Investigating ACO capabilities for solving the Maximum Clique Problem

Christine Solnon and Serge Fenet
LIRIS CNRS FRE 2672, University Lyon I
Nautibus, 43 Bd du 11 novembre, 69622 Villeurbanne cedex, France
{christine.solnon,serge.fenet}@liris.cnrs.fr

May 24, 2004

Abstract

This paper investigates the capabilities of the Ant Colony Optimization (ACO) meta-heuristic for solving the maximum clique problem, the goal of which is to find a largest set of pairwise adjacent vertices in a graph. We propose two ACO algorithms for this problem. Basically, these algorithms successively generate maximal cliques through the repeated addition of vertices into partial cliques, and both of them use "pheromone trails" as a greedy heuristic to choose, at each step, the next vertex to enter the clique. However, these two algorithms differ in the way pheromone trails are laid and exploited, i.e., on edges or on vertices of the graph.

We illustrate and compare the behavior of the two proposed ACO algorithms on a representative benchmark instance and we study the impact of pheromone on the solution process. We consider two measures —the re-sampling and the dispersion ratio— for providing an insight into the two algorithms performances. We also study the benefit of integrating a local search procedure within the proposed ACO algorithms, and we show that this improves the solution process. Finally, we compare ACO performances with three representative heuristic approaches, showing that it obtains competitive results.

1 Introduction

The maximum clique problem is a classical combinatorial optimization problem that has important applications in different domains, such as coding theory, fault diagnosis, or computer vision. Moreover, many important problems —such as constraint satisfaction, subgraph isomorphism, or vertex covering problems— are easily reducible to this maximum clique problem.

Given a non-directed graph $G = (V, E)$, such that V is a set of vertices, and $E \subseteq V \times V$ is a set of edges, a *clique* is a set of vertices $\mathcal{C} \subseteq V$ such that every

pair of distinct vertices of $\mathcal{C}$ is connected with an edge in G, i.e., the subgraph induced by $\mathcal{C}$ is complete. A clique is *partial* if it is strictly included in another clique; otherwise it is *maximal*. The goal of the maximum clique problem is to find a *maximum clique*, i.e., a clique of maximum cardinality.

The maximum clique problem is very closely related to the maximum independent (or stable) set problem, the goal of which is to find the maximum subset of V such that no two vertices of the subset are pairwise adjacent: a maximum clique of a graph $G = (V, E)$ is a maximum independent set of the complement graph $G' = (V, V \times V - E)$ of G.

The clique problem, the goal of which is to decide if a graph contains a clique of size k, is one of the first problems shown to be NP-complete in [24]. More generally, the problem of finding a maximum clique is NP-hard, and does not admit polynomial-time approximation algorithms (unless P=NP) [5]. Hence, complete approaches —usually based on a branch-and-bound tree search— become intractable when the number of vertices increases, and much effort has recently been directed on heuristic incomplete approaches.

Heuristic approaches for the maximum clique problem

Heuristic approaches leave out exhaustivity and use heuristics to guide the search towards promising areas of the search space. They run in polynomial time and quickly find "rather good" solutions, that may be optimal, but there is no formal guarantee of performance.

Many heuristic approaches are based on sequential greedy heuristics e.g., [22, 14, 3, 4, 21, 19]. The idea is to build maximal cliques, starting from an empty clique, and then iterating through the repeated addition of vertices. Decisions on which vertex to be added are made with respect to a greedy heuristic such as, e.g., choosing the vertex that has the highest degree among candidate vertices. To avoid the usual greedy traps, greedy heuristics can be improved by injecting a mild amount of randomization, combined with multiple restarts. Also, weights used by the greedy heuristic may be adapted from restart to restart as proposed, e.g., in [21, 19].

To improve the quality of a constructed clique, local search can be used to explore its neighborhood, i.e., the set of cliques that can be obtained by removing and/or adding a given number of vertices: local search iteratively moves in the search space composed of all cliques, from a clique to one of its (best) neighbors. To avoid being trapped in local optima, where all neighbors are cliques of smaller sizes, local search may be combined with some advanced meta-heuristics. For example, in [2, 20], Simulated Annealing is used to jump out of local optima by allowing moves towards smaller cliques with a probability proportional to a decreasing temperature parameter; in [15, 18], Tabu Search is used to prevent local search from cycling through a small set of good but suboptimal cliques, by keeping track in a tabu list of forbidden moves between cliques; in [4], Reactive Search enhances tabu search by reactively adapting the size of the tabu list with respect to the need of diversification; in [27], local search is combined with a Genetic Algorithm that allows to escape from local

maxima by applying crossing-over and mutation operators to a population of maximal cliques.

The Ant Colony Optimization meta-heuristic

In this paper, we investigate the capabilities of another meta-heuristic —Ant Colony Optimization (ACO) [10, 9]— for solving the maximum clique problem. The basic idea of ACO is to model the problem to solve as the search for a minimum cost path in a graph, and to use artificial ants to search for good paths.

The behavior of artificial ants is inspired from real ants: artificial ants lay pheromone trails on components of the graph and they choose their path with respect to probabilities that depend on pheromone trails that have been previously laid; these pheromone trails progressively decrease by evaporation. Intuitively, this indirect stigmergetic communication mean aims at giving information about the quality of path components in order to attract ants, in the following iterations, towards the corresponding areas of the search space. Indeed, for many combinatorial problems, a study of the search space landscape shows a correlation between solution quality and the distance to optimal solutions [23, 28, 33].

The first ant algorithm to be applied to a discrete optimization problem has been proposed by Dorigo in [8]. The problem chosen for the first experiments was the Traveling Salesman Problem and, since then, this problem has been widely used to investigate the solving capabilities of ants [12, 11]. The ACO meta-heuristic, described in [10, 9], is a generalization of these first ant based algorithms, and it has been successfully applied to different hard combinatorial optimization problems such as quadratic assignment problems [16, 26], vehicle routing problems [6, 17], and constraint satisfaction problems [31, 32].

We have proposed in [13] a first ACO algorithm for the maximum clique problem. The contribution of this paper with respect to this preliminary work mainly concerns (i) the definition of a second ACO algorithm, that differs in the way pheromone is laid and exploited; (ii) the introduction of diversity measures in order to provide an insight into algorithms performances; and (iii) an investigation of the benefit of combining ACO with local search for this problem.

Overview of the paper

Section 2 describes the two ACO algorithms —`Vertex-AC` and `Edge-AC`— for the maximum clique problem. In both algorithms, pheromone trails are used as a greedy heuristic for choosing, at each step, the next vertex to enter the clique. However, in `Vertex-AC`, pheromone trails are laid on vertices, and the choice of vertices directly depends on the quantity of pheromone laying on them, whereas in `Edge-AC`, pheromone trails are laid on edges, and the choice of vertices depends on pheromone trails laying on edges connecting candidate vertices with the vertices of the clique under construction.

In Section 3, we illustrate and compare the behavior of these two algorithms on a representative benchmark instance. We introduce two measures in order

to provide a deeper insight into algorithms behavior: the re-sampling ratio allows us to measure how often algorithms re-sample the search space, whereas the dispersion ratio allows us to quantify the differences between the cliques successively computed during the solution process.

In section 4, we show how local search can be combined with our two ACO algorithms, and we study on a representative benchmark instance the influence of local search on the solution process.

Section 5 experimentally compares the four different ACO algorithms on a set of benchmark graphs, showing that `Edge-AC` outperforms `Vertex-AC`, and that the integration of local search improves solutions quality on a majority of instances.

Finally, section 6 compares our approach with other heuristic approaches. We have more particularly chosen for this comparison three recent and representative algorithms within different classes of heuristic approaches: `RLS` [4], which uses reactive local search and obtains the best known results on most benchmark instances, `DAGS` [19], which combines an adaptive greedy approach with swap local moves and obtains the best known results on some other instances, and `GLS` [27], which combines a genetic approach with local search.

2 ACO for the maximum clique problem

In ACO algorithms, ants lay pheromone trails on components of the best constructed solutions in order to attract other ants towards the corresponding area of the search space. To solve a new problem with ACO, one mainly has to define the pheromone laying procedure —i.e., decide on which components of constructed solutions ants should lay pheromone trails— and define the solution construction procedure —i.e., decide how to exploit these pheromone trails when constructing new solutions.

Hence, to solve the maximum clique problem with ACO, the key point is to decide which components of the constructed cliques should be rewarded, and how to exploit these rewards when constructing new cliques. Indeed, given a maximal clique $\mathcal{C}_i$, one can lay pheromone trails either on the vertices of $\mathcal{C}_i$, or on the edges connecting every pair of different vertices of $\mathcal{C}_i$:

- when laying pheromone on the vertices of $\mathcal{C}_i$, the idea is to increase the desirability of each vertex of $\mathcal{C}_i$ so that, when constructing a new clique, these vertices will be more likely to be selected;
- when laying pheromone on the edges of $\mathcal{C}_i$, the idea is to increase the desirability of choosing together two vertices of $\mathcal{C}_i$ so that, when constructing a new clique $\mathcal{C}_k$, the vertices of $\mathcal{C}_i$ will be more likely to be selected *if* $\mathcal{C}_k$ *already contains some vertices of* $\mathcal{C}_i$. More precisely, the more $\mathcal{C}_k$ will contain vertices of $\mathcal{C}_i$, the more the other vertices of $\mathcal{C}_i$ will be attractive.

