résultats prometteurs.

A notre connaissance, l'utilisation de méthodes incomplètes pour la configuration n'a pas encore été étudiée. Cet article propose une extension générique de l'optimisation par colonie de fourmis aux problèmes de configuration sous contraintes. Nous fournissons des résultats expérimentaux, tant pour la satisfiabilité que pour l'optimisation, sur des problèmes aléatoires et sur un benchmark

historique de la configuration : le problème des "racks".

La section courante est une brève introduction à la configuration et la méta-heuristique ACO. La section 2 détaille un cadre ACO pour la configuration. Nous proposons tout d'abord un modèle phéromonal original qui adapte ACO aux variables ensemblistes sur domaines ouverts. Nous présentons ensuite un algorithme permettant d'exploiter ce modèle pour construire des solutions. La section 3 montre l'utilisation de PSO pour converger vers de bons jeux de paramètres pour notre implémentation, et fournit des résultats expérimentaux. La section 4 conclut et ouvre les perspectives de l'approche.

1.1 Brève introduction à la configuration

Configurer consiste à simuler la création d'un produit complexe à partir de *composants* choisis dans un *catalogue de types*. Dans le contexte général, ni le nombre ni le type des composants requis ne sont connus à l'avance. Les composants sont connectés par des *relations*, et leur type est soumis à l'*héritage*. Des *contraintes* définissent les produits valides. Un configurateur prend en entrée un fragment de la structure ciblée et l'étend à une solution du problème, si une solution existe, en ajoutant tous les éléments nécessaires durant la recherche. Ce problème de logique du premier-ordre est semi-décidable dans le cas général. Le lecteur pourra trouver dans [9] une introduction complète à la configuration.

Nous considérons une approche dans laquelle les relations sont décrites grâce à des *variables de ports* au niveau des composants : une variable de port modélise les composants distants auxquels un objet est connecté pour une relation donnée.

Soit U un ensemble (potentiellement infini) d'identifiants d'objets, appelé *univers* ou *domaine des objets*. Par la suite, les éléments de U seront appelés composants ou objets. U peut être interprété par (ou sans perte de généralité être identifié à) l'ensemble $\mathbb{N}$ des entiers.

définition 1 (modèle de configuration) *Un modèle de configuration est défini par :*

- *trois ensembles mutuellement disjoints T (noms des types), A (noms des attributs) et P (noms des ports)*
- *un ensemble D de toutes les valeurs possibles des attributs*
- *une application $supertypes : T \mapsto \mathbb{P}T$ modélisant l'héritage : pour chaque type $t \in T$, $supertypes(t)$ est l'ensemble de ses parents directs*
- *une application $attributes : T \mapsto \mathbb{P}A$ modélisant la structure d'une classe : pour chaque type $t \in T$, $attributes(t)$ est l'ensemble des attributs de la classe t*
- *une application $ports : T \mapsto \mathbb{P}P$ modélisant les connexions d'une classe : pour chaque type $t \in T$, $ports(t)$ est l'ensemble des ports de la classe t*
- *une application $domain : \mapsto \mathbb{P}D$ modélisant les domaines d'un attribut : pour chaque attribut $a \in A$, $domain(a)$ est le domaine des variables CSP modélisant chaque occurrence de l'attribut dans les objets*
- *une application $type : P \mapsto T$ modélisant le type des objets pouvant être connectés via un port donné*
- *une application $card_{min} : P \mapsto \mathbb{N}$ et une fonction partielle $card_{max} : P \mapsto \mathbb{N}$ modélisant la cardinalité minimum (et éventuellement maximum) des ports*
- *un ensemble de contraintes C : prédicats de premier ordre ou supérieur sur le vocabulaire $T \cup A \cup P \cup U \cup D$*

Pour chaque objet $o \in U$: le type de o est modélisé par une variable CSP $t(o)$ sur le domaine T (ou $\mathbb{P}T$ dans certaines implémentations) ; chaque attribut $a \in attributes(t(o))$ est modélisé par une variable CSP $a(o)$ sur le domaine $domain(a)$; chaque port $p \in ports(t(o))$ est modélisé par une variable ensembliste CSP $p(o)$ sur le domaine $\mathbb{P}U$ sous la contrainte que chaque élément de $p(o)$ appartient à $type(p)$. Par la suite, nous appellerons port, attribut ou type la variable CSP correspondante d'un objet lorsque cela n'est pas ambigu.

définition 2 (Configuration, instance de configuration) *Un ensemble de composants (un sous-ensemble de U) dont le type, les attributs et les ports sont instanciés et tel que toutes les contraintes de C soient satisfaites est appelé une* instance de configuration *ou plus simplement une* configuration *du modèle de configuration.*

D'un point de vue logique, une configuration est un *modèle* du modèle de configuration. Pour éviter la confusion nous utiliserons donc les termes *instance* ou *configuration*.

définition 3 (problème de configuration) *Un problème de configuration est composé :*

- *d'un modèle de configuration CM,*
- *d'une requête R contenant un ensemble de contraintes additionnelles et/ou de préférences.*

Un configurateur ou procédure de résolution *doit produire un ou plusieurs modèles de $CM \wedge R$ si il en existe.*

Résoudre le problème d'énumération associé peut se faire en utilisant différents formalismes ou approches techniques : extensions du paradigme CSP [13, 18, 1, 14], approches à base de connaissances [17], logiques terminologiques (ou de description) [2, 12], programmation logique (chaînage avant ou arrière, et sémantique non standard) [15], approches orientées objet [10].

Ces techniques ont été utilisées avec succès sur un certain nombre de problèmes industriels. Cependant ces approches peuvent difficilement traiter l'explosion combinatoire qui accompagne l'augmentation des données en entrée. De plus, la plupart d'entre elles posent des restrictions

sur le contexte de configuration, en bornant le nombre de composants disponibles et/ou les cardinalités des relations.

