Résolution de problèmes combinatoires et optimisation par colonies de fourmis

Christine Solnon

Ce document rassemble différents éléments introduits dans le cours de Master recherche. On définit tout d'abord dans la section 1 ce que l'on entend par « problème combinatoire », et on donne quelques exemples de ces problèmes dans la section 2. La section 3 fait ensuite un panorama des principales approches pour résoudre en pratique ces problèmes. Enfin, la section 4 introduit plus particulièrement les approches inspirées du comportement collectif des colonies de fourmis.

1 Quelques caractéristiques des problèmes combinatoires

On qualifie généralement de « combinatoires » les problèmes dont la résolution se heurte à une explosion du nombre de combinaisons à explorer. C'est le cas par exemple lorsque l'on cherche à concevoir un emploi du temps : s'il y a peu de cours à planifier, le nombre de combinaisons à explorer est faible et le problème sera très rapidement résolu ; cependant, l'ajout de quelques cours seulement peut augmenter considérablement le nombre de combinaisons à explorer de sorte que le temps de résolution devient excessivement long.

Cette notion de problème combinatoire est formellement caractérisée par la théorie de la complexité qui propose une classification des problèmes en fonction de la complexité de leur résolution, et nous introduisons tout d'abord les classes de problèmes qui nous intéressent plus particulièrement. Nous introduisons ensuite les notions de transition de phase et de paysage de recherche qui permettent de caractériser la difficulté « en pratique » des différentes instances d'un même problème combinatoire.

1.1 Complexité théorique d'un problème

On entend ici par « complexité d'un problème » une estimation du nombre d'instructions à exécuter pour résoudre les instances de ce problème, cette estimation étant un ordre de grandeur par rapport à la taille de l'instance. Il s'agit là d'une estimation dans le pire des cas dans le sens où la complexité d'un problème est définie en considérant son instance la plus difficile. Les travaux théoriques dans ce domaine ont permis d'identifier différentes classes de problèmes en fonction de la complexité de leur résolution [Pap94]. Il existe en fait un très grand nombre de classes différentes[1], et on se limitera ici à une présentation succincte nous permettant de caractériser formellement la notion de problème combinatoire.

Problèmes de décision. Les classes de complexité ont été introduites pour les problèmes de décision, c'est-à-dire les problèmes posant une question dont la réponse est « oui » ou « non ». Pour ces problèmes, on définit notamment les deux classes $\mathcal{P}$ et $\mathcal{NP}$:

- La classe $\mathcal{P}$ contient l'ensemble des problèmes polynomiaux, i.e., pouvant être résolus par un algorithme de complexité polynomiale. Cette classe caractérise l'ensemble des problèmes que l'on peut résoudre « efficacement ».
- la classe $\mathcal{NP}$ contient l'ensemble des problèmes polynomiaux non déterministes, i.e., pouvant être résolus par un algorithme de complexité polynomiale pour une machine non déterministe (que l'on peut voir comme une machine capable d'exécuter en parallèle un nombre fini d'alternatives). Intuitivement, cela signifie que la résolution des problèmes de $\mathcal{NP}$ peut nécessiter l'examen d'un grand nombre (éventuellement exponentiel) de cas, mais que l'examen de chaque cas doit pouvoir être fait en temps polynomial.

[1] Le « zoo des complexité », que l'on peut consulter sur `http ://www.complexityzoo.com/`, recense pas moins de 442 classes de complexités différentes.

Les relations d'inclusion entre les classes $\mathcal{P}$ et $\mathcal{NP}$ sont à l'origine d'une très célèbre conjecture que l'on peut résumer par « $\mathcal{P} \neq \mathcal{NP}$ ». En effet, si la classe $\mathcal{P}$ est manifestement inclue dans la classe $\mathcal{NP}$, la relation inverse n'a jamais été ni montrée ni infirmée.

Cependant, certains problèmes de $\mathcal{NP}$ apparaissent plus difficiles à résoudre dans le sens où l'on ne trouve pas d'algorithme polynomial pour les résoudre avec une machine déterministe. Les problèmes les plus difficiles de $\mathcal{NP}$ définissent la classe des *problèmes $\mathcal{NP}$-complets* : un problème de $\mathcal{NP}$ est $\mathcal{NP}$-complet s'il est au moins aussi difficile à résoudre que n'importe quel autre problème de $\mathcal{NP}$, i.e., si n'importe quel autre problème de $\mathcal{NP}$ peut être transformé en ce problème par une procédure polynomiale.

Cette équivalence par transformation polynomiale entre problèmes $\mathcal{NP}$-complets implique une propriété fort intéressante : si l'on trouvait un jour un algorithme polynomial pour un de ces problèmes (n'importe lequel) on pourrait en déduire des algorithmes polynomiaux pour tous les autres problèmes, et on pourrait alors conclure que $\mathcal{P} = \mathcal{NP}$. La question de savoir si un tel algorithme existe a été posée en 1971 par Stephen Cook... et n'a toujours pas reçu de réponse.

Ainsi, les problèmes $\mathcal{NP}$-complets sont des problèmes combinatoires dans le sens où leur résolution implique l'examen d'un nombre exponentiel de cas. Notons que cette classe des problèmes $\mathcal{NP}$-complets contient un très grand nombre de problèmes (dont quelques-uns sont décrits dans la section 2). Pour autant, tous les problèmes combinatoires n'appartiennent pas à cette classe. En effet, pour qu'un problème soit $\mathcal{NP}$-complet, il faut qu'il soit dans la classe $\mathcal{NP}$, i.e., que l'examen de chaque cas puisse être réalisé efficacement, par une procédure polynomiale. Si on enlève cette contrainte d'appartenance à la classe $\mathcal{NP}$, on obtient la classe plus générale des *problèmes $\mathcal{NP}$-difficiles*, contenant l'ensemble des problèmes qui sont « au moins aussi difficiles » que n'importe quel problème de $\mathcal{NP}$, sans nécessairement appartenir à $\mathcal{NP}$.

Problèmes d'optimisation. Le but d'un problème d'optimisation est de trouver une solution maximisant (resp. minimisant) une fonction objectif donnée. A chaque problème d'optimisation on peut associer un problème de décision dont le but est de déterminer s'il existe une solution pour laquelle la fonction objectif soit supérieure (resp. inférieure) ou égale à une valeur donnée. La complexité d'un problème d'optimisation est liée à celle du problème de décision qui lui est associé. En particulier, si le problème de décision est $\mathcal{NP}$-complet, alors le problème d'optimisation est dit $\mathcal{NP}$-difficile.

1.2 ... et en pratique ?

La théorie de la complexité nous dit que si un problème est $\mathcal{NP}$-difficile, alors il est illusoire de chercher un algorithme polynomial pour ce problème (à moins que $\mathcal{P} = \mathcal{NP}$). Toutefois, ces travaux sont basés sur une évaluation de la complexité dans le pire des cas, car la difficulté d'un problème est définie par rapport à son instance la plus difficile. En pratique, si on sait qu'on ne pourra pas résoudre en temps polynomial *toutes* les instances d'un problème $\mathcal{NP}$-difficile, il apparait que certaines instances sont beaucoup plus faciles que d'autres et peuvent être résolues très rapidement.

Considérons par exemple un problème de conception d'emploi du temps, consistant à affecter des créneaux horaires à des cours en respectant des contraintes d'exclusion —exprimant par exemple le fait que certains cours ne doivent pas avoir lieu en même temps. Il s'agit là d'un problème $\mathcal{NP}$-complet classique se ramenant à un problème de coloriage de graphes. Pour autant, la résolution pratique de ce problème peut s'avérer plus ou moins difficile d'une instance à l'autre, même si l'on considère des instances de même taille, i.e., ayant toutes le même nombre de cours et de créneaux horaires. Typiquement, les instances ayant très peu de contraintes d'exclusion sont faciles à résoudre car elles admettent beaucoup de solutions ; les instances ayant beaucoup de contraintes sont aussi facilement traitables car on arrive vite à montrer qu'elles n'admettent pas de solution ; les instances intermédiaires —ayant trop de contraintes pour que l'on trouve facilement une solution, mais pas assez pour que l'on montre facilement qu'elles n'ont pas de solution— sont généralement beaucoup plus difficiles à résoudre.

De façon plus générale, la difficulté des instances d'un problème de décision apparait souvent liée à un phénomène de « transition de phases » que nous introduisons dans le paragraphe suivant. Nous introduisons ensuite la notion de « paysage de recherche » qui permet de caractériser la difficulté des instances de problèmes d'optimisation.

Notion de transition de phases. De nombreux problèmes de décision $\mathcal{NP}$-complets peuvent être caractérisés par un paramètre de contrôle de telle sorte que l'espace des instances du problème comporte deux grandes régions : la région

sous-contrainte où quasiment toutes les instances admettent un grand nombre de solutions, et la région sur-contrainte où quasiment toutes les instances sont insolubles. Très souvent, la transition entre ces deux régions est brusque dans le sens où une très petite variation du paramètre de contrôle sépare les deux régions. Ce phénomène est appelé « transition de phases » par analogie avec la transition de phases en physique qui désigne un changement des propriétés d'un système (e.g., la fusion) provoquée par la variation d'un paramètre extérieur particulier (e.g., la température) lorsqu'il atteint une valeur seuil.

Une caractéristique fort intéressante de ce phénomène est que les instances les plus difficiles à résoudre se trouvent dans la zone de transition [CKT91, Hog96, MPSW98], là où les instances ne sont ni trivialement solubles ni trivialement insolubles. Il s'agit là d'un pic de difficulté dans le sens où une toute petite variation du paramètre de contrôle permet de passer d'instances vraiment très faciles à résoudre à des instances vraiment très difficiles. Notons également que ce pic de difficulté est indépendant du type d'approche de résolution considérée [Dav95, CFG$^+$96].

De nombreux travaux, à la fois théoriques et expérimentaux, se sont intéressés à la localisation de ce pic par rapport au paramètre de contrôle. En particulier, [GMPW96] introduit la notion de « degré de contrainte » (« constrainedness ») d'une classe de problèmes, noté κ, et montre que lorsque κ est proche de 1, les instances sont critiques et appartiennent à la région autour de la transition de phase. Ces travaux sont particulièrement utiles pour la communauté de chercheurs s'intéressant à la résolution pratique de problèmes combinatoires. Ils permettent en particulier de se concentrer sur les instances vraiment difficiles, qui sont utilisées en pratique pour valider et comparer les approches de résolution. Ces connaissances peuvent également être utilisées comme des heuristiques pour résoudre plus efficacement les problèmes $\mathcal{NP}$-complets [Hog98].

Notons cependant que ces travaux se basent généralement sur l'hypothèse d'une génération aléatoire et uniforme des instances par rapport au paramètre de contrôle, de telle sorte que les contraintes sont relativement uniformément réparties sur l'ensemble des variables du problème. Cette hypothèse n'est pas toujours réaliste et les problèmes réels sont souvent plus structurés, exhibant des concentrations irrégulières de contraintes.

Notion de paysage de recherche. La notion de transition de phases n'a de sens que pour les problèmes de décision binaires, pour lesquels on peut partitionner les instances en deux sous-ensembles en fonction de leur réponse. Pour les problèmes d'optimisation, on peut caractériser la difficulté d'une instance d'un problème d'optimisation par différents indicateurs.

Considérons par exemple une instance d'un problème d'optimisation définie par le couple (S, f) tel que S est l'ensemble des configurations candidates et $f : S \rightarrow \mathbb{R}$ est une fonction associant un coût réel à chaque configuration, l'objectif étant de trouver une configuration $s^* \in S$ telle que $f(s^*)$ soit maximal. Un premier indicateur de la difficulté de cette instance est la densité d'états [HR96] qui donne la fréquence d'un coût c par rapport au nombre total de configurations : a priori, plus la densité d'un coût est faible, et plus il sera difficile de trouver une configuration avec ce coût.

Ce premier indicateur, indépendant de l'approche de résolution considérée, ne tient pas compte de la façon d'explorer les configurations. Or, on constate en pratique que cet aspect est déterminant : s'il y a une seule configuration optimale, mais que l'on dispose d'heuristiques simples pour se diriger vers elle dans l'espace des configurations, alors l'instance sera probablement plus facilement résolue que s'il y a plusieurs configurations optimales, mais qu'elles sont « cachées ».

Ainsi, on caractérise souvent la difficulté d'une instance en fonction de la topologie de son paysage de recherche [Sta95, FRPT99]. Cette topologie est définie par rapport à une relation de voisinage $v \subseteq S \times S$ qui dépend de l'approche considérée pour la résolution, ou plus précisémment de la façon d'explorer l'ensemble des configurations : deux configurations s_i et s_j sont voisines si l'algorithme peut « explorer » s_j en une étape de résolution à partir de s_i. Cette relation de voisinage permet de structurer l'ensemble des configurations en un graphe $G = (S, v)$, que l'on peut représenter comme un « paysage » en définissant l'altitude d'un sommet $s_i \in S$ par la valeur de la fonction objectif $f(s_i)$. La topologie du paysage de recherche d'une instance par rapport à un voisinage donne des indications sur la difficulté de sa résolution par un algorithme explorant les configurations selon ce voisinage. En particulier, un paysage « rugueux », comportant de nombreux optima locaux (des sommets dont tous les voisins ont un coût inférieur), est généralement plus difficile qu'un paysage comportant peu d'optima. Un autre indicateur est la corrélation entre le coût d'un sommet et sa proximité à une solution optimale, l'instance étant généralement d'autant plus facilement résolue que cette corrélation est forte [JF95, MF99].

2 Exemples de problèmes combinatoires

De très nombreux problèmes réels sont $\mathcal{NP}$-complets ou $\mathcal{NP}$-difficiles, e.g., pour n'en citer que quelques uns, les problèmes de conception d'emplois du temps, d'équilibrage de chaines de montage, de routage de véhicules, d'affectation de fréquences, de découpe ou d'emballage. Ainsi, [GJ79] décrit près de trois cents problèmes $\mathcal{NP}$-complets ; le « compendium of $\mathcal{NP}$ optimization problems[2] » recense plus de deux cents problèmes d'optimisation $\mathcal{NP}$-difficiles, et la bibliothèque de problèmes en recherche opérationnelle « OR library[3] » en recense plus de cent.

On présente ici deux classes de problèmes combinatoires qui ont un grand nombre d'applications pratiques, à savoir les problèmes de satisfaction de contraintes et les problèmes d'appariement de graphes.

2.1 Problèmes de satisfaction de contraintes

Les contraintes font partie de notre vie quotidienne, qu'il s'agisse par exemple de faire en emploi du temps, de ranger des pièces de formes diverses dans une boîte rigide, ou encore de planifier le trafic aérien pour que tous les avions puissent décoller et atterrir sans se percuter. La notion de « Problème de Satisfaction de Contraintes » (CSP) désigne l'ensemble de ces problèmes, définis par des contraintes, et consistant à chercher une solution les respectant.

De façon plus formelle, un CSP [Tsa93] est défini par un triplet (X, D, C) tel que
– X est un ensemble fini de variables (représentant les inconnues du problème) ;
– D est une fonction qui associe à chaque variable $x_i \in X$ son domaine $D(x_i)$, c'est-à-dire l'ensemble des valeurs que peut prendre x_i ;
– C est un ensemble fini de contraintes. Chaque contrainte $c_j \in C$ est une relation entre certaines variables de X, restreignant les valeurs que peuvent prendre simultanément ces variables.