A goal of this paper is to compare these two different ways of using pheromone and, therefore, we introduce two algorithms: Vertex-Ant-Clique (`Vertex-AC`),

Search of an approximate maximum clique in a graph $G = (V, E)$:

Initialize pheromone trails to τ_{max}
repeat the following cycle:
  for each ant k **in** $1..nbAnts$, construct a maximal clique $\mathcal{C}_k$ as follows:
    Randomly choose a first vertex $v_i \in V$
    $\mathcal{C}_k \leftarrow \{v_i\}$
    $Candidates \leftarrow \{v_j \in V \mid (v_i, v_j) \in E\}$
    while $Candidates \neq \emptyset$ **do**
      Choose a vertex $v_i \in Candidates$ with probability $p(v_i)$
      $\mathcal{C}_k \leftarrow \mathcal{C}_k \cup \{v_i\}$
      $Candidates \leftarrow Candidates \cap \{v_j \mid (v_i, v_j) \in E\}$
    end while
  end for
  Update pheromone trails w.r.t. $\{\mathcal{C}_1, \ldots, \mathcal{C}_{nbAnts}\}$
  if a pheromone trail if lower than τ_{min} **then** set it to τ_{min}
  if a pheromone trail if greater than τ_{max} **then** set it to τ_{max}
until maximum number of cycles reached **or** optimal solution found
return the largest constructed clique since the beginning

Figure 1: Generic algorithmic scheme of `Vertex-AC` and `Edge-AC`

where pheromone is laid on vertices, and Edge-Ant-Clique (`Edge-AC`), where pheromone is laid on edges.

In this section, we first describe the generic algorithmic scheme that is common to the two algorithms. Then, we describe the three points on which the two algorithms differ, i.e., the definition of pheromonal components, the exploitation of pheromone trails when constructing cliques, and the pheromone updating process. Finally, we compare time complexities of the two algorithms.

2.1 Generic Ant-Clique algorithmic scheme

The two algorithms, `Vertex-AC` and `Edge-AC`, both follow the same generic algorithmic scheme displayed in Figure 1. At each cycle of this algorithm, every ant constructs a maximal clique. It first randomly chooses an initial vertex to enter the clique, and then iteratively adds vertices that are chosen within a set *Candidates* that contains all the vertices that are connected to every vertex of the partial clique under construction. This choice is performed randomly with respect to probabilities that are defined in section 2.3. Once each ant has constructed a clique, pheromone trails are updated, as described in section 2.4. The algorithms stop either when an ant has found a maximum clique (when the optimal bound is known), or when a maximum number of cycles has been performed.

For both algorithms, we have more particularly borrowed features from the $\mathcal{MAX} - \mathcal{MIN}$ Ant System [33]: we explicitly impose lower and upper bounds

τ_{min} and τ_{max} on pheromone trails (with $0 < \tau_{min} < \tau_{max}$). The goal is to favor a larger exploration of the search space by preventing the relative differences between pheromone trails from becoming too extreme during processing. Also, pheromone trails are set to τ_{max} at the beginning of the search, thus achieving a higher exploration of the search space during the first cycles.

2.2 Definition of pheromonal components

Pheromone trails are laid by ants on components of the graph $G = (V, E)$ in which they are looking for a maximum clique.

- In `Vertex-AC`, ants lay pheromone trails on the vertices V of the graph. The quantity of pheromone on a vertex $v_i \in V$ is noted τ_i. Intuitively, this quantity represents the learned desirability to select v_i when constructing a clique.
- In `Edge-AC`, ants lay pheromone trails on the edges E of the graph. The quantity of pheromone on an edge $(v_i, v_j) \in E$ is noted τ_{ij}. Intuitively, this quantity represents the learn desirability to select v_i when constructing a clique that already contains v_j. Notice that since the graph is not directed, $\tau_{ij} = \tau_{ji}$.

2.3 Exploitation of pheromone trails

Pheromone trails are used to choose vertices when constructing cliques: at each step, a vertex v_i is randomly chosen within the set *Candidates* with respect to a probability $p(v_i)$. This probability is defined proportionally to pheromone factors, i.e.,

$$p(v_i) = \frac{[\tau fact(v_i)]^\alpha}{\sum_{v_j \in Candidates} [\tau fact(v_j)]^\alpha}$$

where α is a parameter which weights pheromone factors, and $\tau fact(v_i)$ is the pheromone factor of vertex v_i. This pheromone factor depends on the quantity of pheromone laying on pheromonal components of the graph:

- in `Vertex-AC`, it depends on the quantity of pheromone laid on the candidate vertex, i.e.,
$$\tau fact(v_i) = \tau_i$$
- in `Edge-AC`, it depends on the quantity of pheromone laid on edges connecting the vertices that already are in the partial clique and the candidate vertex: let $\mathcal{C}_k$ be the partial clique under construction, the pheromone factor of a candidate vertex v_i is
$$\tau fact(v_i) = \sum_{v_j \in \mathcal{C}_k} \tau_{ij}$$

Note that this pheromone factor can be computed in an incremental way: once the first vertex v_i has been randomly chosen, for each candidate vertex v_j that is adjacent to v_i, the pheromone factor $\tau fact(v_j)$ is initialized to τ_{ij}; then, each time a new vertex v_k is added to the clique, for each candidate vertex v_j that is adjacent to v_k, the pheromone factor $\tau fact(v_j)$ is incremented by τ_{kj}.

One should remark that, for both `Vertex-AC` and `Edge-AC`, the probability of choosing a vertex v_i only depends on a pheromone factor and not on some other heuristic factor that locally evaluates the quality of the candidate vertex, as usually in ACO algorithms. Actually, we have experimented the greedy heuristic proposed, e.g., in [22, 4, 19], the idea of which being to favor vertices with largest degrees in the "residual graph" (the subgraph induced by the set of candidate vertices that can extend the partial clique under construction). The underlying motivation is that a larger degree implies a larger number of candidates once the vertex has been added to the current clique. When using no pheromone, or at the beginning of the search when all pheromone trails have the same value, this heuristic actually allows ants to construct larger cliques than a random choice. However, when combining it with pheromone learning, we have noticed that after a hundred or so cycles, we obtain larger cliques without using the heuristic than when using it.

2.4 Updating pheromone trails

Once each ant has constructed a clique, the amount of pheromone laying on pheromonal components is updated according to the ACO meta-heuristic. First, all amounts are decreased in order to simulate evaporation. This is done by multiplying the quantity of pheromone laying on each pheromonal component by a pheromone persistence rate ρ such that $0 \leq \rho \leq 1$. Then, the best ant of the cycle deposits pheromone. More precisely, let $\mathcal{C}_k \in \{\mathcal{C}_1, \ldots, \mathcal{C}_{nbAnts}\}$ be the largest clique built during the cycle (if there are several largest cliques, ties are randomly broken), and $\mathcal{C}_{best}$ be the largest clique built since the beginning of the run. The quantity of pheromone laid by ant k is inversely proportional to the gap of size between $\mathcal{C}_k$ and $\mathcal{C}_{best}$, i.e., it is equal to $1/(1+ \mid \mathcal{C}_{\text{best}} \mid - \mid \mathcal{C}_k \mid)$.

This quantity of pheromone is deposited on the pheromonal components of $\mathcal{C}_k$, i.e.,

- in `Vertex-AC`, it is deposited on each vertex of $\mathcal{C}_k$,
- in `Edge-AC`, it is deposited on each edge connecting two different vertices of $\mathcal{C}_k$.

2.5 Time complexity of Vertex-AC and Edge-AC

Let *nbMaxCycles* and *nbAnts* respectively be the maximum number of cycles and the number of ants. In the worst case (if an optimal solution is not found), both `Vertex-AC` and `Edge-AC` will have to construct *nbMaxCycles.nbAnts* maximal cliques, and to perform *nbMaxCycles* pheromone updating steps.

To construct a maximal clique, the two proposed algorithms nearly perform the same number of operations. Indeed, to construct a clique $\mathcal{C}$, both algorithms perform $\mid \mathcal{C} \mid -1$ times the "while" loop. At each iteration of this loop, both algorithms have to: (i) compute probabilities for all candidates, (ii) choose a vertex with respect to these probabilities, and (iii) update the list of candidates. These three steps take a linear time with respect to the number of candidate vertices[1]. At the first iteration, the number of candidate vertices is equal to the degree of the initial vertex, and it decreases at each iteration. Hence, in the worst case, the complexity of the construction of a clique in a graph G, for both `Vertex-AC` and `Edge-AC`, is in $\mathcal{O}(\omega(G).max_d(G))$, where $\omega(G)$ is the size of the maximum clique of G, and $max_{d^\circ}(G)$ is the maximum vertex degree in G. Note that both $\omega(G)$ and $max_{d^\circ}(G)$ are bounded by the number of vertices of G.