Les approches basées sur des extensions des CSPs, telles que les CSP génératifs [18, 7], peuvent quant à elles traiter un contexte de configuration non borné :

- l'instantiation du type et des attributs d'un composant est modélisée par une variable CSP classique : choix d'une valeur unique dans un domaine fini et discret
- l'instantiation d'un port de composant peut susciter la génération dynamique de nouveaux composants afin de satisfaire les contraintes de cardinalité (min). La cardinalité d'un port $card(p)$ consiste en un choix de valeur unique dans un domaine discret et potentiellement non borné. Les cibles sont choisies dans un ensemble de composants du type destination $type(p)$, modélisé par une variable ensembliste sur un domaine également non borné. Le choix des cibles peut se faire en sélectionnant itérativement $card(p)$ valeurs différentes dans l'ensemble des composants de type $type(p)$, après en avoir éventuellement introduit de nouveaux.

Une procédure de recherche dans le cas des CSP génératifs démarre avec un ensemble de composants à instancier, contenant au moins un composant *racine*. Chaque fois qu'un composant est sélectionné, ou créé dynamiquement à travers l'instantiation d'un port, il est ajouté à la liste. Pour finir sur cette introduction, les problèmes de configuration, tout comme les CSPs, peuvent être des problèmes de satisfaction ou d'optimisation.

1.2 Brève introduction à ACO

Les recherches sur le comportement des colonies de fourmis ont montré que leur communication est essentiellement basée sur la production et la détection d'agents chimiques appelés *phéromones*. Parmi les différents types de phéromones, certaines sont déposées sur le sol. Un chemin marqué par de telles phéromones, par exemple d'une source de nourriture au nid, est alors suivi par les autres fourmis avec des fluctuations qui sont aléatoires ou dues à la manière dont la fourmi perçoit son environnement et y réagit (sa "visibilité locale"). Ces phéromones sont également volatiles. Leur évaporation progressive permet de diversifier les chemins empruntés ou d'en explorer de totalement nouveaux. Des expérimentations biologiques et des résultats théoriques ont montré que les fourmis utilisent ces phéromones pour trouver le chemin le plus court vers une source de nourriture, et donc de ce fait résolvent un problème d'optimisation.

Déterminer le chemin le plus court est en effet une tâche similaire à de nombreux problèmes combinatoires tels que le problème du voyageur de commerce. Ceci a conduit les chercheurs à s'inspirer du comportement des fourmis pour élaborer de nouveaux algorithmes de résolution [5]. L'approche fut ensuite étendue à une *méta-heuristique* pour les *problèmes d'optimisation discrets* [4] connue sous le nom de *Ant Colony Optimization*. [3] présente une étude récente et complète d'ACO.

1.2.1 Méta-heuristique ACO et algorithmes

Marco Dorigo propose dans [4] une méta-heuristique ACO et un algorithme (*Ant System*) pour les problèmes d'optimisation discrets combinatoires. Il existe de nombreux travaux sur des variantes d'*Ant System*. Une des améliorations les plus connues est *MAX-MIN Ant System* (MMAS) [19]. La modélisation du problème est associée dans ACO à un modèle phéromonal : une valeur phéromonale est associée à chaque affectation possible d'une valeur à une variable X_i. Formellement, la valeur phéromonale τ_{ij} est associée au *composant de solution* c_{ij} , qui consiste en une affectation $X_i = v_i^j$. L'ensemble de tous les composants de solution potentiels est appelé C.

Une fourmi artificielle construit une solution à partir d'une instantiation partielle vide $s^p = \emptyset$. Elle est étendue à chaque étape de la construction en ajoutant un composant de solution c_{ij} de l'ensemble $N(s^p) \subseteq C$ où $N(s^p)$ est l'ensemble des composants pouvant être ajoutés sans violer les contraintes du problème. Le choix du composant est un choix probabiliste influencé par les phéromones et les heuristiques (correspondant à la visibilité locale d'une fourmi biologique). Cette probabilité peut varier en fonction des algorithmes ACO. Dans *Ant System*, la probabilité de choisir un composant c_{ij} pour une fourmi k est définie par :

$$p_{ij}^k = \frac{\tau_{ij}^{\alpha}.\eta_{ij}^{\beta}}{\sum_{c_{il} \in N(s^p)} \tau_{il}^{\alpha}.\eta_{il}^{\beta}}$$

où les paramètres α and β contrôlent l'importance relative des phéromones par rapport à l'information heuristique η_{ij}.

Un *graphe de construction* $G_C(V, E)$ peut être obtenu à partir de C, dans lequel les sommets V et les arêtes E sont les composants de solution. Une fourmi artificielle construit une solution en traversant ce graphe (complet) de construction de sommet en sommet. De plus, les fourmis déposent une quantité $\Delta\tau$ de phéromones sur les composants dépendant de la qualité de la solution.

Les ACO sont utilisés pour résoudre des CSP dans [16]. L'algorithme présenté inclut des améliorations provenant de la variante $MAX - MIN$. Cependant, contrairement aux ACO classiques, le choix probabiliste ne dépend pas de la dernière variable instanciée, car tous les sommets déjà visités sont d'égale importance. L'heuristique de choix de valeur est inversement proportionnelle au nombre de contraintes violées. L'heuristique de choix de variable est classique en optimisation, celle du plus petit domaine.