De façon générale, résoudre un CSP consiste à affecter des valeurs aux variables de telle sorte que toutes les contraintes soient satisfaites. De façon plus précise :
– On appelle *affectation* un ensemble de couples $\langle x_i, v_i \rangle$ tels que x_i est une variable du CSP et v_i est une valeur appartenant au domaine $D(x_i)$ de cette variable.
– Une affectation est dite *totale* si elle instancie toutes les variables du problème ; elle est dite *partielle* si elle n'en instancie qu'une partie.
– Une affectation A *viole* une contrainte c_j si toutes les variables de c_j sont instanciées dans A, et si la relation définie par c_j n'est pas vérifiée pour les valeurs des variables de c_j définies dans A.
– Une affectation (totale ou partielle) est *consistante* si elle ne viole aucune contrainte, et *inconsistante* si elle viole une ou plusieurs contraintes.
– Une *solution* est une affectation totale consistante, c'est-à-dire une valuation de toutes les variables du problème qui ne viole aucune contrainte.

Notons que suivant les applications, « résoudre un CSP » peut signifier différentes choses : on peut chercher, par exemple, une solution (n'importe laquelle), ou bien toutes les solutions, ou bien une solution qui optimise une fonction objectif donnée, ou encore une affectation totale qui satisfait le plus grand nombre de contraintes (on parle alors de max-CSP) [FW92], ou qui minimise une somme pondérée des contraintes violées (on parle alors de CSP valué ou VCSP) [SFV95]. Ces différents types de CSPs ont été généralisés dans le cadre des CSPs basés sur des semi-anneaux [BMR97].

Tous les CSPs ne sont pas des problèmes combinatoires, et leur complexité théorique dépend de la nature des contraintes et des domaines des variables. Dans certains cas le problème peut être polynomial, par exemple lorsque les contraintes sont des équations ou inéquations linéaires et les domaines des variables sont continus. Dans d'autre cas, le problème peut être indécidable, par exemple lorsque les contraintes sont des fonctions mathématiques quelconques et les domaines des variables sont continus. Bien souvent, le problème est $\mathcal{NP}$-complet (lorsqu'il s'agit d'un problème de satisfaction) ou $\mathcal{NP}$-difficile (lorsqu'il s'agit d'un problème d'optimisation). En particulier, les CSPs sur les domaines finis (dont les domaines des variables sont des ensembles finis de valeurs) sont combinatoires dans le cas général.

Le cadre très général des CSPs permet de modéliser de nombreux problèmes réels (voir par exemple la bibliothèque CSPLib [GW99] qui recense 45 problèmes modélisés sous la forme de CSPs). On décrit dans le paragraphe suivant la famille des CSPs binaires aléatoires, qui sont très souvent utilisés comme jeux d'essai pour évaluer la performance d'algorithmes de résolution.

[2] http ://www.nada.kth.se/∼viggo/problemlist/compendium.html
[3] http ://people.brunel.ac.uk/ mastjjb/jeb/info.html

CSPs binaires aléatoires. Les CSPs binaires contiennent uniquement des contraintes binaires, i.e., chaque contrainte porte sur exactement deux variables. Ces CSPs peuvent être générés aléatoirement, ce qui peut sembler quelque peu futile, mais permet en pratique de tester et comparer facilement des algorithmes de résolution. Une classe de CSPs binaires générés aléatoirement est caractérisée par un quadruplet $< n, m, p_1, p_2 >$ où
– n est le nombre de variables du CSP,
– m est la taille du domaine des variables,
– $p_1 \in [0, 1]$ détermine la connectivité du problème, c'est-à-dire le nombre de contraintes,
– $p_2 \in [0, 1]$ détermine la dureté des contraintes, c'est-à-dire le nombre de paires de valeurs incompatibles pour chaque contrainte.
Etant donnée une classe $< n, m, p_1, p_2 >$, on peut considérer différents modèles de génération, dépendant notamment de si l'on considère p_1 et p_2 comme des probabilités ou des ratios [MPSW98].

Une caractéristique très intéressante de cette classe de CSPs est que la région de la transition de phase a été clairement localisée [CFG+96] : les instances les plus difficiles (et donc celles qui vont plus particulièrement nous intéresser) sont celles pour lesquelles $\frac{n-1}{2} p_1 \log_m(\frac{1}{1-p_2}) = 1$.

2.2 Problèmes d'appariement de graphes

Les graphes sont des structures de données utilisées notamment pour modéliser des objets en termes de composants (appelés sommets), et de relations binaires entre composants (appelées arcs). De façon plus formelle, un graphe est défini par un couple $G = (V, E)$ tel que
– V est un ensemble fini de sommets,
– $E \subseteq V \times V$ est un ensemble d'arcs.
Les problèmes d'appariement de graphes consistent à mettre en correspondance les sommets de deux graphes, l'objectif étant généralement de comparer les objets modélisés par les graphes. On introduit ici les quatre problèmes d'appariement les plus connus :
– l'isomorphisme de graphes, qui permet de vérifier que deux graphes sont structurellement identiques ;
– l'isomorphisme de sous-graphes, qui permet de vérifier qu'un graphe est « inclus » dans un autre ;
– le plus grand sous-graphe commun, qui permet d'identifier la plus grande partie commune à deux graphes ;
– la distance d'édition de graphes, qui permet d'identifier le plus petit nombre d'opérations à effectuer pour transformer un graphe en un autre.

Isomorphisme de graphes. Un graphe $G = (V, E)$ est isomorphe à un graphe $G' = (V', E')$ s'il existe une bijection $\phi : V \to V'$ qui préserve les arcs, i.e., $\forall (v_1, v_2) \in V^2, (v_1, v_2) \in E \Leftrightarrow (\phi(v_1), \phi(v_2)) \in E'$.

La complexité théorique du problème de l'isomorphisme de graphes n'est pas précisément établie : le problème est clairement dans $\mathcal{NP}$ mais on ne sait pas s'il est dans $\mathcal{P}$ ou s'il est $\mathcal{NP}$-complet [For96]. La classe des problèmes *isomorphiques-complets* a donc été définie, et on conjecture que cette classe se situe entre la classe P et la classe des problèmes $\mathcal{NP}$-complets. Cependant, il a été montré que pour certains types de graphes (e.g., les graphes planaires [HW74], les arbres [AHU74], ou les graphes à degré borné [Luk82]), le problème devient polynomial.

Isomorphisme de sous-graphes (partiel). Un graphe $G = (V, E)$ est un sous-graphe isomorphe d'un graphe $G' = (V', E')$ s'il existe une injection $\phi : V \to V'$ qui préserve les arcs de G, i.e., $\forall (v_1, v_2) \in V^2, (v_1, v_2) \in E \Leftrightarrow (\phi(v_1), \phi(v_2)) \in E'$.

Notons que l'isomorphisme de sous-graphes impose non seulement de retrouver tous les arcs de G dans G', mais aussi que tout couple de sommets de G non reliés par un arc soit apparié à un couple de sommets de G' également non reliés par un arc. L'isomorphisme de sous-graphe *partiel* relâche cette deuxième condition, i.e., il s'agit de trouver une injection $\phi : V \to V'$ telle que $\forall (v_1, v_2) \in V^2, (v_1, v_2) \in E \Rightarrow (\phi(v_1), \phi(v_2)) \in E'$.

Le problème de décider si un graphe est isomorphe à un sous-graphe (partiel) d'un autre graphe, est un problème $\mathcal{NP}$-complet [GJ79].

Plus grand sous-graphe (partiel) commun. Le plus grand sous-graphe commun de deux graphes $G = (V, E)$ et $G' = (V', E')$ est le plus « grand » graphe qui soit un sous-graphe isomorphe de G et G', la taille du sous-graphe commun pouvant être définie par le nombre de sommets et/ou le nombre d'arcs. Cette définition est naturellement étendue à la notion de plus grand sous-graphe partiel commun en recherchant le plus grand sous-graphe partiel isomorphe à G et G'.

Le problème de la recherche du plus grand sous-graphe (partiel) commun est un problème $\mathcal{NP}$-difficile.

Distance d'édition de graphes. La distance d'édition entre deux graphes G et G' est le coût minimal pour transformer G en G'. Dans le cas général, on considère des graphes étiquetés, i.e., les sommets et les arcs du graphe sont associés à des étiquettes. Pour cette transformation, on dipose de six opérations élémentaires : l'insertion, la suppression et le réétiquetage de sommets et d'arcs. Chaque opération a un coût. L'ensemble d'opérations le moins coûteux permettant de transformer G en un graphe isomorphe à G' définit la distance d'édition entre G et G'. La distance d'édition est en fait une généralisation de la notion de plus grand sous-graphe commun [Bun97] : si l'on considère des graphes non étiquetés, et si l'on définit le coût des opérations d'insertion et de suppression comme étant égal à 1, alors la distance d'édition donne l'ensemble des sommets et arcs qui n'appartiennent pas au plus grand sous-graphe commun.

Le problème du calcul de la distance d'édition de graphes est un problème $\mathcal{NP}$-difficile.

Formulation de problèmes d'appariements de graphes sous la forme de CSPs. Un même problème peut généralement être modélisé de différentes façons, et les problèmes d'appariement de graphes peuvent être formulés sous la forme de problèmes de satisfaction de contraintes [Rég95]. Ainsi, trouver un isomorphisme entre deux graphes $G = (V, E)$ et $G' = (V', E')$ se ramène à résoudre le CSP (X, D, C) tel que :

– X associe une variable x_u à chaque sommet $u \in V$, *i.e.*, $X = \{x_u / u \in V\}$,

– le domaine de chaque variable x_u est l'ensemble des sommets de G' ayant le même nombre d'arcs entrants et le même nombre d'arcs sortants que u, *i.e.*,

$$D(x_u) = \{u' \in V' \ / \quad |\{(u,v) \in E\}| = |\{(u',v') \in E'\}| \text{ et } |\{(v,u) \in E\}| = |\{(v',u') \in E'\}|\}$$

– C contient une contrainte binaire $C_{edge}(x_u, x_v)$ entre chaque paire de variables (x_u, x_v). Cette contrainte exprime le fait que les sommets de G' affectés à x_u et x_v doivent être connectés par un arc si et seulement si les sommets correspondants u et v sont connectés par un arc, *i.e.*,

$$\begin{array}{ll} \text{si } (u,v) \in E, & C_{edge}(x_u, x_v) = E' \\ \text{sinon} & C_{edge}(x_u, x_v) = \{(u',v') \in V'^2 \mid u' \neq v' \text{ et } (u',v') \notin E'\} \end{array}$$

De façon très similaire, on peut modéliser un problème d'isomorphisme de sous-graphe sous la forme d'un CSP, tandis que les problèmes de recherche de plus grands sous-graphes communs et de calcul de distances d'édition, qui sont des problèmes d'optimisation et non plus de satisfaction, peuvent être modélisés sous la forme de CSPs valués.

3 Résolution pratique de problèmes combinatoires

De très nombreux problèmes réels devant être résolus quotidiennement sont combinatoires et il s'agit en pratique de trouver des solutions à ces problèmes. La théorie de la complexité nous disant que l'on ne pourra pas —a priori— trouver un algorithme à la fois rapide, correct et complet, on traite ces problèmes en faisant délibérément l'impasse sur un ou plusieurs de ces critères : l'objectif est d'avoir un comportement « acceptable » en pratique, i.e., d'être capable de trouver une solution de qualité suffisante en un temps raisonnable pour les instances à résoudre.

3.1 Approches complètes

Pour résoudre un problème combinatoire, on peut explorer l'ensemble des configurations de façon exhaustive et systématique, et tenter de réduire la combinatoire en utilisant des techniques d'« élagage » —visant à éliminer le plus tôt possible des sous-ensembles de configurations en prouvant qu'ils ne contiennent pas de solution— et des heuristiques de choix

—visant à se diriger le plus rapidement possible vers les meilleures configurations. Ces approches sont complètes dans le sens où elles sont toujours capables de trouver la solution optimale ou, le cas échéant, de prouver l'absence de solution ; en revanche, le temps de résolution dépend de l'efficacité des techniques d'élagage et des heuristiques considérées, et est exponentiel dans le pire des cas.

Un point clé de ces approches réside dans la façon de structurer l'ensemble des configurations, l'objectif étant de pouvoir éliminer des sous-ensembles de configurations de façon globale, sans avoir à examiner les configurations qu'ils contiennent une par une.

Structuration en arbre. Pour explorer l'ensemble des configurations, on peut le découper en sous-ensembles de plus en plus petits. On construit pour cela un « arbre de recherche » dont la racine correspond à l'ensemble de toutes les configurations à examiner et dont les nœuds correspondent à des sous-ensembles de configurations de plus en plus petits au fur et à mesure que l'on descend dans l'arbre. Plus précisément, les sous-ensembles de configurations associés aux fils d'un nœud constituent une partition de l'ensemble des configurations associées au nœud. Cet arbre est généralement construit selon une stratégie par « séparation et évaluation » (« branch and bound »), en partant de la racine et en itérant sur les opérations suivantes :

– sélectionner une feuille de l'arbre selon une heuristique de sélection donnée (e.g., choisir la dernière feuille créée pour une exploration « en profondeur d'abord », ou choisir la feuille la plus « prometteuse » pour une exploration « du meilleur d'abord ») ;
– séparer (partitionner) l'ensemble des configurations associé à une feuille sélectionnée en plusieurs sous-ensembles (e.g., partitionner l'ensemble des configurations en autant de sous-ensembles qu'il y a de valeurs possibles pour une variable) ;
– évaluer le sous-ensemble de configurations associé à un noeud, afin de déterminer s'il peut contenir une solution (auquel cas le nœud correspondant devra être développé ultérieurement), ou s'il ne contient assurément aucune solution (auquel cas le nœud correspondant peut être « élagué »).

La fonction d'évaluation est déterminante pour la viabilité de l'approche en pratique : cette fonction doit pouvoir déterminer efficacement (i.e., plus rapidement qu'en énumérant toutes les configurations de l'ensemble) qu'un ensemble de configurations ne peut pas contenir de solution.

Pour les problèmes d'optimisation, la fonction d'évaluation retourne une estimation du coût de la meilleure solution de l'ensemble de sorte que si cette estimation est moins bonne que la meilleure solution trouvée jusqu'ici, alors on peut couper le nœud correspondant. Cette estimation du coût est généralement obtenue en relâchant certaines contraintes du problème de sorte que le problème « affaibli » ne soit plus combinatoire et puisse être évalué en temps polynomial. Par exemple, pour résoudre un problème linéaire en nombres entiers (consistant à trouver les valeurs entières à affecter à un ensemble de variables de décision optimisant une fonction objectif linéaire et satisfaisant un certain nombre de contraintes linéaires) selon une stratégie par « séparation et évaluation », on estime généralement le coût d'un nœud de l'arbre de recherche en relâchant les contraintes d'intégralité sur les variables, le problème sur les réels pouvant être résolu en temps faiblement polynomial [Chv83].

Pour les problèmes de décision, la fonction d'évaluation doit être capable de détecter le fait qu'un ensemble ne contient pas de configuration solution. Cette évaluation est souvent réalisée en « filtrant » l'ensemble des configurations, i.e., en éliminant des parties dont on infère qu'elles ne peuvent pas contenir de solution. Ainsi, une stratégie classique pour résoudre des problèmes de satisfaction de contraintes consiste à exploiter (« propager ») les contraintes du problème pour éliminer du domaine des variables les valeurs violant les contraintes, filtrant ainsi l'ensemble des configurations candidates. Le résultat de cette étape de filtrage vérifie généralement une certaine consistance partielle, e.g., la consistance de noeud, d'arc ou de chemin [Tsa93]. L'intégration de ces techniques de propagation de contraintes filtrant les domaines des variables à l'intérieur d'une exploration arborescente est à la base de nombreux « solveurs » de contraintes qui, intégrés dans des langages de programmation, constituent le paradigme de la « programmation par contraintes » [Fag96, MS98].