To update pheromone trails laying on pheromonal components of a graph $G = (V, E)$, the two algorithms perform a different number of operations:

- In `Vertex-AC`, the evaporation step requires $\mathcal{O}(\mid V \mid)$ operations and the reward of a clique $\mathcal{C}$ requires $\mathcal{O}(\mid \mathcal{C} \mid)$ operations. As $\mid \mathcal{C} \mid \leq \mid V \mid$, the whole pheromone updating step requires $\mathcal{O}(\mid V \mid)$ operations.
- In `Edge-AC`, the evaporation step requires $\mathcal{O}(\mid E \mid)$ operations and the reward of a clique $\mathcal{C}$ requires $\mathcal{O}(\mid \mathcal{C} \mid^2)$ operations. As $\mid \mathcal{C} \mid^2 \leq \mid E \mid$, the whole pheromone updating step requires $\mathcal{O}(\mid E \mid)$ operations.

Hence, the overall time complexity of `Vertex-AC` is in

$$\mathcal{O}(nbCycles(nbAnts.\omega(G).max_{d^\circ}(G) + \mid V \mid))$$

whereas the overall time complexity for performing one cycle of `Edge-AC` is in

$$\mathcal{O}(nbCycles(nbAnts.\omega(G).max_{d^\circ}(G) + \mid E \mid))$$

3 Experimental study on the `C500.9` instance

When solving a combinatorial optimization problem with a heuristic approach such as evolutionary computation or ACO, one usually has to find a compromise between two dual goals. On one hand, one has to intensify the search around the most "promising" areas, that are usually close to the best solutions found so far. On the other hand, one has to diversify the search and favor exploration in order

[1] Remember that, as pointed out in 2.3, pheromone factors in `Edge-AC` are computed in an incremental way, i.e., each time a new vertex is added to the clique, the pheromone factor of each candidate vertex is updated by a simple addition.

to discover new, and hopefully more successful, areas of the search space. The behavior of ants with respect to this intensification/diversification duality can be influenced by modifying parameter values [1]. In particular, diversification can be emphasized either by decreasing the value of the pheromone factor weight α —so that ants become less sensitive to pheromone trails— or by increasing the value of the pheromone persistence rate ρ —so that pheromone evaporates more slowly. When increasing the exploratory ability of ants in this way, one usually finds better solutions, but as a counterpart it takes longer time to find them.

In this section, we first illustrate the influence of these two parameters on solutions quality for the two proposed algorithms. Then, we provide an insight into the influence of these parameters on intensification/diversification by means of two diversity measures. This study is performed on instance `C500.9` of the DIMACS benchmark set. This instance, which has 500 vertices and 112,332 edges, is a rather difficult one, for which our two ACO algorithms have difficulties in finding the maximum clique (which contains 57 vertices).

3.1 Influence of α and ρ on the solution process

Figure 2 illustrates the influence of the pheromone factor weight α and the pheromone persistence rate ρ when solving the `C500.9` instance. On this figure, one can first note that when $\alpha = 0$ and $\rho = 1$, the best constructed cliques are much smaller: in this case, pheromone is totally ignored and the resulting search process performs as a random one so that after 500 or so cycles, the size of the best clique nearly stops increasing, and hardly reaches 48 vertices. This shows that pheromone actually improves the solution process with respect to a pure random algorithm.

For both algorithms, we remark that α and ρ influence the solution process in a very similar way: when α increases or ρ decreases, ants are able to find better solutions quicker. However, after a thousand or so cycles, both algorithms find better solutions when α is set to 1 and ρ to 0.99 or 0.995 than when α is set to 2 or ρ to 0.98. Hence, the setting of α and ρ let us balance between two main tendencies. On one hand, when limiting the influence of pheromone with a low pheromone factor weight and a high pheromone persistence rate, the quality of the final solution is better, but the time needed to converge on this value is also higher. On the other hand, when increasing the influence of pheromone with a higher pheromone factor weight and a lower pheromone persistence rate, ants find better solutions during the first cycles, but after 500 or so cycles, they are no longer able to find better solutions.

When comparing `Edge-AC` (upper curves) with `Vertex-AC` (lower curves), one can note that after 2500 cycles, the best cliques found by `Edge-AC` are slightly larger than the best ones found by `Vertex-AC`. For example, when setting α to 1 and ρ to 0.99, the average size of the best constructed cliques is equal to 55.6 for `Edge-AC` and to 55.2 for `Vertex-AC`. Moreover, three runs of `Edge-AC` (over the fifty performed runs) have been able to find a clique of 57 vertices, whereas `Vertex-AC` only found cliques of 56 or less vertices.

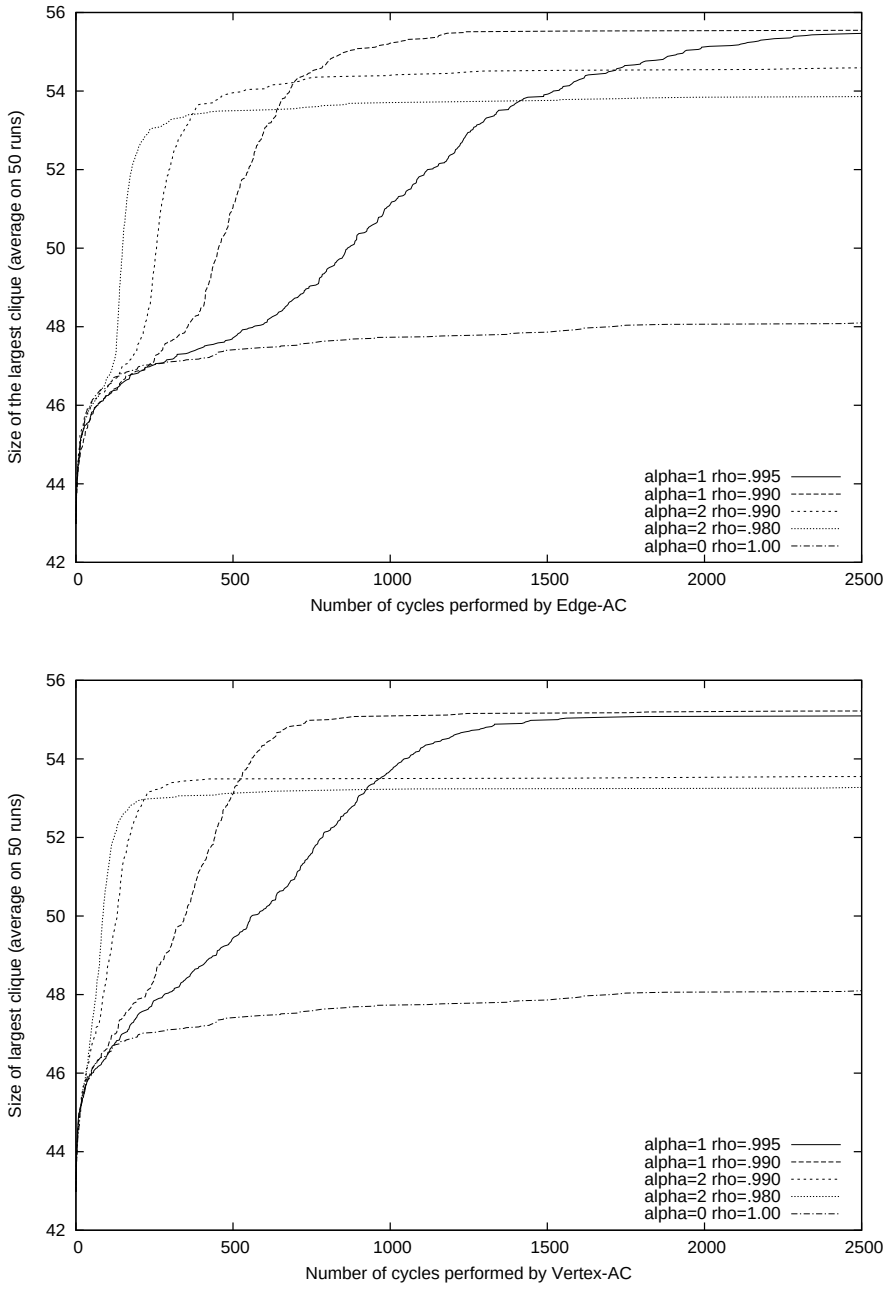

Figure 2: Influence of pheromone on the solution process of `Edge-AC` (upper curves) and `Vertex-AC` (lower curves): each curve plots the evolution of the size of the best constructed clique (average on 50 runs), when the number of cycles increases, for a given setting of α and ρ. The other parameters have been set to nbAnts $= 30$, $\tau_{min} = 0.01$, and $\tau_{max} = 6$.