2 ACO pour la configuration

Dans le contexte général de la configuration, nous avons vu que la modélisation s'appuie en partie sur des variables ensemblistes à domaines ouverts. Ces problèmes n'entrent donc pas dans le cadre de la définition des ACO, et ne peuvent pas non plus être résolus avec une approche CSP standard. En particulier, il n'est pas possible d'exhiber a priori un graphe de construction puisque le nombre de composants d'une solution n'est pas borné. Toutefois, il existe des similitudes, et un graphe utile peut être identifié : le graphe d'interconnexion des composants. Cela fonde une intuition majeure de notre approche.

Graphe de construction La structure d'une (instance de) configuration peut être vue comme un graphe. Dans ce *graphe de structure*, les sommets représentent les composants et les arcs les relations qu'ils entretiennent. La superposition de tous les graphes de structures créés lors de précédentes tentatives d'un algorithme à redémarrage aléatoire reste également un graphe. Des phéromones peuvent donc être déposées sur les arcs, qui représentent alors pour une fourmi artificielle le choix d'un composant (déjà existant) en tant que cible d'un port.

Cependant, à un sommet donné de ce graphe, une fourmi a non seulement la possibilité de suivre un ou plusieurs arcs, mais peut également créer des arcs vers de nouveaux composants quand la décision est prise de générer dynamiquement une nouvelle cible pour le port considéré. Le nombre d'arcs (i.e les cibles d'un port), suivis ou créés par une fourmi, est défini par le choix (généralement non borné) de la cardinalité du port.

Enfin, puisque chaque sommet est un composant de la configuration, les fourmis doivent le classifier (i.e sélectionner son type) et choisir une valeur pour chacun de ses attributs (i.e configurer le composant). Nous avons vu que ces décisions peuvent être modélisées par l'instantiation de variables CSP classiques. Les types et les attributs peuvent être ajoutés au graphe de construction en tant que sommets connectés au sommet du composant. Cette dernière représentation, similaire à celle proposée par [16], permet de fournir un point de vue homogène sur la construction d'une solution.

Le graphe de construction d'ACO pour la configuration, obtenu par combinaison de tous ces éléments, est donc généré dynamiquement durant la recherche par superposition de toutes les instances précédemment créées. Les fourmis artificielles peuvent, guidées par les phéromones et les heuristiques, explorer et étendre ce graphe afin de construire des solutions potentielles.

Instantiation des variables Dans ACO, chaque décision prise par une fourmi est associée à une variable sur un domaine discret. Ce principe peut également être appliqué à la configuration dans le cas des variables à domaines fermés (CSP ou ensembliste). Cependant, en configuration, certaines variables (les ports et leurs cardinalités) peuvent avoir un domaine ouvert. Nous introduisons un mécanisme original pour traiter le choix probabiliste associé : les *simu-finite sets*.

simu-finite sets Nous simulons le choix d'une valeur dans un ensemble non borné via un ensemble fini *évolutif* appelé *simu-finite set*. Un *simu-finite set* contient un certain nombre de valeurs du domaine, plus une valeur *évolutive* (ou *wild card*) représentant toutes les autres valeurs possibles. L'ensemble évolue (i.e est augmenté pour la prochaine itération ACO) chaque fois qu'une fourmi choisit la valeur *wild card*. L'idée d'utiliser de tels *wild cards* au premier ordre est bien connue.

Par exemple, considérons le port p, $card_{min}(p) = 2$, et le domaine associé aux choix de la cardinalité $card(p)$. Soit l'ensemble de départ $p_{card} = \{2, 3, 4, 5\}$, ou 5 est la *wild card*. Si la valeur choisie est inférieure à 5, l'ensemble reste inchangé. Si la valeur 5 est choisie, l'ensemble est modifié pour la prochaine itération en $p_{card} = \{2, 3, 4, 5, 6\}$ avec la valeur 6 comme nouvelle *wild card*. Cette *wild card* représente donc en intention l'infinité potentielle des composants disponibles.

Nous présentons d'abord une définition générale des *simu-finite sets*. Nous montrerons ensuite son application au choix de la cardinalité et des cibles d'un port dans le modèle phéromonal.

définition 4 (*simu-finite sets*) *Soit la sélection d'une valeur v dans l'ensemble fini $V = \{v_0, \ldots, v_i, v_*\}$ où $\{v_0, \ldots, v_i\}$ est un ensemble fini de valeurs différentes tel que :*
(1) il est possible d'exhiber une valeur $v_{i+1} \notin V$ appartenant au domaine ouvert de la variable considérée
(2) quand la valeur v_* est choisie, l'ensemble est modifié en $V = V \cup \{v_{i+1}\} = \{v_0, \ldots, v_{i+1}, v_*\}$ pour la prochaine itération.

2.1 Modèle phéromonal

Nous proposons un modèle phéromonal original permettant d'utiliser la méta-heuristique ACO dans le cadre de la configuration. Nous présentons d'abord comment exploiter les *simu-finite sets* pour les décisions associées à l'instantiation des ports. Nous complétons ensuite le modèle avec les autres types de décisions (classification et attributs). Enfin, nous détaillons comment les phéromones de ce modèle sont mises à jour à chaque itération d'ACO.

2.1.1 Instantiation d'un port

L'approche consiste, pour un port p, à choisir d'abord une cardinalité $card(p)$ puis à choisir itérativement le

même nombre de cibles. Nous introduisons donc un *simu-finite set* pour le choix d'une cardinalité (non bornée) et un autre pour le choix d'une cible (lorsqu'il est possible de créer dynamiquement des composants).