Structuration en treillis. Lorsque les configurations à explorer sont des sous-ensembles d'un ensemble donné, on peut structurer l'ensemble des configurations en un treillis et adopter une stratégie d'exploration par niveaux (levelwise search) [MT97]. Un exemple célèbre est l'algorithme « a priori » [AS94] utilisé pour calculer des ensembles fréquents maximaux ; ce principe peut également être utilisé pour générer toutes les solutions d'un problème de satisfaction de contraintes [Fre78].

De façon assez générale, cette stratégie d'exploration permet de trouver parmi un ensemble initial d'objets tous les

plus grands sous-ensembles satisfaisant certaines contraintes : chaque niveau i du treillis contient l'ensemble des sous-ensembles comportant i éléments ; le treillis est construit en partant du niveau 1, et en construisant chaque niveau i à partir du niveau $i-1$. Comme pour les approches par séparation et évaluation, l'efficacité de l'approche en pratique dépend de la capacité à élaguer le treillis, et de la possibilité d'utiliser activement les contraintes du problème pour examiner le moins de configurations possible [BJ05]. En particulier, les contraintes anti-monotones (telles que lorsqu'un ensemble vérifie les contraintes, alors tous ses sous-ensembles les vérifient aussi) permettent d'élaguer de façon drastique le treillis, et de traiter de très gros volumes de données en pratique.

3.2 Approches incomplètes

Les approches complètes permettent de résoudre en pratique de nombreux problèmes combinatoires, comme en témoigne le succès industriel de la programmation par contraintes. Cependant, les techniques d'élagage ne permettent pas toujours de réduire suffisamment la combinatoire, et certaines instances de problèmes ne peuvent être résolues en un temps acceptable par ces approches exhaustives.

Une alternative consiste alors à explorer l'ensemble des configurations de façon opportuniste, en utilisant des heuristiques pour se guider vers les zones de l'espace des configurations qui semblent plus prometteuses. Ces approches sont incomplètes dans le sens où elles n'explorent délibérément qu'une partie de l'espace des configurations. Par conséquent, elles peuvent ne pas trouver la solution optimale dans certains cas, et encore moins prouver l'optimalité des solutions trouvées ; en contrepartie, le temps de résolution est généralement faiblement polynomial.

On distingue essentiellement deux types d'approches heuristiques :
- Les approches par voisinage explorent l'espace des configurations en le structurant en termes de voisinage ; ces approches sont qualifiées de « instance-based » dans [ZBMD04] dans le sens où les nouvelles configurations sont créées à partir de configurations existantes.
- Les approches constructives génèrent de nouvelles configurations de façon incrémentale en ajoutant itérativement des composants de solution aux configurations en cours de construction ; ces approches sont qualifiées de « model-based » dans [ZBMD04] dans le sens où elles utilisent un modèle, généralement basé sur un mécanisme stochastique adaptatif, pour construire les configurations.

Approches par voisinage. L'idée est de structurer l'ensemble des configurations en terme de voisinage, et d'explorer cet ensemble en partant d'une ou plusieurs configurations initiales et en choisissant à chaque itération une ou plusieurs configurations voisines de une ou plusieurs configurations courantes. Les approches par voisinage les plus connues sont les algorithmes génétiques et la recherche locale.

Dans le cas des algorithmes génétiques [Hol75], on génère une population initiale de configurations que l'on fait évoluer en créant à chaque génération de nouvelles configurations à partir de la population de configurations courante. Pour ces algorithmes, les opérateurs de voisinage sont généralement le croisement —qui permet de générer deux nouvelles configurations en combinant les composants de deux configurations données— et la mutation —qui permet de générer une nouvelle configuration en modifiant aléatoirement quelques composants d'une configuration donnée. Il existe différentes variantes dépendant notamment de la politique de sélection des configurations à croiser et de la politique de remplacement.

Dans le cas de la recherche locale, on génère une configuration initiale, et l'on explore ensuite l'espace des configurations de proche en proche en sélectionnant à chaque itération une configuration voisine de la configuration courante, l'opérateur de voisinage consistant généralement en une modification « élémentaire » de la configuration courante. Il existe différentes variantes dépendant de la politique de sélection de la prochaine configuration parmi l'ensemble des configurations voisines. On peut par exemple sélectionner à chaque itération le meilleur voisin [SLM92], i.e., celui qui améliore le plus la fonction objectif, ou bien le premier voisin trouvé qui améliore la fonction objectif. De telles stratégies « gloutonnes » risquent fort d'être rapidement piégées dans des optima locaux, i.e., sur des configurations dont tous les voisins sont moins bons. Pour s'échapper de ces optima locaux, on peut considérer différentes méta-heuristiques, e.g., pour n'en citer que quelques unes : le recuit simulé [AK89] —qui autorise de sélectionner des voisins de moins bonne qualité selon une probabilité qui décroit avec le temps— la recherche taboue [GL93] —qui empêche de boucler sur un petit nombre de configurations autour des optima locaux en mémorisant les derniers mouvements effectués dans une liste taboue et en interdisant les mouvements inverses à ces derniers mouvements— ou encore la recherche à voisinage variable [MH97] —qui change d'opérateur de voisinage lorsque la configuration courante est un optimum local par rapport au voisinage courant. Notons également que ce processus de recherche locale peut être répété plusieurs fois à partir de

configurations initiales différentes, comme par exemple dans l'approche « iterated local search » [LMS02] où plusieurs recherches locales sont effectuées en séquence, la configuation initiale de chaque recherche locale étant une perturbation de la configuration finale de la recherche locale précédente.

Approches constructives. L'idée est de construire une ou plusieurs configurations de façon incrémentale, i.e., en partant d'une configuration « vide », et en ajoutant des composants de configuration jusqu'à obtenir une configuration complète. Les composants de configuration à ajouter à la configuration en cours de construction sont choisis en fonction d'une heuristique qui estime localement la qualité de ce composant. Il existe différentes stratégies pour choisir les composants à ajouter à chaque itération en fonction de l'heuristique, les plus connues étant les stratégies gloutonnes et gloutonnes aléatoires, la méta-heuristique d'optimisation par colonies de fourmis et les algorithmes par estimation de distribution.

Les algorithmes gloutons (greedy) construisent une configuration en choisissant à chaque itération un composant de solution pour lequel l'heuristique est maximale. Ces algorithmes permettent de construire des configurations très rapidement, les choix effectués n'étant jamais remis en cause. La qualité de la configuration construite dépend de l'heuristique.

Les algorithmes gloutons aléatoires (greedy randomized) construisent plusieurs configurations et adoptent également une stratégie gloutonne pour le choix des composants à ajouter aux configurations en cours de construction, mais ils introduisent un peu d'aléatoire afin de diversifier les configurations construites. On peut par exemple choisir aléatoirement le prochain composant parmi les k meilleurs ou bien parmi ceux qui sont à moins de α pour cent du meilleur composant [FR89]. Une autre possibilité consiste à choisir le prochain composant en fonction de probabilités dépendant de la qualité des différents composants [JS01].

Les algorithmes à base de fourmis —dont le principe général a donné naissance à la méta-heuristique d'optimisation par colonies de fourmis ACO [DD99, DCG99, DS04]— construisent également des configurations de façon incrémentale, mais choisissent le prochain composant en fonction d'une règle de transition probabiliste qui dépend généralement de deux facteurs : un facteur heuristique —similaire à celui utilisé par les algorithmes gloutons aléatoires— et un facteur phéromonal —fonction des « traces de phéromone » déposées autour du composant. Ces traces de phéromone représentent l'expérience passée de la colonie de fourmis concernant le choix de ce composant et sont régulièrement mises-à-jour en fonction des dernières configurations calculées. Cette famille d'algorithmes est plus détaillée dans la section 4 de ce document.

Les algorithmes par estimation de distribution EDA [LL01] font évoluer une population de configurations P et un modèle probabiliste de génération de configurations M en itèrant sur les trois étapes suivantes : (1) construction d'un ensemble de configurations S à partir du modèle probabiliste courant M, (2) mise-à-jour de la population P en remplacant les moins bonnes configurations de P par les meilleures configurations de S, (3) mise-à-jour du modèle probabiliste M en fonction de P. Différents types de modèles probabilistes, plus ou moins simples, peuvent être utilisés pour construire les configurations, e.g., PBIL [Bal94] —qui se base sur la probabilité d'apparition de chaque composant sans tenir compte d'éventuelles relations de dépendances entre les composants— ou BOA [PGCP99] —qui utilise un réseau bayésien où les relations de dépendance entre composants sont représentées par les arcs d'un graphe auxquels est associée une distribution de probabilités conditionnelles.

Il existe un parallèle assez fort entre ACO et EDA [ZBMD04] : ces deux approches utilisent un modèle probabiliste pour générer les configurations, ce modèle évoluant en fonction des configurations construites dans un processus itératif « d'apprentissage ». Ces algorithmes peuvent être qualifiés d'évolutionnistes dans le sens où le modèle probabiliste générant les configurations évolue au cours du temps.

Intensification versus diversification de la recherche. Pour ces différentes approches heuristiques, explorant l'espace des configurations de façon opportuniste, un point critique dans la mise au point de l'algorithme consiste à trouver un compromis entre deux tendances duales :
– il s'agit d'une part d'intensifier l'effort de recherche vers les zones les plus « prometteuses » de l'espace des configurations, i.e., aux alentours des meilleures configurations trouvées ;
– il s'agit par ailleurs de diversifier l'effort de recherche de façon à être capable de découvrir de nouvelles zones contenant (potentiellement) de meilleures solutions.
La façon d'intensifier/diversifier l'effort de recherche dépend de l'approche considérée et se fait en modifiant certains paramétres de l'algorithme. Pour les approches par voisinage, l'intensification de la recherche se fait en favorisant l'exploration des « meilleurs » voisins. Par exemple, pour les algorithmes génétiques, on augmente la probabilité de croisement des

meilleures configurations de la population et/ou on diminue la probabilité de mutation ; pour la recherche locale taboue, on diminue la longueur de la liste taboue ; pour le recuit simulé, on diminue la température pour diminuer la probabilité de choisir des « mauvais » voisins. Pour les approches par construction, l'intensification de la recherche se fait en augmentant la probabilité de choisir les « meilleurs » composants de solution. Par exemple, pour les algorithmes gloutons aléatoires, on augmente la probabilité de choisir les meilleurs composants de solution en diminuant le nombre k ou le ratio α ; pour la méta-heuristique ACO, on augmente la probabilité de choisir les composants de solution ayant appartenu aux meilleurs configurations trouvées en augmentant l'influence de la phéromone. Une étude de l'influence des différents paramètres d'un algorithme ACO sur l'intensification/diversification est développée en section 4.4 de ce document.

En général, plus on intensifie la recherche d'un algorithme en l'incitant à explorer les configurations proches des meilleures configurations trouvées, et plus il « converge » rapidement, trouvant de meilleures solutions plus tôt. Cependant, si l'on intensifie trop la recherche, l'algorithme risque fort de « stagner » autour d'optima locaux, concentrant son effort de recherche sur une petite zone autour d'une assez bonne configuration, sans plus être capable de découvrir de nouvelles configurations. Notons ici que le bon équilibre entre intensification et diversification dépend clairement du temps de calcul dont on dispose pour résoudre le problème : plus ce temps est petit et plus on a intéret à favoriser l'intensification pour converger rapidement, quitte à converger vers des configurations de moins bonne qualité.

Le bon équilibre entre intensification et diversification dépend également de l'instance à résoudre, ou plus particulièrement de la topologie de son paysage de recherche. En particulier, plus la corrélation entre la qualité d'une configuration et sa distance à la solution optimale est forte et plus on a intéret à intensifier la recherche. Différentes approches ont proposé d'adapter dynamiquement le paramétrage au cours de la résolution, en fonction de l'instance à résoudre. Par exemple, l'approche taboue réactive de [BP01] ajuste automatiquement la longueur de la liste taboue, tandis que l'approche locale de [NTG04] adapte automatiquement le nombre de voisins considérés à chaque mouvement.

3.3 ... et les autres approches

Cette présentation des approches pour la résolution pratique de problèmes combinatoires n'est certainement pas exhaustive, cette classe de problèmes ayant stimulé l'imagination d'un très grand nombre de chercheurs. En particulier, la « classification » proposée —distinguant approches complètes et incomplètes, structuration en arbre et en treillis, exploration par voisinage et par construction— n'est clairement pas exclusive, et de nombreuses approches hybrides ont été proposées, combinant les principes de plusieurs approches pour donner naissance à de nouveaux algorithmes. On peut par exemple combiner approches complètes et recherche locale [JL02, PV04, EGN05], algorithmes génétiques et recherche locale [Mos89], approches constructives et recherche locale [PR02].

Une alternative aux approches complètes et incomplètes pour la résolution de problèmes d'optimisation combinatoire, consiste à résoudre le problème en s'autorisant délibérément une marge d'erreur. En particulier, les algorithmes de « ρ-approximation » [Shm95] sont des algorithmes de complexité polynomiale qui calculent une configuration dont la qualité par rapport à la qualité de la configuration optimale est bornée par ρ. Les problèmes d'optimisation $\mathcal{NP}$-difficiles ne sont pas tous équivalents en termes d »'approximabilité » : certains comme le problème du « bin-packing » peuvent être approximés avec un facteur ρ quelconque, la complexité en temps de l'algorithme d'approximation augmentant lorsque le facteur d'erreur diminue ; d'autres comme le problème de la clique maximum ne peuvent pas être approximés avec un facteur constant, ni même polynomial (à moins que $\mathcal{P} = \mathcal{NP}$).

4 Optimisation par colonies de fourmis

4.1 Des fourmis naturelles à la méta-heuristique d'optimisation par colonies de fourmis

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 sa progression, une fourmi laisse derrière elle une trace de phéromone qui augmente la probabilité que d'autres fourmis passant à proximité choisissent le même chemin [DAGP90, DS04].

Ce mécanisme de résolution collective de problèmes est à l'origine des algorithmes à base de fourmis artificielles. Ces

Procédure $\mathcal{MAX} - \mathcal{MIN}$ ***Ant System***
Initialiser les traces de phéromone à τ_{max}
Répéter
Chaque fourmi construit une solution en exploitant les traces de phéromone
Optionnellement : améliorer les solutions construites par recherche locale
Evaporer les traces de phéromone en les multipliant par un facteur de persistance ρ
Récompenser les meilleures solutions en ajoutant de la phéromone sur leurs composants phéromonaux
si une trace de phéromone est inférieure à τ_{min} **alors** la mettre à τ_{min}
si une trace de phéromone est supérieure à τ_{max} **alors** la mettre à τ_{max}
Jusqu'à ce qu'un nombre maximal de cycles soit atteint **ou** une solution de qualité acceptable ait été trouvée
retourner la meilleure solution trouvée

FIG. 1 – Cadre algorithmique général du $\mathcal{MAX} - \mathcal{MIN}$ *Ant System*

algorithmes ont été initialement proposés dans [Dor92, DMC96], 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, appelé graphe de construction, puis d'utiliser des fourmis artificielles pour rechercher de bons chemins dans ce graphe. Le comportement des fourmis artificielles est inspiré des fourmis réelles : elles déposent des traces de phéromone sur les composants du graphe de construction et elles choisissent leurs chemins relativement aux traces de phéromone précédemment déposées ; ces traces sont évaporées au cours du temps.

Intuitivement, cette communication indirecte via l'environnement —connue sous le nom de stigmergie— fournit une information sur la qualité des chemins 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, essentiellement 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 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. Enfin, la probabilité pour une fourmi artificielle de choisir un arc ne dépend généralement pas uniquement des traces de phéromone, mais aussi d'heuristiques dépendantes du problème permettant d'évaluer localement la qualité du chemin.