However, if Edge-AC is able to find better cliques, it needs more cycles to converge towards them. For example, when setting α to 1 and ρ to 0.99, the average number of cycles needed to find the best clique respectively is equal to 923 for Edge-AC and to 722 for Vertex-AC. Moreover, as discussed in section 2.5, each cycle of Edge-AC takes a longer time to perform than Vertex-AC: on this instance, in one second of CPU-time, Edge-AC and Vertex-AC respectively perform 104 and 188 cycles. Hence, to find their best cliques, Edge-AC and Vertex-AC respectively need 8.9 and 3.8 seconds on average.

As a conclusion, on instance C500.9, the two algorithms behave in a rather similar way at run time with respect to pheromone parameters, and a good compromise between solution quality and CPU time is reached when α is set to 1 and ρ to 0.99. With such a parameter setting, Vertex-AC is more than twice as fast as Edge-AC to find its best clique, but the best clique found by Edge-AC is slightly better, on average, than the one found by Vertex-AC, and Edge-AC has been able to find the best known solution for 6% of its runs, whereas Vertex-AC never reached it.

3.2 Measuring the diversity at run time

To provide an insight into algorithms performances, and to explicit the influence of pheromone on the capability of ants to explore the search space, we now propose to measure the diversity of the computed solutions at run time. Many diversity measures have been introduced for evolutionary approaches. Indeed, maintaining population diversity is a key point to prevent from premature convergence and stagnation. Most commonly used diversity measures include the number of different fitness values, the number of different structural individuals, and distances between individuals [7].

To measure the diversification effort of Ant-clique at run time, we propose in this paper to compute the re-sampling and the diversification ratio.

Re-sampling ratio. This measure is used, e.g., in [35, 34], in order to get insight into how efficient algorithms are in sampling the search space: if we define *nbDiff* as the set of unique candidate solutions generated by an algorithm over a whole run and *nbTot* as the total number of generated candidate solutions, then the re-sampling ratio is defined as (*nbTot* - *nbDiff*)/*nbTot*. Values close to 0 correspond with an efficient search, i.e., not many duplicate candidate solutions are generated, whereas values close to 1 indicate a stagnation of the search process around a small set of solutions.

Table 1 provides an insight into the two algorithms performances by means of this re-sampling ratio. This table shows that Vertex-AC is less efficient than Edge-AC in sampling the search space as it often generates cliques that have already been previously generated. For example, when setting α to 1 and ρ to 0.99, 7% of the cliques computed by Vertex-AC during the first thousand of cycles had already been computed. This re-sampling ratio increases very quickly and reaches 48% at cycle 2500, i.e., nearly half of the cliques computed during each run already had been computed before. As a comparison, with the same

Table 1: Evolution of the re-sampling ratio for `Edge-AC` and `Vertex-AC`. Each line successively displays the setting of α and ρ, and the re-sampling ratio after 500, 1000, 1500, 2000, and 2500 cycles (average on 50 runs). The other parameters have been set to $nbAnts = 30$, $\tau_{min} = 0.01$, $\tau_{max} = 6$.

Re-sampling ratio for `Edge-AC`

Number of 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

Re-sampling ratio for `Vertex-AC`

Number of 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

parameter setting, `Edge-AC` nearly never computes twice a same clique during a same run, so that it actually explores twice more states in the search space.

The re-sampling ratio allows one to quantify the size of the searched space, and shows that `Edge-AC` has a higher search capability than `Vertex-AC` (for the considered `C500.9` instance). However, the re-sampling ratio gives no information about the distribution of the computed cliques within the whole search space. Hence, to provide a complementary insight into algorithms performances, we propose to compute a similarity ratio which indicates how much the computed cliques are similar.

Similarity ratio. This ratio corresponds to the pair-wise population diversity measure, introduced for genetic approaches, e.g., in [30, 29]. More precisely, we define the similarity ratio of a set of cliques S by the average number of vertices that are shared by any pair of cliques in S, divided by the average size of the cliques of S. Hence, this ratio is equal to one if all the cliques of S are identical, whereas it is equal to zero if the intersection of every pair of cliques of S is empty. Note that this ratio can be computed very quickly by maintaining an array `freq` such that, for every vertex $v_i \in V$, `freq[i]` is equal to the number of cliques of S which have selected vertex v_i. In this case, the similarity ratio of S is equal to $\frac{\sum_{v_i \in V}(\texttt{freq[i]}.(\texttt{freq[i]}-1))}{(|S|-1).\sum_{\mathcal{C}_k \in S}|\mathcal{C}_k|}$ and it can be easily computed in an incremental way while constructing cliques.

Figure 3 plots the similarity ratio of the cliques computed every 50 cycles, thus giving an information about the distribution of the set of cliques computed

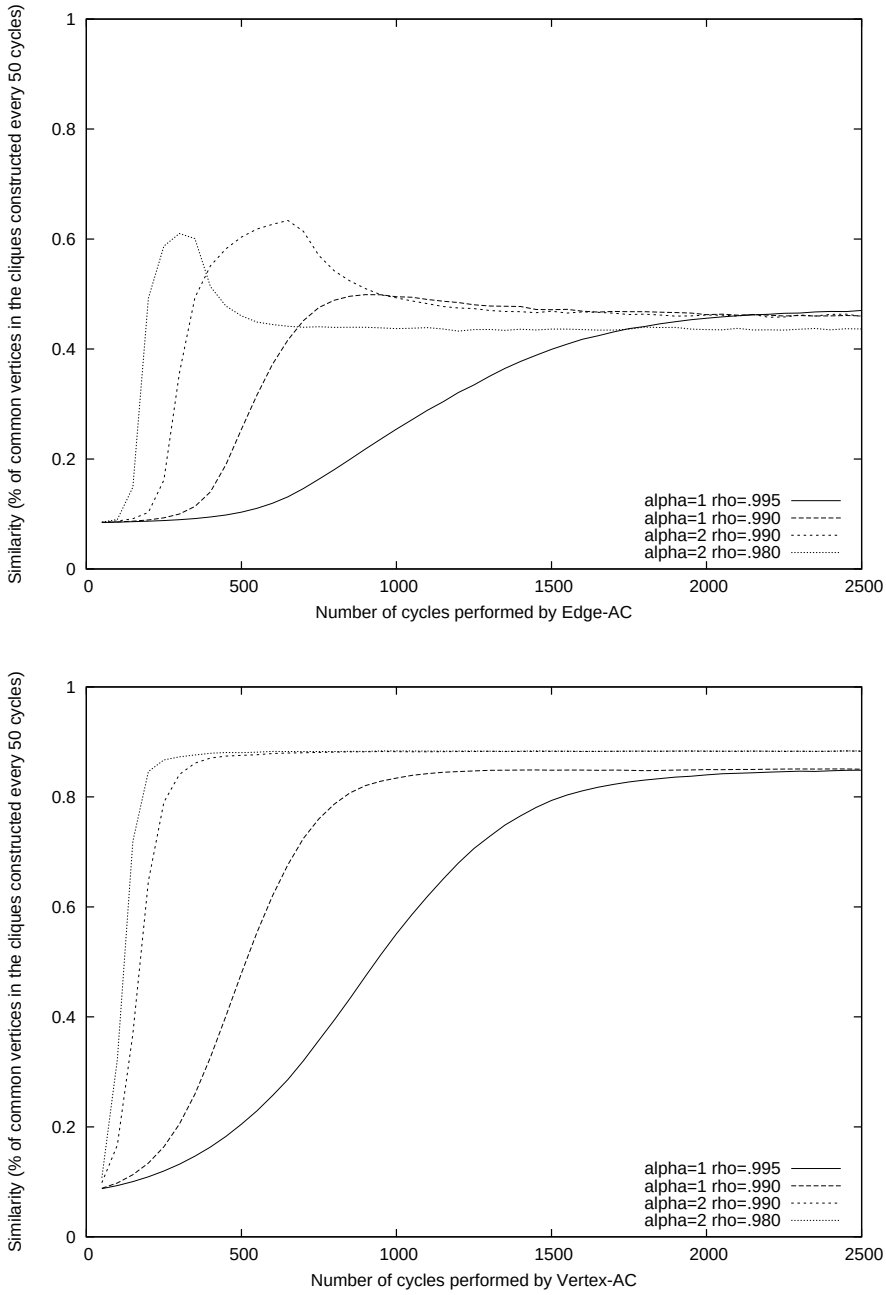

Figure 3: Evolution of the similarity ratio for `Edge-AC` (upper curves) and `Vertex-AC` (lower curves): each curve plots the similarity ratio of the set of cliques constructed every 50 cycles (average on 50 runs), for a given setting of α and ρ. The other parameters have been set to nbAnts=30, $\tau_{min} = 0.01$, $\tau_{max} = 6$.

during these 50 cycles. For example, let us consider the curve plotting the evolution of the similarity ratio for `Edge-AC` when α is set to 1 and ρ to 0.99. The similarity increases from less than 10% at the beginning of the solution process to 45% after a thousand or so cycles. This shows that ants progressively focus to a sub-region of the search space, so that two cliques constructed after cycle 1000 share nearly half of their vertices on average.

