



Inteligencia Artificial. Revista
Iberoamericana de Inteligencia Artificial

ISSN: 1137-3601

revista@aepia.org

Asociación Española para la Inteligencia
Artificial
España

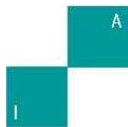
Negri Lintzmayer, Carla; Henrique Mulati, Mauro; da Silva, Anderson Faustino
The Hybrid ColorAnt-RT Algorithms and an Application to Register Allocation
Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial, vol. 18, núm. 55,
2015, pp. 81-111
Asociación Española para la Inteligencia Artificial
Valencia, España

Disponible en: <http://www.redalyc.org/articulo.oa?id=92538718007>

- Cómo citar el artículo
- Número completo
- Más información del artículo
- Página de la revista en redalyc.org

redalyc.org

Sistema de Información Científica
Red de Revistas Científicas de América Latina, el Caribe, España y Portugal
Proyecto académico sin fines de lucro, desarrollado bajo la iniciativa de acceso abierto



The Hybrid *ColorAnt-RT* Algorithms and an Application to Register Allocation

Carla Negri Lintzmayer

Institute of Computing, University of Campinas, Campinas/SP, Brazil
carlanl@ic.unicamp.br

Mauro Henrique Mulati

Department of Computer Science, Midwestern State University, Guarapuava/PR, Brazil
mhmulati@unicentro.br

Anderson Faustino da Silva

Departament of Informatic, State University of Maringá, Maringá/PR, Brazil
anderson@din.uem.br

Abstract Ant Colony Optimization is a metaheuristic used to create heuristic algorithms to find good solutions for combinatorial optimization problems. This metaheuristic is inspired on the behavior present in ants. This specie explores the environment to find and transport food to the nest. Several works have proposed the use of Ant Colony Optimization algorithms to solve problems such as vehicle routing, frequency assignment, scheduling and graph coloring. The graph coloring problem essentially consists in finding a number k of colors to assign to its vertices, so that there are no two adjacent vertices with the same color. This paper presents the hybrid *ColorAnt-RT* algorithms, a class of algorithms for graph coloring problems, which is based on the Ant Colony Optimization metaheuristic and uses Tabu Search as local search. The experiments with *ColorAnt-RT* algorithms indicate that changing the way to reinforce the pheromone trail results in better results. The results with *ColorAnt-RT* show that it is a promising option in finding good approximations of k . The good results obtained by *ColorAnt-RT* motivated its use on a register allocation based on Ant Colony Optimization, called **CARTRA**. As a result, this paper also presents **CARTRA**, an algorithm that extends a classic graph coloring register allocator to use the graph coloring algorithm *ColorAnt-RT*. **CARTRA** minimizes the number of spills, thereby improving the quality of the generated code.

Keywords: Graph Coloring Problem, Ant Colony Optimization, *ColorAnt-RT*, Register Allocation, **CARTRA**

1 Introduction

The Graph Coloring Problem (GCP) consists in finding the minimum number of k colors to assign to the vertices of a graph so that there are no conflicting vertices (adjacent vertices assigned with the same color). It is a \mathcal{NP} -hard combinatorial optimization problem [46]. The GCP shows up in several problems in which it is necessary to partition a set of elements in groups of members with certain features in common, for example, register allocation in compilers [67], scheduling [29], timetabling [58] and communication networks [6, 53].

\mathcal{NP} -hard problems demand exact algorithms in superpolynomial time to obtain an optimal solution, unless $\mathcal{P} = \mathcal{NP}$ [20]. An alternative to find good solutions in an acceptable time, for that class of problems is using heuristic algorithms, which can be based on metaheuristics. Metaheuristic is defined as a set of algorithmic and data structure concepts to the development and application of heuristic algorithms.

The research field of *swarm intelligence* is inspired on the social behavior of swarms: individuals that cooperate and organize themselves without a central control [24]. Examples of these individuals are ants [43], bees [8] and

termites. The algorithms presented in this paper utilize the metaheuristic Ant Colony Optimization (ACO), which is based on the behavior presented by some ants during the search for food in an environment [27]. Among ACO algorithms, *Ant System* [25] was the first one, applied originally to the Traveling Salesman Problem (TSP) [3]. Other ACO algorithms were created, such as *Max-Min Ant System* [69] and *Ant Colony System* [23], obtaining good results for some kinds of problems. Several researches have proposed using ACO algorithms to solve other problems besides TSP such as vehicle routing [17], frequency assignment [51], multiple knapsack [62], constraint satisfaction [72], machine learning [16] and the previously referred graph coloring [27].

This paper presents an investigation of three versions of the hybrid *ColorAnt-RT* algorithms, which are based on ACO combined with local search for the GCP. First, we implemented an algorithm that was able to obtain satisfactory solutions, called *ColorAnt₁-RT*. The investigation with *ColorAnt₁-RT* indicated that changing the way to reinforce the pheromone trail results in a reduction in the number of conflicts, leading us to develop *ColorAnt₂-RT*, and finally *ColorAnt₃-RT*. The three *ColorAnt* algorithms use React-Tabucol (RT) [10] as local search in order to improve the results. Several other papers exploit the application of heuristic algorithms to the GCP [30, 31, 45, 48, 60].

The experiments with *ColorAnt-RT* algorithms were performed in forty-nine graphs, of the well known DIMACS challenge [45]. The results indicate that *ColorAnt₃-RT* is indeed the best among the three algorithms, and it is a good option on obtaining good solutions, besides minimizing the amount of conflicts.