Ces caractéristiques du comportement des fourmis artificielles définissent la « méta-heuristique d'optimisation par une colonie de fourmis » ou « Ant Colony Optimization (ACO) metaheuristic » [DD99, DCG99, DS04]. Cette méta-heuristique a permis de résoudre différents problèmes d'optimisation combinatoire, comme par exemple le problème du voyageur de commerce [DG97], le problème d'affectation quadratique [GTD99] ou le problème de routage de véhicules [BHS99].

4.2 Le $\mathcal{MAX} - \mathcal{MIN}$ *Ant System*

Le premier algorithme à base de fourmis, appelé *Ant System*, a été proposé par Marco Dorigo en 1992 [Dor92, DMC96], et ses performances ont été initialement illustrées sur le problème du voyageur de commerce. Malgré des premiers résultats encourageants, montrant que les performances de *Ant System* sont comparables à celles d'autres approches « généralistes » comme les algorithmes génétiques, l'algorithme *Ant System* n'est pas compétitif avec des approches spécialisées, développées spécialement pour le problème du voyageur de commerce. Ainsi, différentes améliorations ont été apportées à l'algorithme initial, donnant naissance à différentes variantes de *Ant System*, comme par exemple *ACS* (*Ant Colony System*) [DG97] et $\mathcal{MMAS}$ ($\mathcal{MAX} - \mathcal{MIN}$ *Ant System*) [SH00] qui obtiennent en pratique des résultats vraiment compétitifs, parfois meilleurs que les approches les plus performantes.

De nombreux travaux sur l'optimisation par colonies de fourmis se sont inspirés du schéma algorithmique du $\mathcal{MMAS}$ décrit dans la figure 1. Conformément à la méta-heuristique ACO, à chaque cycle de l'algorithme chaque fourmi construit une solution. Ces solutions peuvent éventuellement être améliorées en appliquant une procédure de recherche locale. Les traces de phéromone sont ensuite mises-à-jour : chaque trace est « évaporée » en la multipliant par un facteur de

persistance ρ compris entre 0 et 1 ; une certaine quantité de phéromone, généralement proportionnelle à la qualité de la solution, est ensuite ajoutée sur les composants des meilleures solutions (les meilleures solutions construites lors du dernier cycle ou les meilleures solutions construites depuis le début de l'exécution).

La construction d'une solution par une fourmi dépend de l'application considérée, mais suit toujours le même schéma général et se fait de façon incrémentale selon un principe glouton probabiliste : partant d'une solution initialement vide, la fourmi ajoute incrémentalement des composants de solution choisis selon une règle de transition probabiliste. En général, cette règle de transition définit la probabilité de choisir un composant en fonction de deux facteurs : un facteur phéromonal —qui traduit l'expérience passée de la colonie concernant le choix de ce composant— et un facteur heuristique —qui évalue localement la qualité du composant. L'importance respective de ces deux facteurs dans la probabilité de transition est généralement modulée par deux paramètres α et β. Un exemple d'application de la méta-heuristique ACO à des problèmes de recherche de sous-ensembles optimaux est développée en 4.3.

Une caractéristique essentielle du $\mathcal{MMAS}$ est qu'il impose des bornes maximales et minimales τ_{min} et τ_{max} aux traces de phéromone. L'objectif est d'éviter une stagnation prématurée de la recherche, i.e., une situation où les traces de phéromone sont tellement marquées que les fourmis construisent les mêmes solutions à chaque itération et ne sont plus capables de trouver de nouvelles solutions. En effet, en imposant des bornes sur les quantités maximales et minimales de phéromone, on garantit 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 (après i cycles, cette différence est bornée par un ratio de ρ^i où ρ est le facteur de persistance de la phéromone et a une valeur généralement très proche de 1), de sorte que l'exploration est accentuée.

4.3 Exemple d'application d'ACO à des problèmes de sélection de sous-ensembles

Beaucoup de problèmes résolus par la méta-heuristique ACO sont des problèmes d'ordonnancement qui se modélisent assez naturellement sous la forme de recherche de meilleurs chemins hamiltoniens dans un graphe —où les fourmis doivent visiter chaque sommet du graphe— e.g., les problèmes de voyageur de commerce [DMC96], d'affectation quadratique [SH00], de routage de véhicules [BHS99] ou d'ordonnancement de voitures [GPS03]. Pour résoudre de tels problèmes de recherche de meilleurs chemins hamiltoniens avec la méta-heuristique ACO, on associe généralement une trace de phéromone $\tau(i, j)$ à chaque arc (i, j) du graphe. Cette trace de phéromone représente l'expérience de la colonie concernant l'intéret de visiter le sommet j juste après le sommet i, et elle est utilisée pour guider les fourmis dans leur phase de construction de chemins.

Cependant, de nombreux problèmes combinatoires se ramènent à un problème de sélection plutôt que d'ordonnancement : étant donné un ensemble d'objets S, le but de ces problèmes est de trouver un sous-ensemble de S qui satisfait certaines propriétés et/ou optimise une fonction objectif donnée. Nous appellerons ces problèmes des problèmes de sélection de sous-ensembles, abrégé par SS-problèmes. Quelques exemples classiques de SS-problèmes sont, par exemple, le problème de la clique maximum, le problème du sac-à-dos multidimensionnel, le problème de la satisfiabilité de formules booléennes, et le problème de satisfaction de contraintes.

Pour ces problèmes, les solutions sont des sous-ensembles d'objets, et l'ordre dans lequel les objets sont sélectionnés n'est pas significatif. Par conséquent, cela n'a pas beaucoup de sens de les modéliser comme des problèmes de recherche de chemins et de déposer des traces de phéromone sur des couples de sommets visités consécutivement. Deux structures phéromonales différentes peuvent être considérées pour résoudre un SS-problème à l'aide de la méta-heuristique ACO ;

– On peut associer une trace de phéromone $\tau(i)$ à chaque objet $i \in S$, de telle sorte que $\tau(i)$ représente l'expérience passée de la colonie concernant l'intéret de sélectionner l'objet i. Cette structure phéromonale a été expérimentée sur différents SS-problèmes, comme par exemple des problèmes de sac-à-dos multidimensionnels [LM99], de recouvrement d'ensembles [HRTB00], de satisfaction de contraintes [TB05], de cliques maximums [SF06], d'appariement de graphes [SSSG06] et de recherche de k-arbres de poids minimum [Blu02].
– On peut associer une trace de phéromone $\tau(i, j)$ à chaque paire d'objets différents $(i, j) \in S \times S$, de telle sorte que $\tau(i, j)$ représente l'expérience passée de la colonie concernant l'intéret de sélectionner les objets i et j dans un même sous-ensemble. Cette structure phéromonale a été expérimentée sur différents SS-problèmes : des problèmes de recherche de cliques maximums [FS03], de sac-à-dos multidimensionnels [ASG04], de satisfaction de contraintes [Sol02], de recherche de k-arbres de poids minimum [Blu02], et d'appariement de graphes [SSG05].

On décrit ici ces différents algorithmes dans un cadre unifié, et on introduit donc un algorithme ACO générique pour les

SS-problèmes. Cet algorithme est paramétré d'une part par les caractéristiques du SS-problème à résoudre, et d'autre part par la structure phéromonale considérée —déterminant si les fourmis déposent la phéromone sur des objets ou sur des paires d'objets.

4.3.1 Problèmes de sélection de sous-ensembles

Les problèmes de sélection de sous-ensembles (SS-problèmes) ont pour but de trouver un sous-ensemble consistant et optimal d'objets. Plus formellement, un SS-problème est défini par un triplet $(S, S_{consistant}, f)$ tel que
- S est un ensemble d'objets ;
- $S_{consistant} \subseteq \mathcal{P}(S)$ est l'ensemble de tous les sous-ensembles de S qui sont consistants ;
  Afin de pouvoir construire chaque sous-ensemble de $S_{consistant}$ de façon incrémentale (en partant de l'ensemble vide et en ajoutant à chaque itération un objet choisi parmi l'ensemble des objets consistants avec l'ensemble en cours de construction), on impose la contrainte suivante : pour chaque sous-ensemble consistant non vide $S' \in S_{consistant}$, il doit exister au moins un objet $o_i \in S'$ tel que $S' - \{o_i\}$ est aussi consistant.
- $f : S_{consistant} \rightarrow I\!R$ est la fonction objectif qui associe un coût $f(S')$ à chaque sous-ensemble d'objets consistant $S' \in S_{consistant}$.

Le but d'un SS-problème $(S, S_{consistant}, f)$ est de trouver $S^* \in S_{consistant}$ tel que $f(S^*)$ soit maximal.

On décrit ci-dessous deux problèmes combinatoires selon ce formalisme $(S, S_{consistant}, f)$; on trouvera dans [SB06] une description plus complète de problèmes combinatoires selon ce formalisme.

Le problème de la clique maximum a pour but de trouver le plus grand sous-ensemble de sommets connectés deux à deux dans un graphe :
- S contient l'ensemble des sommets du graphe ;
- $S_{consistant}$ contient toutes les cliques du graphe, i.e., tout ensemble $S' \subseteq S$ tel que chaque paire de sommets distincts de S' soit connectée par une arête du graphe ;
- f est la fonction cardinalité.

Les problèmes de satisfaction maximale de contraintes (Max-CSP) ont pour but de trouver une affectation de valeurs à des variables qui satisfait un maximum de contraintes, i.e.,
- S contient chaque étiquette $\langle X_i, v_i \rangle$ associant une variable X_i à une valeur v_i appartenant au domaine de X_i ;
- $S_{consistant}$ contient tous les sous-ensembles de S qui n'ont pas deux étiquettes différentes pour une même variable, i.e.,

$$S_{consistant} = \{S' \subseteq S \mid \forall(\langle X_i, v_i \rangle, \langle X_j, v_j \rangle) \in S' \times S', X_i = X_j \Rightarrow v_i = v_j\}$$

- f évalue le nombre de contraintes satisfaites, i.e., $\forall S' \in S_{consistant}, f(S')$ est le nombre de contraintes telles que chaque variable impliquée dans la contrainte est affectée à une valeur par une étiquette de S' et la contrainte est satisfaite par cette affectation.

4.3.2 Un algorithme ACO générique pour les problèmes de sélection de sous-ensembles

L'algorithme ACO générique, appelé *Ant-SS*, est paramétré par :
- le SS-problème à résoudre, décrit par un triplet $(S, S_{consistant}, f)$;
  une instanciation de cet algorithme pour résoudre un problème de recherche de clique maximum est décrite en 4.3.4.
- une stratégie phéromonale qui détermine l'ensemble des composants phéromonaux sur lesquels les fourmis déposent des traces de phéromone, et la façon d'exploiter et de renforcer ces traces ;
  deux stratégies phéromonales différentes sont décrites en 4.3.3.
- un ensemble de paramètres numériques, dont le rôle et l'initialisation sont discutés en 4.4.

L'algorithme suit le cadre algorithmique général du $\mathcal{MAX} - \mathcal{MIN}$ *Ant System* rappelé dans la figure 1, et nous décrivons dans les deux paragraphes suivants la procédure utilisée par les fourmis pour construire une solution et la procédure de dépôt de phéromone.

Procédure construire-sous-ensemble
Entrée : un SS-problème $(S, S_{consistant}, f)$ et une fonction heuristique associée $\eta_{factor} : S \times \mathcal{P}(S) \rightarrow \mathbb{R}^+$;
une stratégie phéromonale et un facteur phéromonal associé $\tau_{factor} : S \times \mathcal{P}(S) \rightarrow \mathbb{R}^+$;
deux paramètres à valeurs numériques : α et β
Sortie : un sous-ensemble consistant d'objets $S' \in S_{consistant}$

1. Choisir aléatoirement le premier objet $o_i \in S$
2. $S_k \leftarrow \{o_i\}$
3. $Candidats \leftarrow \{o_j \in S \mid S_k \cup \{o_j\} \in S_{consistant}\}$
4. **Tant que** $Candidats \neq \emptyset$ **faire**
5. Choisir un objet $o_i \in Candidats$ avec la probabilité
6. $p_{o_i} = \frac{[\tau_{factor}(o_i, S_k)]^\alpha \cdot [\eta_{factor}(o_i, S_k)]^\beta}{\sum_{o_j \in Candidats} [\tau_{factor}(o_j, S_k)]^\alpha \cdot [\eta_{factor}(o_j, S_k)]^\beta}$
7. $S_k \leftarrow S_k \cup \{o_i\}$
8. Enlever o_i de $Candidats$
9. Enlever de $Candidats$ chaque objet o_j tel que $S_k \cup \{o_j\} \notin S_{consistant}$
10. **Fin tant que**

FIG. 2 – Algorithme de construction d'un sous-ensemble s par une fourmi

Construction d'une solution par une fourmi. La figure 2 décrit la procédure suivie par les fourmis pour construire un sous-ensemble. Le premier objet est choisi aléatoirement ; les objets suivants sont choisis au sein de l'ensemble *Candidats* —contenant l'ensemble des objets « consistants » par rapport à l'ensemble des objets qui ont déjà été sélectionnés par la fourmi— en fonction d'une règle de transition probabiliste. Plus précisément, la probabilité p_{o_i} de sélectioner un objet $o_i \in Candidats$ pour une fourmi k ayant déjà sélectionné un sous-ensemble d'objets S_k dépend de deux facteurs :
– Le *facteur phéromonal* $\tau_{factor}(o_i, S_k)$ évalue l'expérience de la colonie en ce qui concerne l'ajout de l'objet o_i au sous-ensemble S_k sur la base des traces phéromonales qui ont été déposées précédemment sur les composants phéromonaux. La définition de ce facteur dépend de la stratégie phéromonale, comme décrit en 4.3.3.
– Le *facteur heuristique* $\eta_{factor}(o_i, S_k)$ évalue l'intéret de l'objet o_i en fonction d'informations locales à la fourmi, i.e., en fonction de la solution qu'elle a construite jusqu'ici S_k. La définition de ce facteur dépend du problème $(S, S_{consistant}, f)$.
Conformément à la méta-heuristique ACO, α et β sont deux paramètres qui déterminent l'importance relative de ces deux facteurs.

Une fois que chaque fourmi a construit une solution, certaines des solutions construites peuvent être éventuellement améliorées en utilisant une forme de recherche locale dépendante du problème : dans certains cas, la recherche locale est appliquée à toutes les solutions ; dans d'autres elle n'est appliquée qu'à la meilleure solution du cycle.

Dépot de phéromone. On adopte ici une stratégie élitiste, où seulement les meilleures fourmis du cycle, i.e., celles ayant trouvé les meilleures solutions du cycle, déposent de la phéromone. La quantité de phéromone déposée par ces meilleures fourmis dépend de la qualité des solutions qu'elles ont construites : elle est inversement proportionnelle à l'écart de qualité entre la solution construite et la meilleure solution construite depuis le début de l'exécution S_{best}. Plus précisément, si l'ensemble des solutions construites pendant le dernier cycle est $\{S_1, \ldots, S_{nbAnts}\}$ alors la procédure de dépôt de phéromone est la suivante :

pour toute solution $S_i \in \{S_1, \ldots, S_{nbAnts}\}$ **faire**
si $\forall S_k \in \{S_1, \ldots, S_{nbAnts}\}, f(S_i) \geq f(S_k)$ **alors**
pour chaque composant phéromonal c de S_i **faire**
$\tau(c) \leftarrow \tau(c) + \frac{1}{1+f(S_{\text{best}})-f(S_i)}$

L'ensemble des composants phéromonaux associés à la solution S_i dépend de la stratégie phéromonale considérée et est décrit en 4.3.3.