Choix de la cardinalité le choix d'une cardinalité non-bornée est le choix d'une valeur parmi un ensemble non borné d'entiers. Nous appliquons les *simu-finite sets* comme suit :

(1) Soit la sélection d'une valeur $card(p)$ dans l'ensemble fini

$p_{card} = \{card_{min}(p), \ldots, card_{min}(p) + i, card(p)_*\}$ où :

- $card_{min}(p)$ est la borne inférieure de la cardinalité pour le port p,
- $\{card_{min}(p), \ldots, card_{min}(p) + i\}$ est un intervalle d'entiers, i.e $p_k = p_{k-1} + 1$
- $i >= 0$ est une valeur d'initialisation choisie au départ,
- $card(p)_* = card_{min}(p) + i + 1$ est la *wild card* du *simu-finite set*.

(2) quand la cardinalité $card(p)_*$ est choisie, l'ensemble est modifié en $p_{card} = \{r_{card_{min}}, \ldots, card_{min}(p) + i + 1, card(p)_*\}$ avec $card(p)_* = card_{min}(p) + i + 2$ comme nouvelle *wild card*.

Choix d'une cible le caractère non borné de l'ensemble associé au choix d'une cible provient de la possibilité de créer dynamiquement des composants. Nous appliquons les *simu-finite sets* comme suit :

(1) Soit la sélection d'une valeur p_{t_i} dans l'ensemble fini $p_t = \{p_{t_0}, \ldots, p_{t_i}, p_{t_*}\}$ ou :

- $\{p_{t_0}, \ldots, p_{t_i}\}$ est l'ensemble des composants *existants* qui peuvent être cibles du port p
- p_{t_*} est le choix de créer un nouveau composant (i.e la *wild card*)

(2) Quand la cible p_{t_*} est choisie, l'ensemble est modifié en $p_t = \{p_{t_0}, \ldots, p_{t_i}, p_{t_{i+1}}, p_{t_*}\}$ où $p_{t_{i+1}}$ est le nouveau composant généré, et p_{t_*} est le choix de créer un nouveau composant. De plus, on peut décider de n'autoriser la création que d'un (ou un nombre limité de) composant(s) durant une itération (afin de favoriser l'intensification). Dans ce cas l'ensemble modifié ne contiendra pas la valeur p_{t_*}. Enfin, puisqu'une cible ne peut être choisie qu'une seule fois pour un port donné, elle est supprimée de l'ensemble jusqu'à ce que le port soit entièrement configuré. Le choix probabiliste associé à ces deux *simu-finite sets*, maintenant équivalent au choix d'une valeur unique dans un domaine fini, est obtenu grâce à l'approche standard ACO. Nous définissons donc l'instantiation d'un port pour un problème de configuration dans le modèle phéromonal, suivi des autres types de décisions qui devront être prises par les fourmis artificielles.

2.1.2 Modèle phéromonal pour les ports

définition 5 *Instantiation d'un port*

- *Instancier un port p consiste à choisir une cardinalité p_{card_j} dans le* simu-finite set p_{card}*, puis à choisir p_{card_j} cibles depuis le* simu-finite set p_t.
- *la probabilité de choisir une cardinalité p_{card_j} est :*

$$p_{p_{card_j}} = \frac{\tau^{\alpha}_{p_{card_j}}.\eta^{\beta}_{p_{card_j}}}{\sum_{l=0}^{min+i}(\tau^{\alpha}_{p_{card_l}}.\eta^{\beta}_{p_{card_l}}) + \tau^{\alpha}_{p_{card_*}}.\eta^{\beta}_{p_{card_*}}}$$

où $\tau_{p_{card_j}}$ est la valeur phéromonale pour la cardinalité $card_j$ et $\eta_{p_{card_j}}$ l'information heuristique.

- *la probabilité de choisir une cible p_{t_j} est de :*

$$p_{p_{t_j}} = \frac{\tau^{\alpha}_{p_{t_j}}.\eta^{\beta}_{p_{t_j}}}{\sum_{k=0}^{i}(\tau^{\alpha}_{p_{t_k}}.\eta^{\beta}_{p_{t_k}}) + \tau^{\alpha}_{p_{t_*}}.\eta^{\beta}_{p_{t_*}}}$$

où $\tau_{p_{t_j}}$ est la valeur phéromonale pour la cible p_{t_j} et $\eta_{p_{t_j}}$ l'information heuristique.

2.1.3 Modèle phéromonal pour la classification

Puisque la classification est un choix dans un ensemble fini d'éléments (l'ensemble des sous-types), il est possible d'utiliser l'approche standard.

définition 6 *Classification*

- *Classifier un composant de type t_i consiste à choisir un sous-type depuis l'ensemble $finalsubtypes(t_i) = \{t_i^j, \ldots, t_i^k\}$*
- *la probabilité de choisir un type t_i^j est de :*

$$p_{t_i^j} = \frac{\tau^{\alpha}_{t_i^j}.\eta^{\beta}_{t_i^j}}{\sum_{l=0}^{k}\tau^{\alpha}_{t_i^l}.\eta^{\beta}_{t_i^l}}$$

où $\tau_{t_i^j}$ est la valeur phéromonale pour le type t_i^j et $\eta_{t_i^j}$ l'information heuristique.

2.1.4 Modèle phéromonal pour l'instantiation des attributs

Ici encore, nous avons un choix standard dans un ensemble fini.

définition 7 *Instantiation d'un attribut*

- *Instancier une variable X_i consiste à choisir une valeur dans l'ensemble $D_i = \{x_i^0, \ldots, x_i^k\}$*
- *la probabilité de choisir une valeur x_i^j est de :*

$$p_{x_i^j} = \frac{\tau^{\alpha}_{x_i^j}.\eta^{\beta}_{x_i^j}}{\sum_{l=0}^{k}\tau^{\alpha}_{x_i^l}.\eta^{\beta}_{x_i^l}}$$

où $\tau_{x_i^j}$ est la valeur phéromonale pour la valeur x_i^j et $\eta_{x_i^j}$ l'information heuristique.