Figure 3 also shows that, when α increases or ρ decreases, the similarity ratio both increases sooner and rises more steeply. However, for each algorithm, the similarity ratio of all runs converges towards a same value, for all the considered settings of α and ρ: after two thousands or so cycles, the cliques computed by `Edge-AC` during every cycle share around 45% of their vertices whereas those computed by `Vertex-AC` share around 90% of their vertices.

The difference of diversification between the two considered algorithms may be explained by the choice made about their pheromonal components. Indeed, the considered `C500.9` instance has 500 vertices, so that in `Vertex-AC` ants may lay pheromone on 500 components, whereas in `Edge-AC` they may lay pheromone on $500 * 499/2$ components. Let us now consider the two cliques that are rewarded at the end of the first two cycles of a run. For both `Vertex-AC` and `Edge-AC`, these two cliques contain 44 vertices on average (see Fig. 2), and their similarity ratio is lower than 10% (see Fig. 3) so that they share 4 vertices on average. Under this hypothesis, after the first two cycles of `Vertex-AC`, 4 vertices —corresponding to 1% of the pheromonal components— have been rewarded twice, and 80 vertices —corresponding to 16% of the pheromone components— have been rewarded once. As a comparison, under the same hypothesis, after the first two cycles of `Edge-AC`, 6 edges —corresponding to less than 0.005% of the pheromonal components— have been rewarded twice, and 940 edges —corresponding to less than 0.8% of the pheromonal components— have been rewarded once. This explains why the search of `Edge-AC` is much more diversified than the one of `Vertex-AC`, and therefore why `Edge-AC` needs more time to converge, but as a counterpart, often finds better solutions than `Vertex-AC`.

4 Boosting ACO with local search

Basically, local search searches for a locally optimal solution in the neighborhood of a given constructed solution. Local search may be combined with the ACO meta-heuristic in a very straightforward way: ants construct solutions exploiting pheromone trails, and local search improves their quality by iteratively performing local moves. Actually, the best-performing ACO algorithms for many combinatorial optimization problems are hybrid algorithms that combine probabilistic solution construction by a colony of ants with local search [11, 33, 32].

In this section, we study the benefit of integrating local search within `Vertex-AC` and `Edge-AC`. The hybrid algorithm is derived from the algorithm of Figure 1 as follows: once each ant has constructed a clique, and before updating pheromone trails, we apply a local search procedure on the largest clique of the cycle until

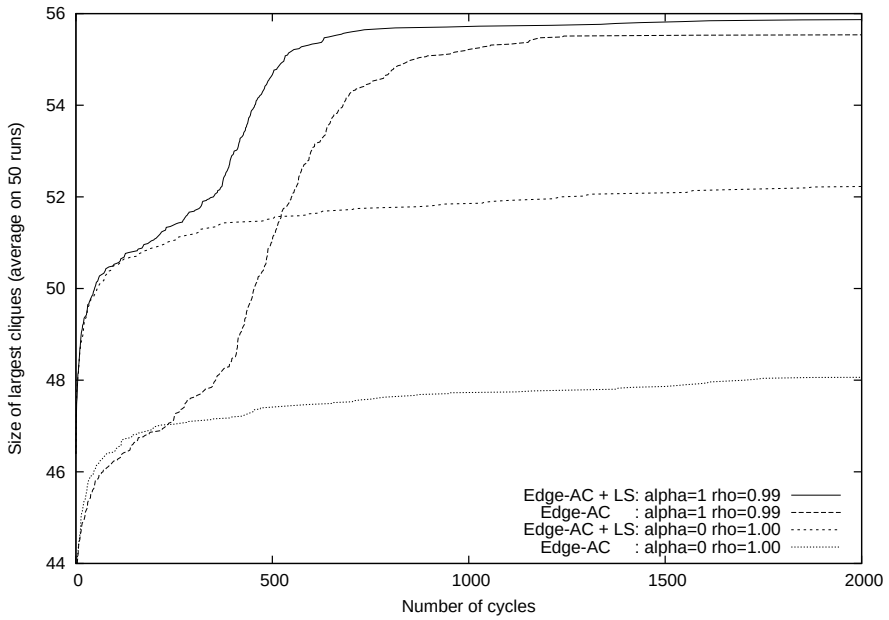

Figure 4: Evolution of the size of the best clique (average on 50 runs), with local search (`Edge-AC + LS`) or without local search (`Edge-AC`), and with pheromone (`alpha=1, rho=0.99`) or without pheromone (`alpha=0, rho=1.00`). The other parameters have been set to nbAnts=30, $\tau_{min} = 0.01$, and $\tau_{max} = 6$.

it becomes locally optimal[2]. Pheromone trails are then updated with respect to this locally optimal clique.

Various local search procedures may be used to improve cliques, e.g., [3, 4, 27]. However, as pointed out in [25], when choosing the local search procedure to use in a meta-heuristic such as evolutionary algorithms, iterated local search or ACO, one has to find a trade-off between computation time and solution quality. In other words, one has to choose between a fast but not-so-good local search procedure or a slower but more drastic one.

For all experiments reported in this paper, we have considered the (2,1)-exchange procedure used in `GRASP` [3]: given a clique $\mathcal{C}$, this local search procedure looks for three vertices v_i, v_j and v_k such that
- v_i belongs to $\mathcal{C}$,
- v_j and v_k do not belong to $\mathcal{C}$,
- v_j and v_k are linked by an edge, and
- v_j and v_k are adjacent to every vertex of $\mathcal{C} - \{v_i\}$;

then, it replaces the vertex v_i by the two vertices v_j and v_k, thus increasing the clique size by one. This local search procedure is iterated until it reaches a locally optimal state that cannot be improved by such a (2,1)-exchange.

Figure 4 shows that this local search procedure actually boosts the perfor-

[2]Local search could be applied to every computed clique (instead of applying it only to the best clique of the cycle). However, experiments showed us that this does not improve significantly solutions quality, whereas it is much more time consuming.

mances of Edge-AC when solving the C500.9 instance. In particular, when α is set to 0 and ρ to 1, i.e., when pheromone is not used, the local search procedure roughly increases the size of the constructed cliques by four, and this improvement in quality is constant during the whole run. When α is set to 1 and ρ to 0.99, so that pheromone actually influences the solution process, local search also improves the quality of the cliques constructed by Edge-AC, but the improvement in quality is not constant during the run: at the beginning of the run, local search increases cliques by four vertices; however, after a thousand or so cycles, the improvement in quality is much smaller; finally, at the end of the run, the best clique found when combining Edge-AC with local search contains 55.9 vertices on average, i.e., it improves the results of Edge-AC of 0.3.

Local search also boosts the performances of Vertex-AC, in a very similar way than for Edge-AC: for the C500.9 instance, it increases the average quality of the best clique after 2500 cycles from 55.2 to 55.4.

5 Experimental comparison of ACO algorithms

In this section, we experimentally evaluate and compare the four proposed ACO algorithms: Edge-AC, Vertex-AC, Edge-AC combined with local search (referred to as Edge-AC+LS) and Vertex-AC combined with local search (referred to as Vertex-AC+LS).

Experimental setup. All algorithms have been implemented in C and run on a 1.9 GHz Pentium 4 processor, under Linux operating system.

In all experiments, we have set α to 1 and ρ to 0.99, thus achieving a good compromise between solution quality and CPU-time as discussed in section 3.

The number of ants has been set to 30: with lower values, solution quality is decreased (as the best clique constructed at each cycle usually is significantly smaller); with greater values, the running time increases while solution quality is not significantly improved (as the best clique constructed at each cycle is not significantly better than with 30 ants).

The maximum number of cycles has been set to 5000. Indeed, for many instances all algorithms have converged to the best solutions within the first three thousands of cycles. However, on larger instances —that have more than one thousand of vertices— Edge-AC may need more cycles to converge so that we have fixed the maximum number of cycles to 5000 for all instances.

Finally, we have set the pheromone bounds τ_{min} to 0.01, and τ_{max} to 6.

Test suite. We consider 36 benchmark graphs provided by the DIMACS challenge on clique coloring and satisfiability[3]. C$n.p$ and DSJC$n.p$ graphs have n vertices and a density of $0.p$; MANN_a27 and MANN_a45 respectively have 378 and 1035 vertices, and a density of 0.99; brockn_m graphs have n vertices; genxxx_p0.9yy graphs have xxx vertices, a density of 0.9 and maximum cliques

[3]These graphs are available at http://dimacs.rutgers.edu/.

of size yy; `hamming8-4` and `hamming10-4` respectively have 256 and 1024 vertices; `keller` 4, 5 and 6 graphs respectively have 171, 776 and 3361 vertices; and `p_hat`n–m graphs have n vertices.

Comparison of solutions quality. Table 2 shows that `Edge-AC` outperforms `Vertex-AC` by means of solutions quality: when considering average results over the 50 runs, `Edge-AC` has found larger cliques than `Vertex-AC` for 21 instances, whereas it has found smaller cliques for 2 instances only. Moreover, `Edge-AC` has been able to find the best known solution for 29 instances, whereas `Vertex-AC` has found it for 24 instances.