4.3.3 Instanciations de l'algorithme par rapport à deux stratégies phéromonales

L'algorithme générique *Ant-SS* est paramétré par une stratégie phéromonal et nous décrivons maintenant les deux instanciations de cet algorithme : *Ant-SS(Vertex)* —où la phéromone est déposée sur les objets— et *Ant-SS(Clique)* —où la phéromone est déposée sur des paires d'objets. Pour chacune de ces deux stratégies phéromonales, nous définissons :
– l'ensemble des composants phéromonaux associés à un SS-problème $(S, S_{consistant}, f)$, i.e., l'ensemble des composants sur lesquels des traces de phéromone peuvent être déposées ;
– l'ensemble des composants phéromonaux associés à une solution $S_k \in S_{consistant}$, i.e., l'ensemble des composants phéromonaux sur lesquels de la phéromone est effectivement déposée lorsque la solution S_k est récompensée ;
– le facteur phéromonal $\tau_{factor}(o_i, S_k)$ associé à un objet o_i et à une solution partielle S_k, facteur phéromonal qui est utilisé dans la règle de transition probabiliste.

La stratégie phéromonale « Vertex ». Dans un algorithme ACO, les fourmis déposent de la phéromone sur les composants des meilleures solutions construites afin d'attirer les autres fourmis vers les zones correspondantes de l'espace de recherche. Pour les problèmes de recherche de sous-ensembles, les solutions construites par les fourmis sont des sous-ensembles d'objets, et l'ordre dans lequel les objets ont été sélectionnés n'est pas significatif. Ainsi, une première stratégie phéromonale pour ce type de problèmes consiste à déposer la phéromone sur les objets sélectionnés.
– L'ensemble des composants phéromonaux contient tous les objets de S. Intuitivement, la quantité de phéromone se trouvant sur un composant phéromonal $o_i \in S$ —$\tau(o_i)$— représente l'expérience passée de la colonie concernant l'intéret de sélectionner l'objet o_i quand on construit une solution.
– L'ensemble des composants phéromonaux associés à une solution S_k est l'ensemble des objets $o_i \in S_k$.
– Le facteur phéromonal dans la règle de transition probabiliste correspond à la quantité de phéromone se trouvant sur l'objet candidat, i.e., $\tau_{factor}(o_i, S_k) = \tau(o_i)$.

La stratégie phéromonale « Clique ». La stratégie phéromonale précédente suppose implicitement que la « désirabilité » d'un objet est relativement indépendante des objets déjà sélectionnés, ce qui n'est pas toujours le cas. Considérons par exemple un problème de satisfaction de contraintes contenant deux variables x et y pouvant prendre pour valeur 0 ou 1, de telle sorte que l'ensemble initial d'objets S contient les étiquettes $\langle x, 0\rangle$, $\langle x, 1\rangle$, $\langle y, 0\rangle$ et $\langle y, 1\rangle$. Supposons maintenant que ce problème contient la contrainte $x \neq y$. Dans ce cas, la « désirabilité » de sélectionner une étiquette pour y, à savoir $\langle y, 0\rangle$ ou $\langle y, 1\rangle$, dépend de l'étiquette déjà sélectionnée pour x (et inversement). Ici, les traces de phéromone pourraient être utilisées pour « apprendre » que $(\langle x, 0\rangle, \langle y, 1\rangle)$ et $(\langle x, 1\rangle, \langle y, 0\rangle)$ sont des paires d'étiquettes qui vont bien ensemble tandis que $(\langle x, 0\rangle, \langle y, 0\rangle)$ et $(\langle x, 1\rangle, \langle y, 1\rangle)$ sont des paires d'étiquettes qui sont moins intéressantes.

Ainsi, une seconde stratégie phéromonale pour les problèmes de sélection de sous-ensembles consiste à déposer de la phéromone sur des paires d'objets.
– L'ensemble des composants phéromonaux contient toutes les paires d'objets différents $(o_i, o_j) \in S \times S$. Intuitivement, la quantité de phéromone se trouvant sur un composant phéromonal $(o_i, o_j) \in S \times S$ —$\tau(o_i, o_j)$— représente l'expérience passée de la colonie concernant l'intéret de sélectionner les objets o_i et o_j dans une même solution.
– L'ensemble des composants phéromonaux associés à une solution S_k est l'ensemble des paires d'objets différents $(o_i, o_j) \in S_k \times S_k$. Autrement dit, la clique d'arêtes associée à S_k est récompensée.
– Le facteur phéromonal dans la règle de transition probabiliste dépend de la quantité de phéromone se trouvant entre l'objet candidat o_i et l'ensemble des objets déjà sélectionnés dans S_k, i.e.,

$$\tau_{factor}(o_i, S_k) = \sum_{o_j \in S_k} \tau(o_i, o_j)$$

Notons que ce facteur phéromonal peut être calculé de façon incrémentale : une fois que le premier objet o_i a été choisi aléatoirement, pour chaque objet candidat o_j, le facteur phéromonal $\tau_{factor}(o_j, S_k)$ est initialisé à $\tau(o_i, o_j)$; ensuite, à chaque fois qu'un nouvel objet o_l est ajouté à S_k, pour chaque objet candidat o_j, le facteur phéromonal $\tau_{factor}(o_j, S_k)$ est incrémenté de $\tau(o_l, o_j)$.

Comparaison des deux stratégies. Les deux stratégies phéromonales ont été introduites indépendamment d'un graphe de construction et du parcours d'une fourmi dans ce graphe. Nous illustrons et comparons maintenant les deux stratégies selon ce point de vue. Dans les deux cas, le graphe de construction est un graphe complet non-orienté associant un sommet à chaque objet, et le résultat d'un parcours d'une fourmi dans ce graphe est l'ensemble des sommets par lesquels elle est

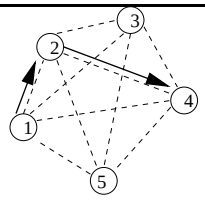
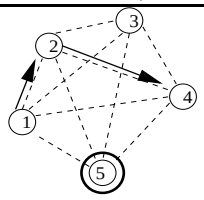
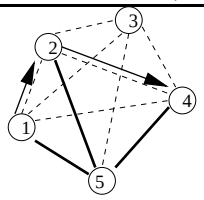
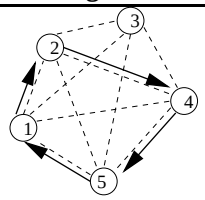
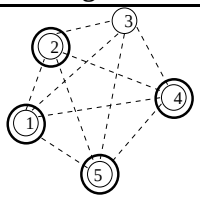
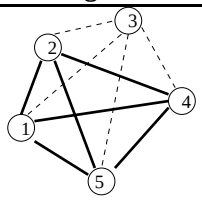
	Chemin de la fourmi	*Ant-SS(Vertex)*	*Ant-SS(Clique)*
Facteur phéromonal d'un sommet par rapport à un début de parcours	 $S_k = \{1, 2, 4\}$ **Fig. 3a.**	 $\tau_{factor}(5, S_k) = \tau(5)$ **Fig. 3b.**	 $\tau_{factor}(5, S_k) = \tau(1,5) + \tau(2,5) + \tau(4,5)$ **Fig. 3c.**
Composants phéromonaux récompensés par rapport à un parcours complet	 **Fig. 3d.**	 **Fig. 3e.**	 **Fig. 3f.**

FIG. 3 – Stratégies phéromonales

passée. La figure 3a représente un début de parcours pour une fourmi ayant successivement sélectionné les sommets 1, 2 et 4. Dans la figure 3b et 3c, on représente en gras les composants phéromonaux respectivement considérés par la fourmi dans *Ant-SS(Vertex)* et *Ant-SS(Clique)* pour définir sa probabilité de choisir l'objet 5 : alors que le facteur phéromonal dans *Ant-SS(Vertex)* ne dépend que de la quantité de phéromone se trouvant sur le sommet candidat, le fateur phéromonal dans *Ant-SS(Clique)* dépend de toutes les traces de phéromone déposées entre l'objet candidat et l'ensemble courant d'objets sélectionnés.

La figure 3d représente ensuite le parcours complet d'une fourmi, l'ensemble d'objets sélectionnés correspondant étant $\{1, 2, 4, 5\}$. Dans les figures 3e et 3f, on représente en gras les composants phéromonaux sur lesquels la phéromone sera respectivement déposée par *Ant-SS(Vertex)* et *Ant-SS(Clique)*. Notons que pour *Ant-SS(Clique)* toutes les arêtes appartenant à la clique des sommets visités sont récompensées (et non seulement celles qui ont été « traversées » lors de la construction de la solution).

En termes de complexité algorithmique, *Ant-SS(Vertex)* et *Ant-SS(Clique)* exécutent à peu près le même nombre d'opérations pour construire une solution (algorithme de la figure 2). La seule différence réside dans le calcul des facteurs phéromonaux et, comme nous l'avons souligné précédemment, dans *Ant-SS(Clique)* les facteurs phéromonaux peuvent être calculés de façon incrémentale : à chaque fois qu'un objet est sélectionné le facteur phéromonal de chaque objet candidat peut être mis-à-jour par une simple addition, de sorte que les complexités en temps des deux instanciations sont les mêmes (dans la mesure où les objets appartenant à l'ensemble *Candidats* doivent être énumérés à chaque itération afin de déterminer s'ils sont encore candidats à l'itération suivante).

Cependant, pour mettre à jour les traces de phéromone *Ant-SS(Vertex)* et *Ant-SS(Clique)* ne font pas le même nombre d'opérations : dans *Ant-SS(Vertex)*, l'étape d'évaporation est en $\mathcal{O}(\mid S \mid)$ et la récompense d'un sous-ensemble S_k est en $\mathcal{O}(\mid S_k \mid)$, tandis que dans *Ant-SS(Clique)*, l'étape d'évaporation est en $\mathcal{O}(\mid S \mid^2)$ et la récompense d'un sous-ensemble S_k est en $\mathcal{O}(\mid S_k \mid^2)$.

4.3.4 Instanciation de *Ant-SS* pour résoudre le problème de la clique maximum

Pour résoudre un problème particulier de sélection de sous-ensembles avec *Ant-SS*, il s'agit essentiellement de définir le facteur heuristique $\eta_{factor}(o_i, S_k)$, qui évalue localement l'intéret de l'objet o_i par rapport à la solution courante S_k et qui est utilisé dans la règle de transition probabiliste. Optionnellement, une procédure de recherche locale peut être définie afin d'améliorer les solutions construites.

On illustre ici la généricité de *Ant-SS* à travers le problème de la clique maximum. Différents exemples d'instanciations d'*Ant-SS* pour la résolution d'autres SS-problèmes peuvent être trouvés dans [SB06].

Facteur heuristique. Curieusement, pour le problème de la clique maximum il s'est avéré qu'il est généralement préférable de ne pas utiliser de fateur heuristique, i.e., $\eta_{factor}(o_i, S_k) = 1$ de telle sorte que les fourmis choisissent les objets sur la base du facteur phéromonal uniquement.

En effet, les heuristiques classiques pour ce problème consistent à favoriser les sommets ayant un degré important dans le graphe « résiduel », i.e., le sous-graphe induit par l'ensemble des sommets candidats, la motivation étant que plus le sommet sélectionné a un degré important, plus il y aura de candidats pour entrer dans la clique à l'itération suivante. Quand on n'utilise pas la phéromone (dans un algorithme de construction incrémentale gloutonne classique par exemple), cette heuristique permet clairement d'obtenir des cliques plus grandes qu'un choix aléatoire [BP01]. Cependant, quand on utilise un mécanisme de rétroaction phéromonal, on obtient de plus grandes cliques en fin d'exécution lorsque l'on n'utilise pas cette heuristique. De fait, des expérimentations sur différentes instances de ce problème nous ont montré que l'utilisation de l'heuristique a pour conséquence d'augmenter très fortement le taux de ré-échantillonnage, i.e., le pourcentage de solutions qui sont re-calculées. Cela montre que l'heuristique attire trop fortement la recherche autour d'un ensemble de cliques localement optimales, de sorte que la recherche n'est plus assez diversifiée pour trouver des cliques plus grandes.

Recherche locale. Différentes procédures de recherche locale peuvent être considérées. Nous avons plus particulièrement utilisé la procédure basée sur un échange $(2, 1)$ utilisée dans *GRASP* [APR99] : étant donnée une solution (clique) S_k, chaque mouvement consiste à rechercher trois objets (sommets) o_i, o_j et o_k tels que (i) $o_i \in S_k$; (ii) $o_j \notin S_k$ et $o_l \notin S_k$; et (iii) o_j et o_l sont adjacents à chaque sommet de $S_k - \{o_i\}$. On remplace alors o_i par o_j et o_l, augmentant ainsi de un la taille de la clique. Ces mouvements sont itérativement répétés jusqu'à obtenir une clique localement optimale, i.e., ne pouvant plus être améliorée par un tel échange $(2, 1)$.

Cette procédure de recherche locale est appliquée sur la meilleure clique du cycle seulement. En effet, nous avons constaté que si on l'applique à toutes les cliques construites pendant le cycle, la qualité de la solution finale n'est pas vraiment améliorée tandis que le temps d'exécution est significativement augmenté.

4.3.5 Résultats expérimentaux

Nous présentons ici quelques résultats expérimentaux permettant de comparer les deux stratégies phéromonales de *Ant-SS* sur le problème de la clique maximum et les problèmes de satisfaction de contraintes.

Instances considérées. Pour le problème de la clique maximum, on décrit les résultats obtenus sur six graphes de la famille $Cn.9$ du challenge DIMACS[4]. Ces graphes ont respectivement 125, 250, 500, 1000 et 2000 sommets, et leurs plus grandes cliques connues ont respectivement 34, 44, 57, 68 et 78 sommets. Ce premier ensemble d'instances nous permet notamment d'illustrer le passage à l'échelle de *Ant-SS*.

Pour les problèmes de satisfaction de contraintes, on décrit les résultats obtenus sur quatre ensembles d'instances générées aléatoirement selon le modèle A décrit dans [MPSW98]. On ne considère ici que des instances satisfiables, dont on sait qu'elles admettent une solution. Chacun des quatre ensembles contient vingt instances différentes, chacune ayant 100 variables, des domaines de taille 8, et une connectivité égale à 0.14 (i.e., la probabilité p_1 de poser une contrainte entre deux variables est égale à 0.14). Les quatre ensembles d'instances ont été générés avec différentes valeurs pour la probabilité p_2 —déterminant la dureté de chaque contrainte en terme du nombre de couples de valeurs enlevés pour chaque contrainte. Le but est d'illustrer le comportement de *Ant-SS* dans la région de la transition de phase. Ainsi, les quatre ensembles d'instances ont été générés en fixant p_2 respectivement à 0.2, 0.23, 0.26 et 0.29 de telle sorte que le taux de difficulté κ soit respectivement égal à 0.74, 0.87, 1.00 et 1.14, la transition de phase étant située autour du point pour lequel $\kappa = 1$.

Conditions expérimentales. Pour toutes les instances, nous avons fixé le nombre de fourmis *nbAnts* à 30, α à 1, ρ à 0.99 et τ_{min} à 0.01. Pour les instances du problème de la clique maximum, nous avons fixé τ_{max} à 6 et β à 0 ; pour les instances du problème de satisfaction de contraintes, nous avons fixé τ_{max} à 4 et β à 10.