2.1.5 Mise à jour des phéromones

La mise à jour des phéromones, de manière similaire à l'approche de $MAX-MINAntSystem$, est basée sur la meilleure instance créée par la colonie de fourmis à chaque itération :

$$\tau_{ij} \longleftarrow \left[(1-\rho).\tau_{ij}^{best} + \Delta\tau_{ij}^{best}\right]_{\tau_{min}}^{\tau_{max}}$$

où $\Delta\tau_{ij}^{best}$ est une valeur dépendant de la qualité de la solution. Dans le cas de la satisfiabilité, la qualité peut être définie par le rapport $\frac{numberOfConstraintsFulfilled}{totalNumberOfConstraints}$.

Cependant, des questions originales se posent pour les problèmes de configuration. Nous introduisons donc dans ce qui suit un certain nombre de mécanismes supplémentaires, dont les effets positifs ont été étudiés par de nombreuses expérimentations.

Premièrement, l'intérêt de l'évaporation est d'oublier les *mauvais* choix (qui ne participent pas ou peu aux meilleures instances ultérieures). Mais si un composant c_i ne participe pas à la meilleure instance courante, le bénéfice d'évaporer les phéromones qui lui sont associées n'est pas évident. Nous proposons donc une *évaporation restreinte* comme alternative à l'évaporation totale standard. Avec l'évaporation restreinte, les phéromones associées à c_i (classification, attributs et ports ayant c_i comme source) ne sont pas modifiées. Les phéromones associées à un port $c_j \rightarrow c_i$ sont tout de même évaporées, prenant ainsi en compte le fait que le composant n'a pas été choisi comme cible du port dans la meilleure instance courante.

Deuxièmement, nous devons considérer les phéromones associées aux *wild cards* des *simu-finite sets*. Dans le cas des cibles d'un port, la valeur associée à la création d'un composant n'est mise à jour que si la meilleure solution a effectivement créé une cible durant l'itération. La valeur modifiée est alors également utilisée pour initialiser la valeur associée au nouveau composant de l'ensemble. Dans le cas du choix de cardinalité, toutes les valeurs ajoutées dynamiquement à l'ensemble partagent la même valeur d'initialisation (définie par un paramètre).

Pour finir, différents types de choix peuvent nécessiter des paramètres différents. Chaque paramètre relatif aux phéromones est donc dupliqué pour chaque type de décision. Par exemple, la valeur minimum autorisée pour une phéromone peut être différente pour la classification ou pour le choix des cibles d'un port.

2.2 Algorithmes

Nous proposons un algorithme ($ACOC_{graph}$) qui exploite le modèle phéromonal présenté pour construire des solutions aux problèmes de configuration. $ACOC_{graph}$ est proche de l'algorithme ACO originel, mais diffère sur la façon dont les fourmis construisent les solutions. La partie commune (la méta-heuristique ACO) est présentée dans le tableau 1.

```
function instance configure
  for i := 0 to numberOfiterations do
    for j := 0 to numberOfAnts do
      instance:=generateInstance
      quality:=evaluateInstance
      if (quality>bestQuality) do
        bestInstance:=instance
        bestQuality:=quality
    updatepheromones
    updateSimuFiniteSets
  return bestInstance
```

TAB. 1 – La fonction **configure**

$ACOC_{graph}$ démarre par la classification d'un composant racine. Les attributs et les ports de ce composant sont instanciés puis les cibles des ports sont classifiées. Chaque cible est ensuite traitée récursivement (parcours en profondeur de la structure). L'algorithme est présenté dans le tableau 2.

La figure 1 donne un exemple graphique de création d'une structure. Un nombre représente un composant classifié et un "+" représente un composant dont les attributs ont été instanciés.

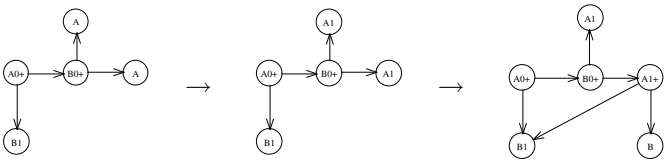

FIG. 1 – $ACOC_{graph}$: instantiation des attributs et des ports, classification des cibles, itération sur les nouveaux composants ajoutés

2.2.1 Chemin de construction

Nous avons montré que le graphe de construction de notre approche n'est pas calculé à l'avance mais défini par la superposition de toutes les instances générées par les fourmis précédentes. Une autre différence avec la méta-heuristique ACO originelle est le *chemin de construction* suivi par les fourmis à l'intérieur de ce graphe.

Dans $ACOC_{graph}$, le chemin de construction est lié à la recherche en profondeur effectuée sur la partie structurelle du graphe de construction (i.e les composants et les ports). Cependant, à chaque sommet, une fourmi va suivre (ou créer) une arête puis retourner au sommet précédent jusqu'à ce que toutes les décisions liées à ce composant soient prises et que les cibles des ports aient été configurées. Ce comportement est illustré sur la figure 1. Une fois que les composants "B" (partie basse) ont été configurés,

```
function instance generateInstance
    instance=initializeInstance
    constructionStack :=root
    classifyRoot
    while (constructionStack≠ ∅) do
        constructObject
        removeObjectFromConstructionStack
    return instance
procedure constructObject
    for i := 0 to numberOfAttributes do
        instantiateAttribute
    for i := 0 to numberOfPorts do
        instantiatePort
        classifyTargets
        addTargetsToConstructionStack
    return
```

TAB. 2 – La fonction **generateInstance pour** $ACOC_{graph}$

la fourmi va retourner au composant central “B0+" et se déplacer vers le composant “A1" (partie haute).

Les choix des fourmis artificielles sont également indépendants des décisions précédentes effectuées sur l'instance courante. Dans ce sens notre approche est similaire à l'application d'ACO aux CSPs [16].