This table also shows that local search improves the solution process of both algorithms for many instances: for `Vertex-AC`, the integration of local search improves average results for 22 instances, whereas it deteriorates them for 1 instance, and for `Edge-AC`, the integration of local search improves average results for 16 instances, whereas it deteriorates them for 2 instances.

`Edge-AC+LS` is the best performing of the four proposed algorithms, for a majority of instances, and it has been able to find the best known solution for 31 instances, over the 36 considered instances. However, for 2 instances (`brock400_2` and `keller6`), `Edge-AC` has found the best known solution whereas `Edge-AC+LS` did not.

Comparison of CPU time. Table 3 first shows that, for each considered algorithm, the number of cycles —and therefore the CPU time— needed to find the best solution mainly depends on the number of vertices and the connectivity of the graph. For example, `Edge-AC` respectively performs, on average, 473, 923, 2359, and 3278 cycles to solve the `C250.9`, `C500.9`, `C1000.9`, and `C2000.9` instances that respectively have 250, 500, 1000 and 2000 vertices and a connectivity close to 0.9, whereas it performs 1062 cycles to solve the `C2000.5` instances that have 2000 vertices and a connectivity of 0.5.

Table 3 also shows that `Vertex-AC` nearly always performs less cycles than `Edge-AC` and, on average for all instances, it needs 1.8 times less cycles to converge towards its best solution. When considering CPU-times, the difference becomes more important as `Edge-AC` needs more time to perform one cycle than `Vertex-AC`. Hence, `Vertex-AC` is from 1.5 to 26 times as fast as `Edge-AC`, and it is 5.7 times as fast on average for all instances.

Finally, one can remark that the integration of local search always decreases the number of cycles: on average, both `Vertex-AC+LS` and `Edge-AC+LS` perform 1.8 times less cycles than `Vertex-AC` and `Edge-AC` respectively. However, as local search is time consuming, CPU-times are rather comparable.

Table 2: Comparison of ACO algorithms by means of solutions quality. Each line successively displays the instance name, the size of its maximum clique, and the results obtained by `Vertex-AC`, `Edge-AC`, `Vertex-AC+LS`, and `Edge-AC+LS` (best and average solution found over 50 runs; standard deviation in brackets).

		Size of the best found clique							
		Vertex-AC		Edge-AC		Vertex-AC+LS		Edge-AC+LS	
Graph	$\omega(G)$	Max	Avg(Stdv)	Max	Avg(Stdv)	Max	Avg(Stdv)	Max	Avg(Stdv)
C125.9	34	34	34.0 (0.0)	34	34.0 (0.0)	34	34.0 (0.0)	34	34.0 (0.0)
C250.9	44	44	43.9 (0.3)	44	44.0 (0.0)	44	44.0 (0.1)	44	44.0 (0.0)
C500.9	$\geq$57	56	55.2 (0.8)	57	55.6 (0.8)	56	55.4 (0.8)	57	55.9 (0.6)
C1000.9	$\geq$68	67	65.3 (1.0)	67	66.0 (0.8)	67	65.7 (0.6)	68	66.2 (0.7)
C2000.9	$\geq$78	76	73.4 (1.1)	76	74.1 (1.3)	77	74.5 (1.1)	78	74.3 (1.4)
DSJC500.5	14	13	13.0 (0.0)	13	13.0 (0.0)	13	13.0 (0.0)	13	13.0 (0.0)
DSJC1000.5	15	15	14.1 (0.3)	15	14.1 (0.4)	15	14.2 (0.4)	15	14.3 (0.4)
C2000.5	$\geq$16	16	14.9 (0.4)	16	15.1 (0.3)	16	15.1 (0.3)	16	15.3 (0.5)
C4000.5	$\geq$18	17	15.9 (0.4)	16	15.8 (0.4)	17	16.2 (0.4)	18	16.8 (0.6)
MANN_a27	126	126	125.5 (0.5)	126	126.0 (0.2)	126	125.8 (0.4)	126	126.0 (0.0)
MANN_a45	345	344	342.8 (0.8)	344	343.3 (0.6)	344	342.6 (0.7)	344	342.9 (0.6)
brock200_2	12	12	11.9 (0.2)	12	12.0 (0.0)	12	11.9 (0.4)	12	12.0 (0.0)
brock200_4	17	17	16.1 (0.3)	17	16.8 (0.4)	17	16.1 (0.3)	17	16.8 (0.4)
brock400_2	29	25	24.5 (0.5)	29	25.0 (0.7)	25	24.7 (0.5)	25	24.8 (0.4)
brock400_4	33	25	24.0 (0.1)	33	25.1 (2.7)	25	24.2 (0.4)	33	27.1 (4.0)
brock800_2	24	21	20.0 (0.5)	21	19.8 (0.5)	21	20.4 (0.5)	24	20.1 (0.6)
brock800_4	26	21	19.8 (0.6)	26	19.9 (1.0)	21	20.2 (0.4)	26	20.0 (0.8)
gen200_p0.9_44	44	44	41.4 (1.9)	44	43.7 (1.1)	44	43.3 (1.5)	44	44.0 (0.0)
gen200_p0.9_55	55	55	55.0 (0.0)	55	55.0 (0.0)	55	55.0 (0.0)	55	55.0 (0.0)
gen400_p0.9_55	55	52	51.2 (0.5)	53	51.9 (0.5)	52	51.3 (0.5)	53	52.2 (0.4)
gen400_p0.9_65	65	65	65.0 (0.0)	65	65.0 (0.0)	65	65.0 (0.0)	65	65.0 (0.0)
gen400_p0.9_75	75	75	75.0 (0.0)	75	75.0 (0.0)	75	75.0 (0.0)	75	75.0 (0.0)
hamming8_4	16	16	16.0 (0.0)	16	16.0 (0.0)	16	16.0 (0.0)	16	16.0 (0.0)
hamming10_4	40	40	38.0 (1.5)	40	38.6 (1.2)	39	38.7 (0.6)	40	39.3 (0.9)
keller4	11	11	11.0 (0.0)	11	11.0 (0.0)	11	11.0 (0.0)	11	11.0 (0.0)
keller5	27	27	26.7 (0.5)	27	26.9 (0.2)	27	26.9 (0.3)	27	27.0 (0.0)
keller6	$\geq$59	55	50.8 (1.9)	59	53.1 (1.7)	55	51.5 (1.5)	57	55.1 (1.3)
p_hat300_1	8	8	8.0 (0.0)	8	8.0 (0.0)	8	8.0 (0.0)	8	8.0 (0.0)
p_hat300_2	25	25	25.0 (0.0)	25	25.0 (0.0)	25	25.0 (0.0)	25	25.0 (0.0)
p_hat300_3	36	36	35.9 (0.5)	36	36.0 (0.1)	36	36.0 (0.3)	36	36.0 (0.0)
p_hat700_1	11	11	10.8 (0.4)	11	11.0 (0.1)	11	10.9 (0.3)	11	11.0 (0.1)
p_hat700_2	44	44	44.0 (0.0)	44	44.0 (0.0)	44	44.0 (0.0)	44	44.0 (0.0)
p_hat700_3	$\geq$62	62	62.0 (0.0)	62	62.0 (0.0)	62	62.0 (0.0)	62	62.0 (0.0)
p_hat1500_1	12	12	11.0 (0.2)	12	11.1 (0.3)	12	11.2 (0.4)	12	11.1 (0.2)
p_hat1500_2	$\geq$65	65	64.9 (0.2)	65	64.9 (0.2)	65	65.0 (0.0)	65	65.0 (0.0)
p_hat1500_3	$\geq$94	94	93.1 (0.2)	94	93.9 (0.4)	94	93.6 (0.5)	94	94.0 (0.0)

Table 3: Comparison of ACO algorithms by means of CPU-time. Each line successively displays the number of cycles (average on 50 runs) and the CPU-time (average and standard deviation on 50 runs) for `Vertex-AC`, `Edge-AC`, `Vertex-AC+LS`, and `Edge-AC+LS`.