[4] http ://dimacs.rutgers.edu/

Problème de la clique maximum

		Sans recherche locale		Avec recherche locale	
Graphe	$\omega(G)$	*Ant-SS(Vertex)*	*Ant-SS(Clique)*	*Ant-SS(Vertex)*	*Ant-SS(Clique)*
C125.9	34	34.0 (34)	34.0 (34)	34.0 (34)	34.0 (34)
C250.9	44	43.9 (44)	44.0 (44)	44.0 (44)	44.0 (44)
C500.9	≥ 57	55.2 (56)	55.6 (57)	55.3 (56)	55.9 (57)
C1000.9	≥ 68	65.3 (67)	66.0 (67)	65.7 (67)	66.2 (68)
C2000.9	≥ 78	73.4 (76)	74.1 (76)	74.5 (77)	74.3 (78)

Problèmes de satisfaction de contraintes

		Sans recherche locale		Avec recherche locale	
p_t	κ	*Ant-SS(Vertex)*	*Ant-SS(Clique)*	*Ant-SS(Vertex)*	*Ant-SS(Clique)*
0.20	0.74	100%	100%	100%	100%
0.23	0.87	45%	93%	91%	100%
0.26	1.00	89%	97%	99%	100%
0.29	1.14	100%	100%	100%	100%

TAB. 1 – Résultats expérimentaux : comparaison de la qualité des solutions. Chaque ligne donne la qualité des solutions obtenues avec *Ant-SS(Vertex)* et *Ant-SS(Clique)*, sans recherche locale puis avec recherche locale. Pour le problème de la clique maximum, on donne la taille moyenne des cliques trouvées suivie entre parenthèses de la taille de la plus grande clique trouvée pour les 50 exécutions ; pour les problèmes de satisfaction de contraintes, on donne le pourcentage d'exécutions qui ont trouvé une solution.

Pour les instances du problème de la clique maximum, chaque algorithme a été exécuté 50 fois sur chacune des six instances ; pour les instances du problème de satisfaction de contraintes, chaque algorithme a été exécuté 5 fois sur chacune des vingt instances des quatre ensembles d'instances (ce qui fait un total de cent exécutions par ensemble).

Toutes les exécutions ont été faites sur un Pentium 4 cadencé à 2GHz.

Comparaison de la qualité des solutions. Les deux premières colonnes du tableau 1 donnent la qualité des solutions trouvées par *Ant-SS(Vertex)* et *Ant-SS(Clique)* lorsque l'on n'utilise pas de recherche locale pour améliorer les solutions construites par les fourmis. Sur les instances considérées ici *Ant-SS(Clique)* est généralement meilleur que *Ant-SS(Vertex)* : les deux variantes sont capables de trouver les solutions optimales aux instances « faciles » (i.e., le plus petit graphe C125.9 pour la clique maximum, et les instances suffisamment éloignées de la région de la transition de phases pour les problèmes de satisfaction de contraintes). Cependant, pour les instances plus difficiles (i.e., quand on augmente la taille du graphe pour la clique maximum, ou quand on se rapproche de la région de la transition de phases pour les problèmes de satisfaction de contraintes), *Ant-SS(Clique)* obtient toujours de meilleurs résultats que *Ant-SS(Vertex)*.

Les deux dernières colonnes du tableau 1 donnent la qualité des solutions trouvées lorsque l'on applique une procédure de recherche locale pour améliorer les solutions construites par les fourmis. On constate ici que l'intégration de la recherche locale dans *Ant-SS* améliore effectivement la qualité des solutions, pour les deux stratégies phéromonales.

Comparaison du temps CPU. Le tableau 2 montre que le nombre de cycles —et par conséquent le temps CPU— nécessaires pour trouver la meilleure solution dépend de la taille du problème et de sa difficulté. Par exemple, Ant-SS(Vertex) exécute respectivement 60, 359, 722, 1219 et 1770 cycles en moyenne pour résoudre les instances du problème de la clique maximum qui ont respectivement 125, 250, 500, 1000 et 2000 sommets respectivement. De la même façon, pour les problèmes de satisfaction de contraintes le nombre de cycles augmente lorsque l'on se rapproche de la région de la transition de phases.

Ce tableau montre aussi que Ant-SS(Vertex) converge en moins de cycles que Ant-SS(Clique) pour les problèmes de cliques maximums tandis qu'il fait quasiment toujours plus de cycles pour les problèmes de satisfaction de contraintes. Cependant, comme un cycle de Ant-SS(Clique) est beaucoup plus long à exécuter qu'un cycle de Ant-SS(Vertex), Ant-SS(Vertex) converge toujours plus rapidement que Ant-SS(Clique).

Notons également que le nombre de cycles est toujours diminué quand la recherche locale est utilisée pour améliorer les solutions construites par les fourmis. Cependant, comme cette étape de recherche locale prend du temps, le temps CPU

Problème de la clique maximum

	Sans recherche locale				Avec recherche locale			
	Ant-SS(Vertex)		*Ant-SS(Clique)*		*Ant-SS(Vertex)*		*Ant-SS(Clique)*	
Graphe	Cycles	Temps	Cycles	Temps	Cycles	Temps	Cycles	Temps
C125.9	60	0.1	126	0.2	14	0.0	23	0.0
C250.9	359	0.8	473	1.7	172	0.5	239	1.0
C500.9	722	3.8	923	8.9	477	4.6	671	8.6
C1000.9	1219	13.2	2359	55.0	832	23.4	1242	49.8
C2000.9	1770	41.3	3268	214.4	1427	112.4	2067	238.7

Problèmes de satisfaction de contraintes

	Sans recherche locale				Avec recherche locale			
	Ant-SS(Vertex)		*Ant-SS(Clique)*		*Ant-SS(Vertex)*		*Ant-SS(Clique)*	
p_t	Cycles	Temps	Cycles	Temps	Cycles	Temps	Cycles	Temps
0.20	90	1.2	63	4.8	2	0.0	2	0.0
0.23	1084	19.5	849	78.2	472	30.8	230	31.6
0.26	1019	18.1	737	66.1	365	26.9	260	38.7
0.29	449	7.8	516	45.2	27	1.8	38	5.4

TAB. 2 – Résultats expérimentaux : comparaison du temps CPU. Chaque ligne donne le nombre moyen de cycles et le temps CPU (en secondes) utilisés pour trouver la meilleure solution.

global n'est pas toujours diminué.

Temps CPU vs qualité de solution. Les tableaux 1 et 2 montrent que pour choisir entre *Ant-SS(Vertex)* et *Ant-SS(Clique)*, il s'agit de prendre en compte le temps CPU disponible pour le processus de résolution. En effet, *Ant-SS(Clique)* trouve généralement de meilleures solutions que *Ant-SS(Vertex)* à la fin de son processus de résolution, mais il est aussi plus long à converger. Par conséquent, si l'on doit trouver une solution rapidement, il vaut mieux utiliser *Ant-SS(Vertex)*, tandis que si l'on dispose de plus de temps, il vaut mieux utiliser *Ant-SS(Clique)*. Cela est illustré dans la figure 4 sur le problème de la recherche d'une clique maximum pour le graphe `C500.9`. Cette figure trace l'évolution de la taille de la plus grande clique trouvée (en moyenne sur 50 exécutions) en fonction du temps CPU. On constate sur cette figure que pour des temps CPU inférieurs à 8 secondes, *Ant-SS(Vertex)* trouve de plus grandes cliques que *Ant-SS(Clique)*, tandis que pour des temps CPU plus importants *Ant-SS(Clique)* trouve de plus grandes cliques que *Ant-SS(Vertex)*.

Cette figure compare aussi *Ant-SS* avec une procédure de recherche locale « multi-start » appelée *multi-start LS*. Cette procédure de recherche locale itère sur les deux étapes suivantes : (1) construction aléatoire d'une clique maximale, et (2) amélioration de cette clique par la procédure de recherche locale décrite en 4.3.4. On remarque que pour des temps CPU inférieurs à 1 seconde, *multi-start LS* trouve de meilleures solutions que *Ant-SS*. En effet, *Ant-SS* prend du temps pour gérer la phéromone (la déposer, l'exploiter et l'évaporer) tandis que cette phéromone ne commence à influencer les fourmis qu'après quelques centaines de cycles. Par conséquent *Ant-SS(Vertex)* (resp. *Ant-SS(Clique)*) devient meilleur que *multi-start LS* seulement après une seconde (resp. cinq secondes) de temps CPU.

Comparaison avec d'autres approches pour la résolution de CSPs. Le tableau 1 montre que pour les CSPs binaires toutes les exécutions de *Ant-SS(Clique)*, lorsqu'il est combiné avec une procédure de recherche locale, ont réussi à trouver une solution, même pour les instances les plus difficiles. On donne dans [vHS04] plus de résultats expérimentaux et on compare les performances de *Ant-SS(Clique)* avec celles d'un algorithme génétique —l'algorithme *Glass-Box* [Mar97] qui s'est avéré le plus performant dans l'étude comparative de [CEv03]— et celles d'une approche complète —l'algorithme de [Pro93] qui combine un processus de vérification en avant (forward checking) avec des techniques de sauts en arrière dirigés par les conflits (conflict directed backjumping). On montre dans [vHS04] que *Ant-SS(Clique)* est nettement meilleur que *Glass-Box* —qui a beaucoup de mal à trouver des solutions pour les instances proches de la transition de phases. On montre également que si l'approche complète est plus rapide pour les petites instances (moins de 40 variables), ses temps d'exécution progressent de façon exponentielle lorsque l'on augmente le nombre de variables tandis que les temps d'exécution de *Ant-SS(Clique)* progressent de façon polynomiale, de sorte que *Ant-SS(Clique)* devient plus rapide dès lors que le nombre de variables devient supérieur à 40.

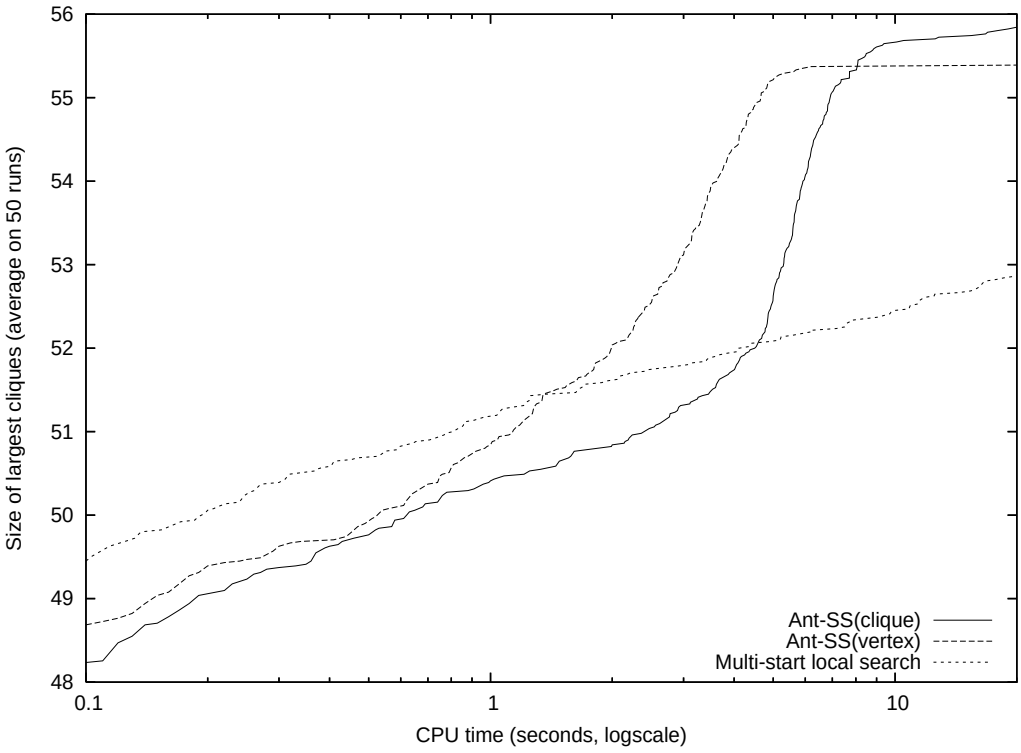


FIG. 4 – Comparaison des stratégies « clique » et « vertex » et d'une procédure de recherche locale « multi-start » : chaque courbe trace l'évolution de la taille de la plus grande clique trouvée (en moyenne sur 50 exécutions) au cours du temps.

Ainsi, *Ant-SS(Clique)* apparait comme une approche robuste pour résoudre les CSPs binaires aléatoires dans le sens où ses taux de réussite sont très proches de 100%, même pour les instances les plus difficiles. Notons cependant que si les temps d'exécution de *Ant-SS(Clique)* progressent plus doucement que ceux des approches complètes lorsque l'on augmente la taille des instances, ils restent tout de même assez élevés et, de façon générale, ils sont plus importants que ceux des approches par recherche locale taboue comme celle, e.g., de [GH97].

Comparaison avec d'autres approches pour la recherche de cliques maximums. Pour les problèmes de recherche de cliques maximums, on compare dans [SF06] les performances de *Ant-SS* avec celles de trois approches parmi les plus performantes : une approche taboue réactive [BP01], une approche gloutonne adaptative [GLC04] et une approche combinant un algorithme génétique avec une procédure de recherche locale [Mar02]. On montre que *Ant-SS(Clique)* obtient des résultats très compétitifs : il est clairement meilleur que l'approche génétique ; il obtient des résultats comparables à ceux de l'approche gloutonne adaptative —meilleurs sur certaines instances et moins bons sur d'autres— ; il est légèrement moins bon que l'approche taboue réactive —qui est la meilleure approche, à notre connaissance, pour ce problème.

4.4 Intensification versus diversification de la recherche avec ACO

4.4.1 Influence des paramètres sur le processus de résolution

Le comportement des fourmis par rapport à la dualité intensification/diversification introduite en 3.2 peut être influencé en modifiant les valeurs des paramètres numériques. En particulier, les paramètres α et ρ ont une influence prépondérante sur le processus de résolution : la diversification peut être accentuée soit en diminuant la valeur du poids du facteur phéromonal α —de telle sorte que les fourmis deviennent moins sensibles aux traces de phéromone et donc à l'expérience passée de la colonie— soit en augmentant la valeur du taux de persistance phéromonale ρ —de telle sorte que la phéromone s'évapore plus doucement.

Quand on augmente les capacités d'exploration des fourmis de cette façon, l'algorithme trouve généralement de meilleures

solutions, mais en contrepartie il met plus de temps pour trouver ces solutions. Ainsi, l'initialisation de ces deux paramètres dépend essentiellement du temps dont on dispose pour la résolution : si le temps est limité, il vaut mieux choisir des valeurs qui favorisent une convergence rapide, comme $\alpha \in \{3, 4\}$ et $\rho < 0.99$; si en revanche le temps n'est pas limité, il vaut mieux choisir des valeurs qui favorisent l'exploration, comme $\alpha \in \{1, 2\}$ et $\rho \geq 0.99$.

Cette dualité a été observée sur de nombreux problèmes résolus avec ACO. Nous l'illustrons dans la figure 5 sur l'instance `C500.9` du problème de recherche de cliques maximums (instance relativement difficile, pour laquelle *Ant-SS* a du mal à trouver la clique maximum qui a 57 sommets). Sur cette figure, on remarque qu'en début de résolution, aussi bien pour *Ant-SS(Clique)* que pour *Ant-SS(Vertex)*, les fourmis trouvent de meilleures solutions avec des valeurs de α et ρ favorisant l'intensification (comme par exemple $\alpha = 2$ et $\rho = 0.98$). En revanche, après un millier de cycles, les fourmis trouvent de meilleures solutions avec des valeurs de α et ρ favorisant l'exploration (comme par exemple $\alpha = 1$ et $\rho = 0.99$). On constate également que quand $\alpha = 0$ et $\rho = 1$, les meilleures cliques construites sont beaucoup plus petites : dans ce cas, la phéromone est complètement ignorée et le processus de résolution se comporte de façon purement aléatoire de telle sorte qu'après quelques centaines de cycles la taille de la meilleure clique n'augmente quasiment plus, et atteint péniblement 48 sommets. L'écart entre cette courbe et les autres permet de quantifier l'apport de la phéromone dans le processus de résolution.