3 Implémentation et expérimentations

Le programme utilisé pour nos expérimentations a été développé en langage Java et inclut une librairie de modèles objets contraints destinée à comparer différentes méthodes de recherche sur une plate-forme commune. Ces expérimentations ont été conduites sur un Pentium IV 3.2 Ghz / 1go RAM. Nous ne décrivons que les résultats les plus significatifs issus d'une large gamme d'expérimentations.

3.1 Heuristiques

Dans ces expérimentations, nous n'avons pas utilisé d'heuristiques de valeurs. Les fourmis artificielles sont donc uniquement guidées par les phéromones. Concernant les heuristiques de choix de variables, l'ordre d'instantiation est partiellement induit par la recherche en profondeur dans le graphe de construction. Les variables de classification, d'attributs et de ports sont traitées dans un ordre aléatoire prédéfini.

3.2 Paramètres

Les paramètres ont une grande influence sur le comportement des algorithmes ACO. Ils sont soit liés à l'algorithme (nombre d'itérations, nombre de fourmis, heuristiques, satisfiabilité ou optimisation) soit aux phéromones (valeurs d'initialisation, taux d'évaporation, poids des phéromones par rapport aux heuristiques, paramètres propres aux *simu-finite sets*). Nous considérons les paramètres suivants :

- nbIte pour le nombre maximum d'itérations,
- nbAnts pour le nombre de fourmis à chaque itération,
- pMax (pMin) pour la valeur maximum (minimum) d'augmentation des phéromones,
- ρ pour le taux d'évaporation,
- evaS pour le choix entre évaporation totale ou restreinte,
- ObjBV,ObjMin,ObjMax pour respectivement les valeurs d'initialisation, minimum et maximum des phéromones associées aux ports,
- RelBV(...),AttBV(...),ClassBV(...) : idem pour les cardinalités des ports, les attributs et les types,
- NOBV pour la valeur d'initialisation des *wild cards* associées aux ports.

Les paramètres constituent un problème récurrent dans les ACO, et sont liés à l'équilibre entre exploration et intensification. En configuration, en plus de s'appliquer aux attributs et types des objets participants (similaires à une affectation dans les CSP), ils influencent également la croissance en taille des solutions. Ce point est primordial puisque les structures sont de taille a priori inconnue. Le grand nombre de paramètres est également une difficulté et requiert des techniques avancées pour leur identification et leur optimisation. Bien sûr, on tente d'isoler des jeux de paramètres qui possèdent une portée générale et qui ne sont pas spécifiques d'un problème donné.

3.2.1 Optimisation par essaim de particules

L'espace multidimensionnel des jeux de paramètres est tellement vaste qu'il est impossible de l'explorer exhaustivement par programme. Nous avons choisi d'utiliser l'optimisation par essaim de particules (PSO - *Particle Swarm Optimization*) : une technique d'optimisation stochastique à base de population. Elle fut développée pour la première fois dans [6]. Intuitivement, chaque jeu de paramètres définit la position d'une particule, qui se déplace dans l'espace avec une certaine vitesse. La meilleure des particules joue le rôle de pôle d'attraction, et infléchit les trajectoires des autres, qui restent partiellement attirées par leur meilleure position passée. On vise ainsi à converger vers des optimums locaux des jeux de paramètres. Un *problème PSO* est défini par une fonction objectif sur un espace multidimensionnel $f : \mathbb{R}^m \rightarrow \mathbb{R}$, avec $[min_j, max_j]$, $0 <= j < m$ comme borne des domaines pour chaque dimension j. Soit n *particules* p_i, $0 <= i < n$. Chaque particule représente une solution potentielle et est définie par une *position* $x_i \in \mathbb{R}^m$ et une *vitesse* $v_i \in \mathbb{R}^m$. La position est l'affectation courante. La vitesse est la direction courante de la particule dans l'espace du problème. Chaque particule a

la connaissance de sa meilleure position et de la meilleure position globale.

Le tableau 3 présente un algorithme PSO. ω est une constante d'inertie, généralement de valeur légèrement inférieure à 1 et décroissante au cours du temps. $c1$ et $c2$ sont des constantes contrôlant l'attrait des positions optimum pour une particule. Elles sont respectivement appelées composantes "cognitive" et "sociale". Les valeurs $c1 = c2 = 2$ sont habituellement retenues. $r1$ et $r2$ sont des nombres aléatoires dans $[0, 1]$. $x_{i,j}$ (la position d'une particule) est généralement initialisée aléatoirement, alors que $v_{i,j}$ (sa vitesse) est initialisée à 0. Les *conditions d'arrêt* sont déterminées ici par un nombre donné d'itérations de l'algorithme.

```
procedure pso
  for i := 0 to n − 1 do
    initialize x_i and v_i
  initialize x̂_i and ĝ
  while terminationConditionsNotMet do
    for i := 0 to n − 1 do
      for j := 0 to m − 1 do
        v_{i,j} = ω * v_{i,j} + c1 * r1 * (x̂_{i,j} − x_{i,j}) +
          c2 * r2 * (ĝ_j − x_{i,j})
        x_{i,j} = x_{i,j} + v_{i,j}
        if ( f(x_{i,j}) > f(x̂_{i,j}) ) then x̂_{i,j} = x_{i,j}
        if ( f(x_{i,j}) > f(ĝ_j) ) then ĝ_j = x_{i,j}
  return
```

TAB. 3 – Un algorithme **PSO**

Utilisation de PSO pour le choix des paramètres ACO Nous avons appliqué PSO de manière directe pour trouver les meilleurs paramètres d'$ACOC_{graph}$. Chaque paramètre est une dimension du problème PSO, les domaines pouvant être discrets (booléens, entiers) ou continus (flottants). La fonction objectif est une combinaison de la qualité de la solution obtenue par $ACOC_{graph}$ avec ces paramètres et du temps de calcul de cette solution.