	Number of cycles and time to find the best clique							
	Vertex-AC		Edge-AC		Vertex-AC+LS		Edge-AC+LS	
Graph	Cycles	Time(Stdv)	Cycles	Time(Stdv)	Cycles	Time(Stdv)	Cycles	Time(Stdv)
C125.9	60	0.1 (0.0)	126	0.2 (0.1)	14	0.0 (0.0)	23	0.0 (0.0)
C250.9	359	0.8 (0.7)	473	1.7 (0.3)	172	0.5 (0.1)	239	1.0 (0.3)
C500.9	722	3.8 (1.9)	923	8.9 (4.0)	477	4.6 (2.7)	671	8.6 (4.7)
C1000.9	1219	13.2 (5.8)	2359	55.0 (21.9)	832	23.4 (8.5)	1242	49.8 (26.6)
C2000.9	1770	41.3 (11.0)	3278	214.4 (49.8)	1427	112.4 (16.8)	2067	238.7 (98.3)
DSJC500.5	588	0.5 (0.2)	832	2.6 (1.8)	249	0.7 (0.5)	285	1.4 (1.3)
DSJC1000.5	820	1.4 (1.0)	1017	9.6 (6.4)	561	3.8 (4.0)	567	7.8 (8.8)
C2000.5	957	3.3 (3.0)	1062	30.4 (16.8)	312	5.9 (5.7)	957	40.6 (56.0)
C4000.5	927	6.0 (5.1)	1116	108.3 (73.8)	445	22.4 (33.1)	1915	257.6 (190.2)
MANN_a27	616	11.5 (3.5)	2274	61.7 (25.3)	574	11.1 (4.9)	1824	44.8 (19.3)
MANN_a45	1771	271.3 (77.0)	4360	877.4 (69.1)	1498	239.2 (56.9)	3639	749.4 (132.5)
brock200_2	115	0.0 (0.0)	127	0.1 (0.1)	104	0.1 (0.1)	115	0.1 (0.1)
brock200_4	156	0.1 (0.0)	1627	1.3 (1.2)	69	0.1 (0.0)	1356	1.7 (1.9)
brock400_2	650	1.0 (0.6)	1004	3.5 (1.9)	424	1.4 (0.6)	720	3.8 (3.5)
brock400_4	498	0.8 (0.2)	977	3.4 (3.3)	254	0.8 (0.4)	1141	5.7 (6.3)
brock800_2	1145	2.6 (1.1)	1238	9.1 (6.1)	881	6.3 (3.6)	959	11.6 (10.5)
brock800_4	1173	2.7 (1.0)	1288	9.8 (7.1)	858	6.1 (2.5)	906	11.1 (10.7)
gen200_p0.9_44	297	0.6 (0.2)	338	0.9 (0.2)	136	0.3 (0.1)	165	0.5 (0.1)
gen200_p0.9_55	102	0.2 (0.0)	130	0.3 (0.1)	57	0.2 (0.0)	100	0.3 (0.1)
gen400_p0.9_55	475	2.0 (1.2)	1634	11.5 (6.4)	270	1.8 (0.6)	760	6.7 (3.7)
gen400_p0.9_65	337	1.4 (0.2)	391	2.7 (0.3)	206	1.5 (0.1)	255	2.3 (0.3)
gen400_p0.9_75	251	1.0 (0.1)	293	2.1 (0.2)	153	1.2 (0.1)	198	2.1 (0.2)
hamming8_4	34	0.0 (0.0)	42	0.1 (0.0)	49	0.1 (0.1)	42	0.1 (0.1)
hamming10_4	2308	13.7 (1.9)	1474	28.0 (10.5)	1204	26.2 (17.3)	865	29.3 (16.3)
keller4	2	0.0 (0.0)	2	0.0 (0.0)	0	0.0 (0.0)	0	0.0 (0.0)
keller5	1652	4.9 (1.2)	1136	10.8 (7.9)	979	9.4 (6.0)	830	12.3 (9.7)
keller6	2514	55.3 (15.2)	3034	308.8 (111.8)	1657	206.8 (145.3)	2617	549.2 (250.6)
p_hat300_1	40	0.0 (0.0)	46	0.0 (0.0)	18	0.0 (0.0)	20	0.0 (0.0)
p_hat300_2	113	0.1 (0.0)	206	0.3 (0.1)	26	0.1 (0.0)	54	0.2 (0.1)
p_hat300_3	281	0.5 (0.5)	457	1.3 (0.3)	110	0.3 (0.1)	176	0.7 (0.2)
p_hat700_1	379	0.2 (0.1)	624	2.6 (1.6)	253	0.7 (0.3)	391	2.4 (2.0)
p_hat700_2	297	0.7 (0.1)	445	3.2 (0.6)	128	2.0 (0.6)	227	4.5 (1.0)
p_hat700_3	575	3.0 (2.2)	878	9.8 (3.2)	220	3.8 (1.0)	333	7.8 (1.4)
p_hat1500_1	391	0.4 (0.3)	662	10.5 (8.5)	318	3.0 (4.4)	438	9.7 (17.0)
p_hat1500_2	499	3.3 (1.0)	801	20.1 (4.9)	238	17.3 (2.7)	379	35.2 (3.9)
p_hat1500_3	581	8.6 (6.5)	2199	74.3 (41.6)	551	37.0 (34.4)	528	54.9 (8.8)

6 Experimental comparison with other heuristic approaches

We now compare `Edge-AC+LS`, which is our best performing ACO algorithm on a majority of instances, with three recent and representative algorithms, i.e., `RLS`, `DAGS`, and `GLS`.

RLS (Reactive Local Search) [4] is based on a tabu local search heuristic. Starting from an empty clique, `RLS` iteratively moves in the search space composed of all cliques by adding/removing one vertex to/from the current clique. A tabu list is used to memorize the T last moves and, at each step, `RLS` greedily selects a move that is not prohibited by the tabu list. The key point is that the length T of the tabu list is dynamically updated with respect to the need for diversification. `RLS` appears to be the best heuristic algorithm for the maximum clique problem we are aware of for a majority of DIMACS benchmark instances.

Table 4 displays results reported in [4], where CPU times have been multiplied by 0.027, corresponding to the ratio of speed of computers with respect to the Spec benchmark on floating point units.

DAGS [19] is a two-phase procedure: in a first phase, a greedy procedure combined with "swap" local moves is applied, starting from each node of the graph; in a second phase, nodes are scored with respect to the number of times they have been selected during the first phase, and an adaptive greedy algorithm is repeatedly started to build cliques around the nodes with the least scores, in order to diversify the search towards less explored areas.

Experiments reported in [19] show that the first phase is able to find best known solutions for many instances. For harder instances, the second phase improves solutions quality in many cases. This improvement in quality is dramatic on the set of `brock` instances, that are known to be very difficult for greedy approaches. For these instances, `DAGS` outperforms `RLS` performances.

Table 4 displays results reported in [19]. Note that the second phase of `DAGS` has been performed only for the harder instances, that have not been solved during the first phase. Hence, we specify in table 4 if these results have been obtained after the first or the second phase. We multiplied CPU times by 0.84, corresponding to the ratio of speed of computers with respect to the Spec benchmark on floating point units.

GLS [27] combines a genetic algorithm with local search, and we consider it for comparison, though it does not outperforms the performances of `RLS` and `DAGS`, because it presents some similarities with ACO: both approaches use a bio-inspired metaphor to intensify the search towards the most "promising" areas with respect to previously computed solutions. `GLS` generates successive populations of maximal cliques from an initial one by repeatedly selecting two parent cliques from the current population, recombining them to generate two children cliques, applying local search on children to obtain maximal cliques,

and adding to the new population the best two cliques of parents and children. GLS can be instantiated to different algorithms by modifying its parameters. In particular, [27] compares results obtained by the three following instances of GLS: GENE performs genetic local search; ITER performs iterated local search, starting from one random point; and MULT performs multi-start local search, starting from a new random point at each time.

Table 4 displays, for each considered instance, the results obtained by the GLS algorithm (over GENE, ITER, and MULT) that obtained the best average results. We multiplied CPU times by 0.037, corresponding to the ratio of speed of computers with respect to the Spec benchmark on floating point units.

By means of solutions quality, the results of table 4 can be summarized in the following table, giving for each heuristic approach, the number of times Edge-AC+LS has found larger (+), equal (=), and smaller (−) cliques.

	Comparison of Edge-AC+LS with								
	RLS			DAGS			GLS		
	+	=	−	+	=	−	+	=	−
Best clique	2	30	4	2	29	1	11	23	2
Average size	0	20	16	6	15	11	28	7	1

Hence, when considering the best found results, Edge-AC+LS is competitive with both RLS and DAGS, being able to find better solutions than RLS on two brock instances, and better solutions than DAGS on two C*.9 instances. However, when considering average results RLS outperforms Edge-AC+LS on 16 instances. The comparison with DAGS on average results depends on the considered instances. In particular, DAGS outperforms Edge-AC+LS on brock instances, whereas Edge-AC+LS outperforms DAGS on gen instances.

This table also shows that Edge-AC+LS outperforms GLS both with respect to best and average results. Hence, for this problem ACO is better suited than evolutionary computation for guiding the search towards promising areas.

When considering CPU-times reported in table 4, one can note that Edge-AC+LS is an order slower than RLS and GLS, whereas it is rather comparable with DAGS: Edge-AC+LS is quicker than DAGS for 24 instances, and it is slower for 8 instances.

7 Conclusion

We have described two basic ACO algorithms for searching for maximum cliques. These two algorithms differ in the choice of their pheromonal components. A main motivation was to answer the following question: should we lay pheromone on the vertices or on the edges of the graph?