4.4.2 Evaluation de l'effort de diversification/intensification pendant une exécution

Pour évaluer le degré d'intensification de la recherche lors de l'exécution d'un algorithme ACO, on peut mesurer la distribution des quantités de phéromone déposées sur le graphe de construction [DG97] : si lors du choix d'un composant de solution, la majorité des candidats ont un facteur phéromonal très faible et seulement quelques candidats ont un facteur phéromonal important, les fourmis choisiront quasiment toujours les mêmes composants, qui seront de nouveau renforcés dans un processus auto-catalytique. Notons cependant que l'introduction des bornes τ_{min} et τ_{max} dans le $\mathcal{MMAS}$ permet de limiter ce phénomène de stagnation.

Une autre possibilité pour évaluer le degré d'intensification de la recherche consiste à mesurer la diversité des solutions calculées pendant une exécution. De nombreuses mesures de diversité ont été introduites pour les approches évolutionnistes car le maintien de la diversité dans la population est un point clé pour empêcher une convergence prématurée de la résolution. Les mesures de diversité les plus courantes sont notamment le nombre de valeurs de « fitness » différentes, le nombre d'individus structurellement différents, et la distance moyenne entre les individus de la population [BGK02].

Pour mesurer l'effort de diversification dans les algorithmes ACO, on peut notamment utiliser deux indicateurs : le taux de ré-échantillonnage et le taux de similarité.

Taux de ré-échantillonnage. Cette mesure a été utilisée, par exemple, dans [vHB02, vHS04], afin de fournir des informations sur la capacité d'un algorithme heuristique à « échantillonner » l'espace de recherche : si on définit *nbDiff* comme étant le nombre de solutions différentes générées par un algorithme pendant une de ses exécutions, et *nbTot* comme étant le nombre total de solutions générées, alors le taux de ré-échantillonnage est défini par $(\textit{nbTot} - \textit{nbDiff})/\textit{nbTot}$. Un taux proche de to 0 correspond à une recherche « efficace », i.e., les solutions générées sont généralement différentes les unes des autres, tandis qu'un taux proche de 1 indique une stagnation du processus de recherche autour d'un petit nombre de solutions qui sont reconstruites régulièrement.

Le tableau 3 donne un aperçu des performances de *Ant-SS* en termes de ré-échantillonnage. On constate que *Ant-SS(Vertex)* est moins « efficace » que *Ant-SS(Clique)* pour explorer l'espace de recherche car il reconstruit souvent des cliques qu'il avait déjà construites auparavant. Par exemple, quand on fixe α à 1 et ρ à 0.99, 7% des cliques construites par *Ant-SS(Vertex)* pendant les mille premiers cycles avaient déjà été construites avant. Ce taux de ré-échantillonnage augmente ensuite très rapidement pour atteindre 48% au bout de 2500 cycles, i.e., près de la moitié des cliques construites avaient déjà été construites auparavant. En comparaison, *Ant-SS(Clique)* ne recalcule quasiment jamais deux fois une même clique de sorte qu'il explore effectivement deux fois plus d'états dans l'espace de recherche.

Taux de similarité. Le taux de ré-échantillonnage nous permet de quantifier la taille de l'espace exploré, et nous montre par exemple que *Ant-SS(Clique)* a une plus grande capacité d'exploration que *Ant-SS(Vertex)* pour l'instance `C500.9` du problème de la clique maximum. Cependant, le taux de ré-échantillonnage ne donne aucune information sur la distribution des états explorés dans l'espace de recherche. De fait, une recherche purement aléatoire ne recalcule quasiment jamais

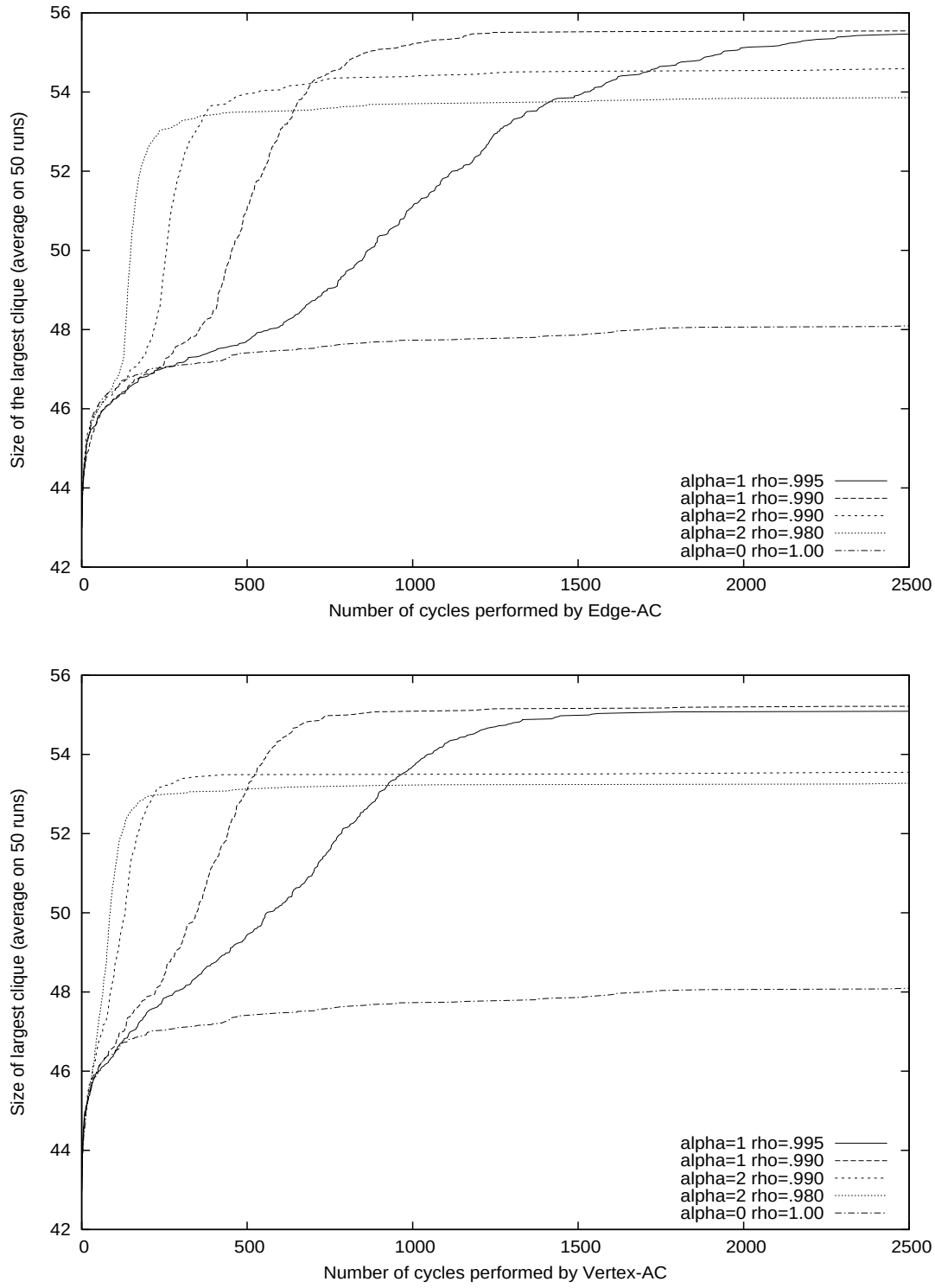


FIG. 5 – Influence de la phéromone sur le processus de résolution de *Ant-SS(Clique)* (courbes du haut) et *Ant-SS(Vertex)* (courbes du bas) pour le problème de la recherche d'une clique maximum dans le graphe `C500.9` : chaque courbe trace l'évolution de la taille de la meilleure clique construite (moyenne sur 50 exécutions) quand le nombre de cycles augmente, pour des valeurs données de α et ρ. Les autres paramètres ont été fixés à $nbAnts = 30$, $\tau_{min} = 0.01$, et $\tau_{max} = 6$.

Taux de ré-échantillonnage de *Ant-SS(Clique)*

Nombre de cycles :	500	1000	1500	2000	2500
$\alpha = 1, \rho = 0.995$	0.00	0.00	0.00	0.00	0.00
$\alpha = 1, \rho = 0.99$	0.00	0.00	0.00	0.00	0.00
$\alpha = 2, \rho = 0.99$	0.00	0.04	0.06	0.07	0.07
$\alpha = 2, \rho = 0.98$	0.06	0.10	0.12	0.13	0.13

Taux de ré-échantillonnage de *Ant-SS(Vertex)*

Nombre de cycles :	500	1000	1500	2000	2500
$\alpha = 1, \rho = 0.995$	0.00	0.00	0.02	0.13	0.25
$\alpha = 1, \rho = 0.99$	0.00	0.07	0.26	0.39	0.48
$\alpha = 2, \rho = 0.99$	0.38	0.68	0.78	0.84	0.87
$\alpha = 2, \rho = 0.98$	0.58	0.78	0.85	0.88	0.91

TAB. 3 – Evolution du taux de ré-échantillonnage de *Ant-SS(Clique)* et *Ant-SS(Vertex)* pour le problème de recherche d'une clique maximum dans le graphe `C500.9`. Chaque ligne donne successivement les valeurs de α et ρ, et les taux de ré-échantillonnage après 500, 1000, 1500, 2000, et 2500 cycles (en moyenne sur 50 exécutions). Les autres paramètres ont été fixés à $nbAnts = 30$, $\tau_{min} = 0.01$, $\tau_{max} = 6$.

deux fois la même solution mais, comme elle n'intensifie pas non plus la recherche autour des zones les plus prometteuses, elle n'est généralement pas capable de trouver de bonnes solutions. Afin de fournir une vue complémentaire sur les capacités des algorithmes ACO en termes de diversification et intensification, on peut également calculer le taux de similarité qui quantifie le degré de similarité des solutions construites, i.e., d'intensification de la recherche. Ce taux de similarité correspond à la « pair-wise population diversity measure » introduite pour les algorithmes génétiques, e.g., dans [SBC01, MJ01].

De façon générale, le taux de similarité d'une suite de solutions S correspond au ratio entre la taille moyenne de l'intersection de chaque paire de solutions de S par rapport à la taille moyenne des solutions de S. Un taux de similarité égal à 1 correspond à une suite de solutions toutes identiques tandis qu'un taux de similarité à 0 correspond à une suite dont toutes les paires de solutions ont une intersection vide.

La figure 6 trace l'évolution du taux de similarité de l'ensemble des cliques construites tous les 50 cycles, et montre ainsi l'évolution de la répartition de l'ensemble des cliques calculées au sein de l'espace des cliques, pour le problème de la recherche d'une clique maximum dans le graphe `C500.9`. Considérons par exemple la courbe traçant l'évolution du taux de similarité pour *Ant-SS(Clique)* lorsque $\alpha = 1$ et $\rho = 0.99$. La similarité passe de moins de 10% au début du processus de résolution à 45% après un millier de cycles. Cela montre que les fourmis se focalisent progressivement sur une partie de l'espace de recherche, de telle sorte que deux cliques construites après 1000 cycles partagent près de la moitié de leurs sommets en moyenne. Si l'on ajoute à cela le fait que le taux de ré-échantillonnage est nul, on peut conclure que dans ce cas *Ant-SS(Clique)* a trouvé un bon compromis entre la diversification —étant donné qu'il ne recalcule jamais deux fois une même solution— et l'intensification —étant donné que la similarité des solutions calculées augmente.

Les courbes de la figure 6 nous montrent également que lorsque α augmente ou ρ diminue, le taux de similarité augmente à la fois plus tôt et plus fortement. Cependant, à la fois pour *Ant-SS(Clique)* et *Ant-SS(Vertex)*, le taux de similarité des différentes exécutions converge en fin de résolution vers une même valeur, quelles que soient les valeurs de α et ρ : après deux mille cycles, les cliques calculées par *Ant-SS(Clique)* partagent environ 45% de leurs sommets tandis que celles calculées par *Ant-SS(Vertex)* partagent environ 90% de leurs sommets.

La différence de diversification entre *Ant-SS(Clique)* et *Ant-SS(Vertex)* peut s'expliquer par les choix faits au sujet de leurs composants phéromonaux. Considérons par exemple le problème de la recherche d'une clique maximum dans le graphe `C500.9`. Ce graphe a 500 sommets de sorte que *Ant-SS(Vertex)* peut déposer de la phéromone sur 500 composants phéromonaux tandis que *Ant-SS(Clique)* peut en déposer sur $500 \cdot 499/2 = 124750$ composants. Considérons maintenant les deux cliques qui sont récompensées à la fin des deux premiers cycles d'une exécution de *Ant-SS*. Pour les deux algorithmes *Ant-SS(Vertex)* et *Ant-SS(Clique)*, ces deux cliques contiennent 44 sommets en moyenne (voir la figure 5) et leur taux de similarité est inférieur à 10% (voir la figure 6) de sorte qu'elles partagent en moyenne 4 sommets. Sous cette hypothèse, après les deux premiers cycles de *Ant-SS(Vertex)*, 4 sommets —soit 1% des composants phéromonaux— ont été récompensés deux fois, et 80 sommets —soit 16% des composants phéromonaux— ont été récompensés une

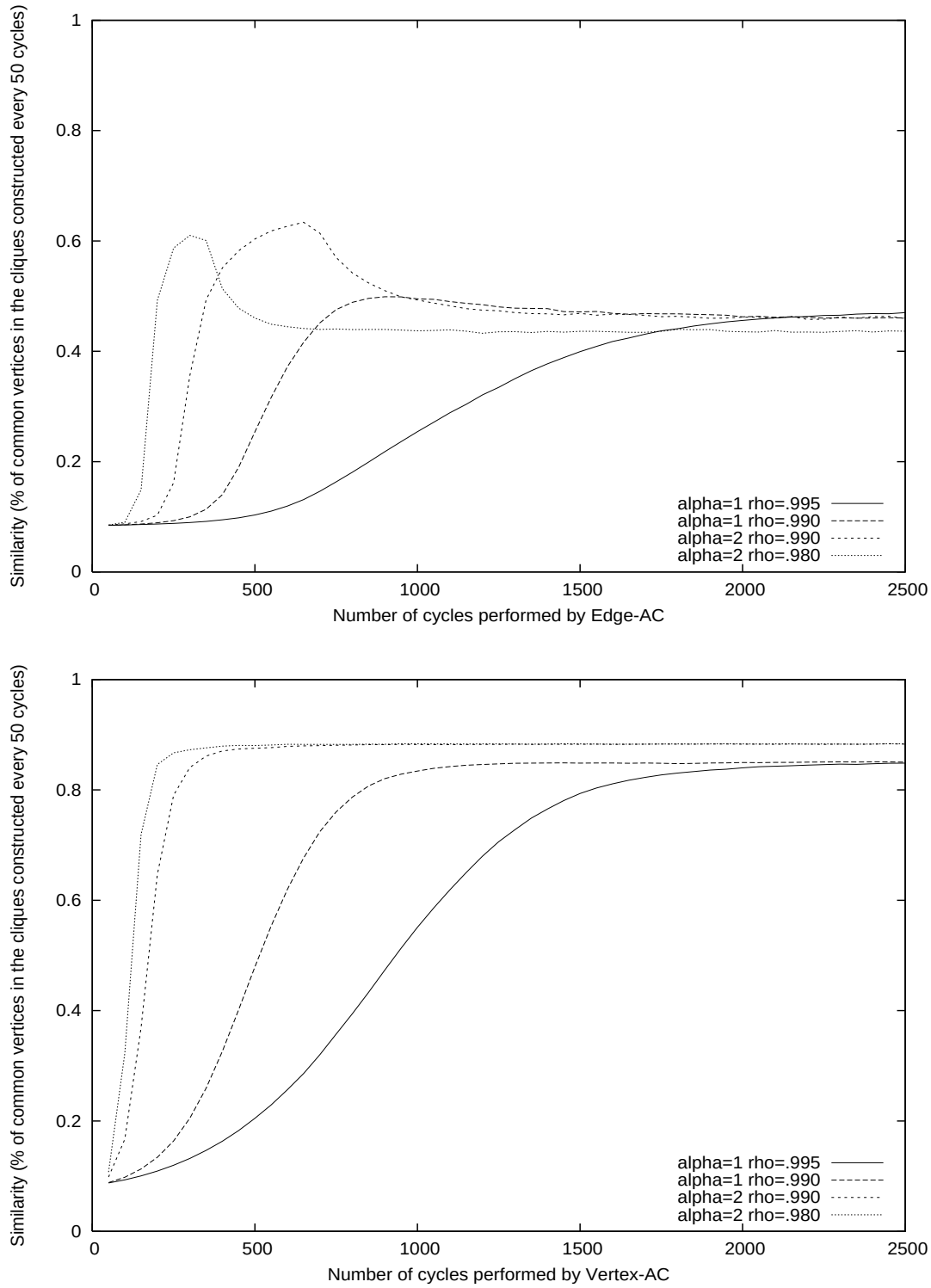


FIG. 6 – Evolution du taux de similarité pour *Ant-SS(Clique)* (courbes du haut) et *Ant-SS(Vertex)* (courbes du bas) pour le problème de la recherche d'une clique maximum dans le graphe `C500.9` : chaque courbe donne le taux de similarité de l'ensemble des cliques construites tous les 50 cycles (en moyenne sur 50 exécutions), pour différentes valeurs de α et ρ. Les autres paramètres ont été fixés à $nbAnts = 30$, $\tau_{min} = 0.01$, $\tau_{max} = 6$.

fois. A titre de comparaison, et sous les mêmes hypothèses, après les deux premiers cycles de *Ant-SS(Clique)*, 6 arêtes —soit moins de 0.005% des composants phéromonaux— ont été récompensées deux fois, et 940 arêtes —soit moins de 0.8% des composants phéromonaux— ont été récompensées une fois. Cela explique pourquoi la recherche est plus diversifiée avec *Ant-SS(Clique)* qu'avec *Ant-SS(Vertex)*, et également pourquoi *Ant-SS(Clique)* a besoin de plus de cycles que *Ant-SS(Vertex)* pour converger.