3.3 Résultats expérimentaux et analyse

Nous présentons des expérimentations sur des problèmes aléatoires et sur un benchmark de configuration de la littérature (problème des racks) qui est très structuré. Ces deux problèmes ont des propriétés fondamentalement différentes. Pour chaque type de problème, nous avons utilisé l'algorithme PSO, avec 20 particules, pour trouver des paramètres efficaces. Il va de soi que les points de convergence des jeux de paramètres ainsi déterminés sont potentiellement spécifiques au problème. En appliquant l'approche à des problèmes de natures différentes, il devient donc possible de s'approcher de jeux de paramètres génériques. Dans les expérimentations qui suivent, *nbIte*, *nbAnts*, *NOBV* et toutes les valeurs maximum ont été fixées. rho, *evaS*, valeurs minimum et d'initialisation, *pMin* et *pMax* varient avec chaque particule.

3.3.1 Problèmes aléatoires

Nous avons généré des problèmes aléatoires prenant en compte : le nombre de types de composants, la densité des relations (probabilité d'avoir une relation entre deux classes), et la difficulté des relations (écart moyen entre cardinalités maximum et minimum). Les problèmes n'ont pas de contraintes additionnelles et sont créés de telle manière qu'une solution finie existe. Nous considérons donc ici un problème de satisfaction. Nous avons généré 10 problèmes aléatoires hétérogènes, qui sont tous résolus 200 fois par une particule dans une position donnée. Nous montrons ici la moyenne des résultats obtenus sur ces 200 essais. Le tableau 4 montre les résultats de la meilleure particule après 50 itérations de PSO.

3.3.2 Problème des racks

Le problème des racks est un benchmark d'optimisation pour la configuration[1] dans lequel des cartes doivent être branchées à des racks. L'objectif est de minimiser le coût total (somme des prix des racks). Le benchmark propose quatre instances définissant chacune un nombre défini de cartes à brancher. De nouveau, chaque particule résout les 4 instances 200 fois avec les mêmes paramètres et nous retenons les résultats moyens. Le tableau 5 montre les résultats de la meilleure particule après 50 itérations PSO. Notre approche est comparée à une méthode de recherche complète basée sur l'élimination des structures isomorphes [8], et aux résultats présentés dans [11] (qui ont à notre connaissance obtenu les meilleurs résultats sur ce problème).[2].

3.3.3 Analyse expérimentale

PSO et les paramètres ACO Afin d'analyser les effets des nombreux paramètres de notre outil, nous avons utilisé PSO séparément sur chaque type de problème.

Le premier résultat important est que les meilleurs jeux de paramètres d'$ACOC_{graph}$ trouvés sont très proches pour les problèmes aléatoires et pour les racks. Ils sont pourtant de nature très différente puisque les instances aléatoires sont des problèmes de satisfaction sur des structures aléatoires non bornées, alors que les racks sont des problèmes d'optimisation structurés sur un nombre borné de composants. Ces expérimentations pourraient donc suggérer que les paramètres trouves sont peu dépendants du problème. Ceci constitue un élément important puisque le calcul de paramètres optimaux est un long processus (environ 20 jours de calcul pour un jeu de problèmes).

[1] http ://www.csplib.org, problème 31

[2] Les deux méthodes exhaustives trouvent évidemment toujours la solution optimale

paramètres d'$ACOC_{graph}$								
nbIte	nbAnts	ρ	evaS	pMin	pMax	noBV	objMin	objMax
200	30	4	restricted	0	9	100	2	90
attBV	attMin	attMax	relBV	relMin	relMax	classBV	classMin	classMax
n/a	n/a	n/a	100	12	90	n/a	n/a	n/a

résultats d'$ACOC_{graph}$			
% de solutions trouvées	nbiterations	taille	temps
100	11	13	89

TAB. 4 – $ACOC_{graph}$ sur les problèmes aléatoires, temps in millisecondes

paramètres d'$ACOC_{graph}$								
nbIte	nbAnts	ρ	evaS	pMin	pMax	objBV	objMin	objMax
500	30	2	restricted	0	11	100	0	90
attBV	attMin	attMax	relBV	relMin	relMax	classBV	classMin	classMax
100	5	90	100	13	90	100	8	90

instance		$ACOC_{graph}$						[8]	[11]
	solution optimale	% de sol. trouvees	% de sol. optimales	prix moyen	nbitérations	taille	temps	temps	temps
1	550	100	88	632	79	21	2113	51	340
2	1100	100	37	1521	240	43	6753	36000	3700
3	1200	100	18	1886	301	56	10690	66	45000
4	1150	97	12	1643	249	33	4729	1800	/

TAB. 5 – $ACOC_{graph}$ sur le problème des racks, temps en millisecondes

Nous pouvons également observer que PSO converge vers des valeurs d'initialisation (*RelBV*, *ObjBV*, *ClassBV*, *AttBV*) proches du maximum. Le bénéfice de ce type d'initialisation avait déjà été souligné pour les algorithmes ACO existants.

Enfin, la nécessité d'une évaporation restreinte est également confirmée par ces expérimentations. Ce paramètre provient des propriétés originales de la configuration : tous les composants générés ne sont pas nécessairement utilisés dans une instance donnée. L'évaporation restreinte permet de "préserver" les phéromones propres à un composant jusqu'à ce qu'il soit sélectionné à nouveau.