Experiments have shown that both algorithms are able to find optimal solutions on many benchmark instances, showing that ACO is actually able to guide the search towards promising areas. However, when comparing the diversification capability of the two algorithms, by means of re-sampling and similarity ratio, we have noticed that the search is more diversified when pheromone is laid

Table 4: Comparison of `Edge-AC+LS` with other heuristic approaches. Each line successively displays the results obtained by `Edge-AC+LS`, `RLS`, `DAGS` (followed by (1) or (2) if they have been obtained after the first or the second phase), and `GLS` (followed by (G), (I), or (M) if they have been obtained by `GENE`, `ITER`, or `MULT`). For each approach, the table reports the best and average solution found (over 50 runs for `Edge-AC+LS`, 100 runs for `RLS`, 20 runs `DAGS`, and 10 runs for `GLS`), and the estimated CPU-time w.r.t. the Spec floating points benchmark.

	Edge-AC+LS			RLS			DAGS				GLS			
Graph	Max	Avg	Time	Max	Avg	Time	Max	Avg	Time		Max	Avg	Time	
C125.9	34	34.0	0.0	34	34.0	0.0	34	34.0	0.2	(1)	34	34.0	0.0	(I)
C250.9	44	44.0	1.0	44	44.0	0.0	44	44.0	0.7	(1)	44	43.0	0.1	(I)
C500.9	57	55.9	8.6	57	57.0	0.0	56	55.8	13.5	(2)	55	52.7	0.1	(I)
C1000.9	68	66.2	49.8	68	68.0	1.1	68	65.9	148.2	(2)	66	61.6	0.5	(G)
C2000.9	78	74.3	238.7	78	77.6	22.1	76	75.4	1824.0	(2)	70	68.7	0.9	(I)
DSJC500.5	13	13.0	1.4	13	13.0	0.0	-	-	-	-	13	12.2	0.1	(G)
DSJC1000.5	15	14.3	7.8	15	15.0	0.2	-	-	-	-	14	13.5	0.1	(I)
C2000.5	16	15.3	40.6	16	16.0	0.3	16	15.9	24.8	(1)	15	14.2	0.1	(I)
C4000.5	18	16.8	257.6	18	18.0	58.7	18	17.5	3229.0	(2)	16	15.6	0.6	(I)
MANN_a27	126	126.0	44.8	126	126.0	0.1	126	126.0	6.1	(1)	126	126.0	0.6	(I)
MANN_a45	344	342.9	749.4	345	343.6	10.7	344	343.9	1921.0	(2)	345	343.1	2.0	(I)
brock200_2	12	12.0	0.1	12	12.0	0.3	12	12.0	0.1	(2)	12	12.0	0.1	(M)
brock200_4	17	16.8	1.7	17	17.0	0.5	17	16.8	0.3	(2)	17	15.7	0.1	(M)
brock400_2	25	24.8	3.8	29	26.0	1.1	29	28.1	2.8	(2)	25	23.2	0.1	(I)
brock400_4	33	27.1	5.7	33	32.4	2.9	33	33.0	2.8	(2)	25	23.6	0.0	(G)
brock800_2	24	20.1	11.6	21	21.0	0.1	24	20.8	16.8	(2)	20	19.3	0.2	(G)
brock800_4	26	20.0	11.1	21	21.0	0.2	26	22.6	16.9	(2)	20	19.0	0.1	(I)
gen200_p0.9_44	44	44.0	0.5	44	44.0	0.0	44	41.1	0.9	(2)	44	39.7	0.1	(G)
gen200_p0.9_55	55	55.0	0.3	55	55.0	0.0	55	55.0	0.4	(1)	55	50.8	0.1	(G)
gen400_p0.9_55	53	52.2	6.7	55	55.0	0.0	53	51.8	7.2	(2)	55	49.7	0.1	(G)
gen400_p0.9_65	65	65.0	2.3	65	65.0	0.0	65	55.4	7.3	(2)	65	53.7	0.2	(G)
gen400_p0.9_75	75	75.0	2.1	75	75.0	0.0	75	55.2	7.8	(2)	75	62.7	0.2	(I)
hamming8_4	16	16.0	0.1	16	16.0	0.0	-	-	0.0	(1)	16	16.0	0.0	(G)
hamming10_4	40	39.3	29.3	40	40.0	0.0	40	40.0	12.8	(1)	40	38.2	0.2	(I)
keller4	11	11.0	0.0	11	11.0	0.0	-	-	0.0	(1)	11	11.0	0.0	(G)
keller5	27	27.0	12.3	27	27.0	0.0	27	27.0	4.3	(1)	27	26.3	0.2	(I)
keller6	57	55.1	549.2	59	59.0	5.1	57	56.4	12326.0	(2)	56	52.7	1.3	(I)
p_hat300_1	8	8.0	0.0	8	8.0	0.0	8	8.0	0.1	(1)	8	8	0.0	(G)
p_hat300_2	25	25.0	0.2	25	25.0	0.0	25	25.0	0.5	(1)	25	25.0	0.0	(I)
p_hat300_3	36	36.0	0.7	36	36.0	0.0	36	36.0	0.8	(1)	36	35.1	0.1	(I)
p_hat700_1	11	11.0	2.4	11	11.0	0.0	11	11.0	0.7	(1)	11	9.9	0.1	(I)
p_hat700_2	44	44.0	4.5	44	44.0	0.0	44	44.0	5.5	(1)	44	43.6	0.0	(I)
p_hat700_3	62	62.0	7.8	62	62.0	0.0	62	62.0	8.5	(1)	62	61.8	0.2	(I)
p_hat1500_1	12	11.1	9.7	12	12.0	0.8	12	11.7	31.1	(2)	11	10.8	0.5	(G)
p_hat1500_2	65	65.0	35.2	65	65.0	0.0	65	65.0	47.7	(1)	65	63.9	0.5	(I)
p_hat1500_3	94	94.0	54.9	94	94.0	0.0	94	94.0	82.2	(1)	94	93.0	0.3	(I)

on edges so that better solutions are found on a wide majority of benchmark instances, but as a counterpart, more time is needed to converge.

Experiments also shown that the integration of local search techniques improves the solution process, and makes ACO competitive with state-of-the-art heuristic approaches, though it is more time consuming.

We believe that this comparison of two ACO models for the maximum clique problem could be useful to solve other similar problems. Indeed, for many combinatorial optimization problems such as multi-knapsacks problems or generalized assignment problems, the goal is to find, given an initial set of objects, the best subset with respect to some objective function. To solve this kind of problems with ACO, one has to choose between laying pheromone on the objects to choose, or on edges linking the objects to choose. In the first case, one will increase the desirability of choosing each rewarded object independently from the others, whereas in the second case, one will increase the desirability of choosing together two objects. Hence, further work will concern a generalization of this comparative study.

References

[1] M. Dorigo A. Colorni and V. Maniezzo. An investigation of some properties of an ant algorithm. In *Proceedings of PPSN'92, Elsevier*, pages 509–520, 1992.

[2] E.H.L. Aarts and J.H.M. Korst. In *Simulated annealing and the Boltzmann machines*. John Wiley & Sons, Chichester, U.K., 1989.

[3] J. Abello, P.M. Pardalos, and M.G.C. Resende. On maximum clique problems in very large graphs. In *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, volume 50, pages 119–130. American Mathematical Society, 2000.

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

[5] I. Bomze, M. Budinich, P. Pardalos, and M. Pelillo. The maximum clique problem. In D.-Z. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 4. Kluwer Academic Publishers, Boston, MA, 1999.

[6] 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.

[7] E. Burke, S. Gustafson, and G. Kendall. A survey and analysis of diversity measures in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 02)*, pages 716–723. Morgan Kaufmann Publishers, 2002.

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

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

[10] 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.

[11] 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.

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

[13] 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.

[14] T.A. Feo, M.G.C. Resende, and S.H. Smith. A greedy randomized adaptive search procedure for maximum independent set. *Operations Research*, 42:860–878, 1994.

[15] C. Friden, A. Hertz, and D. de Werra. Stabulus: a technique for finding stable sets in large graphs with tabu search. *Computing*, 42:35–44, 1989.

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

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

[18] F. Glover and M. Laguna. Tabu search. In *Modern Heuristics Technics for Combinatorial Problems*, pages 70–141. Blackwell Scientific Publishing, Oxford, UK, 1993.

[19] 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.

[20] S. Homer and M. Peinado. On the performance of polynomial-time clique approximation algorithms on very large graphs. In *Cliques, Coloring, and Satisfiability: second DIMACS Implementation Challenge*, volume 26, pages 103–124. DIMACS American Mathematical Society, 1996.

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

[22] D.S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer Science*, 9:256–278, 1974.

[23] 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.

[24] R.M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1973.

[25] H.R. Lourenço, O. Martin, and T. Stützle. Iterated local search. In F. Glover and G. Kochenberger, editors, *Handbook of metaheuristics*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2002, in press.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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. Springer-Verlag, 2001.

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

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

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

[34] J. 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.

[35] J.I. van Hemert and T. Bäck. Measuring the searched space to guide efficiency: The principle and evidence on constraint satisfaction. In J.J. Merelo, A. Panagiotis, H.-G. Beyer, José-Luis Fernández-Villacañas, and Hans-Paul Schwefel, editors, *Proceedings of the 7th International Conference on Parallel Problem Solving from Nature*, number 2439 in LNCS, pages 23–32, Berlin, 2002. Springer-Verlag.