Références

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms*. Addison Wesley, 1974.

[AK89] E.H.L. Aarts and J.H.M. Korst. *Simulated annealing and Boltzmann machines : a stochastic approach to combinatorial optimization and neural computing*. John Wiley & Sons, Chichester, U.K., 1989.

[APR99] J. Abello, P.M. Pardalos, and M.G.C. Resende. On maximum clique problems in very large graphs. In J.M. Abello, editor, *External Memory Algorithms*, volume 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 119–130. American Mathematical Society, Boston, MA, USA, 1999.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 580–592, 1994.

[ASG04] I. Alaya, C. Solnon, and K. Ghédira. Ant algorithm for the multi-dimensional knapsack problem. In *Proceedings of International Conference on Bioinspired Optimization Methods and their Applications (BIOMA 2004)*, pages 63–72, 2004.

[Bal94] S. Baluja. Population-based incremental learning : A method for integrating genetic search based function optimization and competitive learning. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[BGK02] R. Burke, S. Gustafson, and G. Kendall. A survey and analysis of diversity measures in genetic programming. In *GECCO 2002 : Proceedings of the Genetic and Evolutionary Computation Conference*, pages 716–723, New York, 2002. Morgan Kaufmann Publishers.

[BHS99] 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.

[BJ05] J.-F. Boulicaut and B. Jeudy. Constraint-based data mining. In *The Data Mining and Knowledge Discovery Handbook*, pages 399–416. Springer, 2005.

[Blu02] C. Blum. Ant colony optimization for the edge-weighted k-cardinality tree problem. In *GECCO 2002*, pages 27–34, 2002.

[BMR97] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2) :201–236, 1997.

[BP01] R. Battiti and M. Protasi. Reactive local search for the maximum clique problem. *Algorithmica*, 29(4) :610–637, 2001.

[Bun97] H. Bunke. On a relation between graph edit distance and maximum common subgraph. *PRL : Pattern Recognition Letters*, 18, 1997.

[CEv03] B.G.W. Craenen, A.E. Eiben, and J.I. van Hemert. Comparing evolutionary algorithms on binary constraint satisfaction problems. *IEEE Transactions on Evolutionary Computation*, 7(5) :424–444, 2003.

[CFG+96] D.A. Clark, J. Frank, I.P. Gent, E. MacIntyre, N. Tomv, and T. Walsh. Local search and the number of solutions. In *Proceedings of CP'96, LNCS 1118, Springer Verlag, Berlin, Germany*, pages 119–133, 1996.

[Chv83] V. Chvatal. *Linear Programming*. W.H. Freeman and Co, 1983.

[CKT91] P. Cheeseman, B. Kanelfy, and W. Taylor. Where the *really* hard problems are. In *Proceedings of IJCAI'91, Morgan Kaufmann, Sydney, Australia*, pages 331–337, 1991.

[DAGP90] J.-L. Deneubourg, S. Aron, S. Goss, and J.-M. Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behavior*, 3 :159–168, 1990.

[Dav95] A. Davenport. A comparison of complete and incomplete algorithms in the easy and hard regions. In *Proceedings of CP'95 workshop on Studying and Solving Really Hard Problems*, pages 43–51, 1995.

[DCG99] M. Dorigo, G. Di Caro, and L.M. Gambardella. Ant algorithms for discrete optimization. *Artificial Life*, 5(2) :137–172, 1999.

[DD99] 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.

[DG97] 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.

[DMC96] 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.

[Dor92] M. Dorigo. *Optimization, Learning and Natural Algorithms* (in Italian). PhD thesis, Dipartimento di Elettronica, Politecnico di Milano, Italy, 1992.

[DS04] M. Dorigo and T. Stuetzle. *Ant Colony Optimization*. MIT Press, 2004.

[EGN05] 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.

[Fag96] F. Fages. *Programmation Logique par Contraintes*. Collection Cours de l'Ecole Polytechnique. Ed. Ellipses, 1996.

[For96] S. Fortin. The graph isomorphism problem. Technical report, Dept of Computing Science, Univ. Alberta, Edmonton, Alberta, Canada, 1996.

[FR89] T.A. Feo and M.G.C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letter*, 8 :67–71, 1989.

[Fre78] E. Freuder. Synthesizing constraint expressions. *Communications ACM*, 21(11) :958–966, 1978.

[FRPT99] C. Fonlupt, D. Robilliard, P. Preux, and E. Talbi. Fitness landscape and performance of meta-heuristics, 1999.

[FS03] S. Fenet and C. Solnon. Searching for maximum cliques with ant colony optimization. In *Applications of Evolutionary Computing, Proceedings of EvoWorkshops 2003 : EvoCOP, EvoIASP, EvoSTim*, volume 2611 of *LNCS*, pages 236–245. Springer-Verlag, 2003.

[FW92] E. Freuder and R. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58 :21–70, 1992.

[GH97] P. Galinier and J.-K. Hao. Tabu search for maximal constraint satisfaction problems. In *Proceedings of CP'97, LNCS 1330, Springer Verlag, Berlin, Germany*, pages 196–208, 1997.

[GJ79] M.R. Garey and D.S. Johnson. *A guide to the theory of NP-Completeness*. Freeman, 1979.

[GL93] F. Glover and M. Laguna. Tabu search. In C.R. Reeves, editor, *Modern Heuristics Techniques for Combinatorial Problems*, pages 70–141. Blackwell Scientific Publishing, Oxford, UK, 1993.

[GLC04] A. Grosso, M. Locatelli, and F. Della Croce. Combining swaps and node weights in an adaptive greedy approach for the maximum clique problem. *Journal of Heuristics*, 10(2) :135–152, 2004.

[GMPW96] I.P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In *Proceedings of AAAI-96, AAAI Press, Menlo Park, California*, 1996.

[GPS03] 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.

[GTD99] 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.

[GW99] 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.

[Hog96] T. Hogg. Refining the phase transition in combinatorial search. *Artificial Intelligence*, pages 127–154, 1996.

[Hog98] T. Hogg. Exploiting problem structure as a search heuristic. *Intl. J. of Modern Physics C*, 9 :13–29, 1998.

[Hol75] J. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[HR96] T. Asselmeyer H. Rosé, W. Ebeling. Density of states - a measure of the difficulty of optimisation problems. In *Proceedings of PPSN'96*, pages 208–217. LNCS 1141, Springer Verlag, 1996.

[HRTB00] R. Hadji, M. Rahoual, E.G. Talbi, and V. Bachelet. Ant colonies for the set covering problem. In *Proceedings of ANTS 2000*, pages 63–66, 2000.

[HW74] J.E. Hopcroft and J.-K. Wong. Linear time algorithm for isomorphism of planar graphs. *Proc.* 6^{th} *Annual ACM Symp. theory of Computing*, ACM Press :172–184, 1974.

[JF95] T. Jones and S. Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *Proceedings of International Conference on Genetic Algorithms, Morgan Kaufmann, Sydney, Australia*, pages 184–192, 1995.

[JL02] N. Jussien and O. Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139, 2002.

[JS01] A. Jagota and L.A. Sanchis. Adaptive, restart, randomized greedy heuristics for maximum clique. *Journal of Heuristics*, 7(6) :565–585, 2001.

[LL01] P. Larranaga and J.A. Lozano. *Estimation of Distribution Algorithms. A new tool for Evolutionary Computation*. Kluwer Academic Publishers, 2001.

[LM99] G. Leguizamon and Z. Michalewicz. A new version of ant system for subset problem. In *Proceedings of Congress on Evolutionary Computation*, pages 1459–1464, 1999.

[LMS02] H.R. Lourenco, O. Martin, and T. Stuetzle. *Handbook of Metaheuristics*, chapter Iterated Local Search, pages 321–353. Kluwer Academic Publishers, 2002.

[Luk82] E.M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer System Science*, 25 :42–65, 1982.

[Mar97] E. Marchiori. Combining constraint processing and genetic algorithms for constraint satisfaction problems. In *Proceedings of the 7th International Conference on Genetic Algorithms*, pages 330–337, 1997.

[Mar02] E. Marchiori. Genetic, iterated and multistart local search for the maximum clique problem. In *Applications of Evolutionary Computing, Proceedings of EvoWorkshops 2002 : EvoCOP, EvoIASP, EvoSTim*, volume 2279 of *LNCS*, pages 112–121. Springer-Verlag, 2002.

[MF99] P. Merz and B. Freisleben. Fitness landscapes and memetic algorithm design. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 245–260. McGraw Hill, UK, 1999.

[MH97] N. Mladenović and P. Hansen. Variable neighborhood search. *Comps. in Oerations Research*, 24 :1097–1100, 1997.

[MJ01] R.W. Morrison and K.A. De Jong. Measurement of population diversity. In *5th International Conference EA 2001*, volume 2310 of *LNCS*, pages 31–41. Springer-Verlag, 2001.

[Mos89] P. Moscato. On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts : Towards Memetic Algorithms. Technical Report Caltech Concurrent Computation Program, Report. 826, California Institute of Technology, Pasadena, California, USA, 1989.

[MPSW98] E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. Random constraints satisfaction : theory meets practice. In *CP98, LNCS 1520*, pages 325–339. Springer Verlag, Berlin, Germany, 1998.

[MS98] K. Marriott and P.J. Stuckey. *Programming with Constraints : an Introduction*. MIT Press, 1998.

[MT97] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, pages 241–258, 1997.

[NTG04] 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.

[Pap94] C. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[PGCP99] M. Pelikan, D.E. Goldberg, and E. Cantú-Paz. BOA : The Bayesian optimization algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99*, volume I, pages 525–532. Morgan Kaufmann Publishers, San Fransisco, CA, 13-17 1999.

[PR02] L. Pitsoulis and M. Resende. *Handbook of Applied Optimization*, chapter Greedy randomized adaptive search procedures, pages 168–183. Oxford University Press, 2002.

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3) :268–299, August 1993.

[PV04] C. Pralet and G. Verfaillie. Travelling in the world of local searches in the sapce of partial assignments. In *Proceedings of CP-AI-OR 2004*, volume 3011 of *LNCS*, pages 240–255. Springer Verlag, 2004.

[Rég95] J.-C. Régin. *Développement d'Outils Algorithmiques pour l'Intelligence Artificielle. Application à la Chimie Organique*. PhD thesis, Université Montpellier II, 1995.

[SB06] C. Solnon and D. Bridge. *System Engineering using Particle Swarm Optimization*, chapter An Ant Colony Optimization Meta-Heuristic for Subset Selection Problems, pages 7–29. Nova Science Publisher, 2006.

[SBC01] A. Sidaner, O. Bailleux, and J.-J. Chabrier. Measuring the spatial dispersion of evolutionist search processes : application to walksat. In *5th International Conference EA 2001*, volume 2310 of *LNCS*, pages 77–90. Springer-Verlag, 2001.

[SF06] C. Solnon and S. Fenet. A study of aco capabilities for solving the maximum clique problem. *Journal of Heuristics*, 12(3) :155–180, 2006.

[SFV95] T. Shiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems : Hard and easy problems. In *Proceedings of IJCAI-95, MIT Press, Cambridge, MA*, pages 631–637, 1995.

[SH00] T. Stützle and H.H. Hoos. $\mathcal{MAX} - \mathcal{MIN}$ Ant System. *Journal of Future Generation Computer Systems, special isuue on Ant Algorithms*, 16 :889–914, 2000.

[Shm95] D.B. Shmoys. Computing near-optimal solutions to combinatorial optimization problems. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 20 :355–397, 1995.

[SLM92] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of AAAI'92, AAAI Press, Menlo Park, California*, pages 440–446, 1992.

[Sol02] C. Solnon. Ants can solve constraint satisfaction problems. *IEEE Transactions on Evolutionary Computation*, 6(4) :347–357, 2002.

[SSG05] O. Sammoud, C. Solnon, and K. Ghédira. Ant algorithm for the graph matching problem. In *5th European Conference on Evolutionary Computation in Combinatorial Optimization*, volume 3448 of *LNCS*, pages 213–223. Springer Verlag, 2005.

[SSSG06] O. Sammoud, Sébastien Sorlin, C. Solnon, and K. Ghédira. A comparative study of ant colony optimization and reactive search for graph matching problems. In *6th European Conference on Evolutionary Computation in Combinatorial Optimization*, volume 3906 of *LNCS*, pages 234–246. Springer Verlag, 2006.

[Sta95] P.F. Stadler. Towards a theory of landscapes. In *Complex Systems and Binary Networks*, volume 461, pages 77–163. Springer Verlag, 1995.

[TB05] F. Tarrant and D. Bridge. When ants attack : Ant algorithms for constraint satisfaction problems. *à paraître dans Artificial Intelligence Review*, 2005.

[Tsa93] E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, UK, 1993.

[vHB02] J.I. van Hemert and T. Bäck. Measuring the searched space to guide efficiency : The principle and evidence on constraint satisfaction. In *Proceedings of the 7th International Conference on Parallel Problem Solving from Nature*, volume 2439 of *LNCS*, pages 23–32, Berlin, 2002. Springer-Verlag.

[vHS04] J.I. van Hemert and C. Solnon. A study into ant colony optimization, evolutionary computation and constraint programming on binary constraint satisfaction problems. In *Evolutionary Computation in Combinatorial Optimization (EvoCOP 2004)*, volume 3004 of *LNCS*, pages 114–123. Springer-Verlag, 2004.

[ZBMD04] M. Zlochin, M. Birattari, N. Meuleau, and M. Dorigo. Model-based search for combinatorial optimization : A critical survey. *Annals of Operations Research*, 131 :373–395, 2004.