Résultats d'$ACOC_{graph}$ Les résultats sur les problèmes aléatoires montrent que l'approche peut résoudre efficacement des problèmes de satisfaction. Ces instances aléatoires ne sont pas faciles à résoudre. En effet l'instance la plus difficile requiert la construction d'une structure contenant (au moins) environ 45 composants et n'offre que peu de solutions. Il faut en moyenne 47 itérations pour trouver une solution sur cette instance.

Les résultats sur les racks sont plus contrastés. D'un côté l'approche ACO trouve des solutions correctes avec des temps de calcul acceptables. D'un autre côté, les solutions trouvées sont en moyenne relativement éloignées de l'optimum. La solution optimale n'est pas non plus trouvée suffisamment fréquemment pour une application réelle. Nous pouvons cependant noter que la meilleure solution est trouvée après un petit nombre d'itérations. L'incapacité actuelle d'améliorer cette solution par les itérations restantes semble donc indiquer un bon potentiel d'amélioration du comportement de l'algorithme.

Ces expérimentations sont encore préliminaires, mais peuvent néanmoins servir de "preuve de concept" car elles représentent les premières tentatives d'utilisation de méthodes stochastiques en configuration. De nombreuses perspectives sont envisagées afin d'améliorer l'efficacité de l'outil. Tout d'abord, comme dans toutes les approches ACO existantes, nous pensons que l'ajout d'heuristiques permettra d'améliorer nettement les résultats. De plus, certains paramètres ont été fixés dans ces expérimentations et des variantes efficaces connues de l'algorithme ACO doivent être essayées. Nous voulons également observer l'évolution des structures après que la meilleure solution ait été trouvée afin d'implanter, si cela se révèle pertinent, des périodes d'intensification. Le contrôle de la taille des solutions est un problème propre à la configuration. Nous pensons qu'il constitue un élément primordial pour l'application d'ACO à ce formalisme. Enfin, nous prévoyons de combiner ACO et recherche locale (combinaison appliquée avec succès aux CSPs dans [16]).

4 Conclusion

Nous avons présenté un cadre théorique et les premiers résultats expérimentaux d'utilisation de l'optimisation par colonie de fourmis pour la recherche de modèles finis en configuration. A notre connaissance, l'utilisation de méthodes stochastiques n'avait pas encore été explorée dans ce domaine. De plus, les solutions apportées aux problèmes posés par la logique du premier-ordre dans les ACO peuvent être ré-utilisées en dehors de la configuration. Etant donné la difficulté des problèmes non bornés de configuration, les résultats sont encourageants. Parmi les nombreuses perspectives, nous prévoyons l'ajout d'heuristiques de choix de variables et de valeurs, l'implémentation de variantes de l'algorithme ACO, ou encore la combinaison d'ACO avec une recherche locale.

Références

[1] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic csps application to configuration. *Artif. Intell.*, 135(1-2) :199–234, 2002.

[2] Ronald J. Brachman and James G. Schmolze. An overview of the kl-one knowledge representation system. *Cognitive Science*, 9(2) :171–216, 1985.

[3] Marco Dorigo and Christian Blum. Ant colony optimization theory : A survey. *Theor. Comput. Sci.*, 344(2-3) :243–278, 2005.

[4] Marco Dorigo, Gianni Di Caro, and Luca Maria Gambardella. Ant algorithms for discrete optimization. *Artificial Life*, 5(2) :137–172, 1999.

[5] Marco Dorigo and Luca Maria Gambardella. Ant colony system : a cooperative learning approach to the traveling salesman problem. *IEEE Trans. Evolutionary Computation*, 1(1) :53–66, 1997.

[6] R.C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Proceedings of the Sixth International Symposium on Micromachine and Human Science*, pages 39–43, 1995.

[7] G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner. Configuring large-scale systems with generative constraint satisfaction. *IEEE Intelligent Systems - Special issue on Configuration*, 13(7), 1998.

[8] Laurent Henocque, Mathias Kleiner, and Nicolas Prcovic. Advances in polytime isomorph elimination for configuration. In *proceedings of Principles and Practice of Constraint Programming - CP 2005,*, pages 301–313, Sitges Barcelona, Spain, 2005. Springer.

[9] Ulrich Junker. Configuration. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 13. Elsevier, 2006.

[10] Ulrich Junker and Daniel Mailharro. The logic of ilog (j)configurator : Combining constraint programming with a description logic. In *proceedings of Workshop on Configuration, IJCAI'03*, 2003.

[11] Z. Kiziltan and B. Hnich. Symmetry breaking in a rack configuration problem. In *Proc. of the IJCAI'01 Workshop on Modelling and Solving Problems with Constraints, Seattle*, 2001.

[12] Deborah L. McGuinness and Jon R. Wright. Conceptual modelling for configuration : A description logic-based approach. *AI EDAM*, 12(4) :333–344, 1998.

[13] Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction problems. In *AAAI*, pages 25–32, 1990.

[14] D. Sabin and E.C. Freuder. Configuration as composite constraint satisfaction. *Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161, 1996.

[15] Timo Soininen, Ilkka Niemelä, Juha Tiihonen, and Reijo Sulonen. Representing configuration knowledge with weight constraint rules. In *Answer Set Programming*, 2001.

[16] Christine Solnon. Ants can solve constraint satisfaction problems. *IEEE Trans. Evolutionary Computation*, 6(4) :347–357, 2002.

[17] Markus Stumptner. An overview of knowledge-based configuration. *AI Communications*, 10(2) :111–125, June 1997.

[18] Markus Stumptner and Alois Haselböck. A generative constraint formalism for configuration problems. In Pietro Torasso, editor, *AI*IA*, volume 728 of *Lecture Notes in Computer Science*, pages 302–313. Springer, 1993.

[19] Thomas Stützle and Holger H. Hoos. Max-min ant system. *Future Generation Comp. Syst.*, 16(8) :889–914, 2000.