Register allocation, a problem that can be mapped as a GCP, determines which of the program values (variables and temporaries) should be on machine registers or memory during the execution of the program [2, 28, 57]. In a real machine, registers are usually few and fast to access [59, 68], so the problem addressed here is how to minimize the traffic between registers and memory. Therefore, the challenge is to relegate the minor amount of program values to memory.

The mapping of register allocation as a GCP [15, 34] is done in a way that the vertices represent the values of a program, the edges are related to the interference of these and the colors represent the machine registers. A conflicting vertices means that the values represented by them cannot be allocated to the same register, at the same time. Note that, in register allocation we have to consider a slight situation: it is forced to eliminate conflicting vertices before coloring the graph with at most k colors, what is done by spilling some values to be represented in the memory. In this way, a *ColorAnt-RT* algorithm can be applied in the resolution of the register allocator problem.

We also present an intraprocedural register allocation algorithm called *ColorAnt₃-RT Register Allocator* (CARTRA), which is based on the *ColorAnt₃-RT* algorithm. CARTRA extends the Iterated Register Coalescing Allocator (IRA) [2] to use the ACO-based *ColorAnt₃-RT* algorithm. The results with CARTRA have indicated that CARTRA outperforms IRA in terms of program values that are effectively represented in memory, besides in code size. Moreover, the results have indicated that CARTRA is useful in situations where compile time is not important, but code quality, such as a compiler that generates code to embedded systems [54, 74].

The remaining of this paper is organized as follows: Section 2 presents concepts and definitions of graph coloring problem, ACO and register allocation; Section 3 presents some related works founded on literature and describes the *ColorAnt-RT* algorithms as well as their results; Section 4 describes a real application of *ColorAnt₃-RT* algorithm on the register allocation problem and its results; and concluding remarks are discussed in Section 5.

2 Definitions

The ACO metaheuristic, the GCP, and the register allocation problem are the starting point of this study. This section aims to present the definitions related to the entire work.

2.1 The Graph Coloring Problem

A k -coloring of a graph $G = (V, E)$ is the assignment of k colors to its vertices. A coloring is called *proper* if there is no *conflicting vertices*. A graph is k -colorable if it has a proper k -coloring. The minimum value of k for which a graph G is k -colorable is called *chromatic number* of G and it is denoted $\chi(G)$. The graph G is considered k -chromatic if $k = \chi(G)$ [11].

The GCP, which is an optimization problem, consists in finding the minimum value of k for which a graph G is k -colorable, i.e., it searches for the $\chi(G)$. Given a graph G and an integer k , the GCP decision problem could be formulated as: is the graph G k -colorable?

The solutions to GCP can be treated according to two approaches. The first one is a mapping of k colors to its vertices, that is, a mapping

$$s : V \rightarrow \{1, \dots, k\} \quad \forall (v_i, v_j) \in E : s(v_i) \neq s(v_j).$$

Another approach is the partitioning of V in k independent sets or legal classes (classes of colors)

$$s = \{C_1, \dots, C_k\} \quad \forall i, j, i \neq j: C_i \cap C_j = \{\}.$$

k -GCP consists in minimizing the number of conflicting vertices, given that the k -GCP colors a graph with a fixed number of k colors, independently whether or not there are conflicts. If the k -GCP finds zero as the number of conflicts, it found a solution to the GCP related to a decision problem. An algorithm for the k -GCP can be used as an GCP algorithm: it must start coloring the graph with an upper bound value for k , e.g. $|V|$, and after trying to find a proper k -coloring with low values for k . This approach is done in some heuristic algorithms.

Some graphs with specific features have a known and fixed value of χ , such as bipartite graphs (2-colorable) and planar graphs (they are at most 4-colorable). In order to determine if a graph is 2-colorable, there are algorithms in polynomial time [11]. For other non-special cases, satisfactory techniques are necessary since there is not an exact algorithm that is able to find optimal solutions for arbitrary instances, unless $\mathcal{P} = \mathcal{NP}$ [20].

An example of real application for k -GCP is register allocation [2]. For this problem, there must be used a heuristic to “eliminate” conflicting vertices as best as possible, since it is required to color the graph with just k colors (registers).

2.2 Ant Colony Optimization

Colonies of real ants are well organized and present behavior that allow them to perform some tasks that would not be possible for a single ant to do. The indirect communication that coordinates and guides them is something possible, due to modifications that ants cause in the environment, in a process called “*stigmergy*” [27]. In most cases, this communication is done by depositing a chemical substance called *pheromone* on the ground, forming trails that guide the paths of the ants.

The pheromone concentration in a path indicates the probability of an ant to choose it. A behavior like this is called *autocatalytic*: a process that reinforces itself causing convergence [26]. The pheromone is a substance that evaporates with time, besides shortest paths are traversed more quickly, encouraging ants to pass through them more often. Therefore, at some point the trend is that the colony is traversing the shortest possible path between two points.

This feature caught the attention of researchers to the fact that the behavior of ants could be mapped and utilized computationally. So that, it was well exploited and firstly applied to the Traveling Salesman Problem (TSP) [3, 25]. Since then, studies have been conducted to map the behavior of ants to many others optimization problems, like GCP [16, 17, 24, 27, 51, 62, 72].

ACO is a metaheuristic of combinatorial optimization, which is based on the behavior of real ants. A metaheuristic is “a set of algorithmic concepts that can be used to define heuristic methods applicable to a wide set of different problems” [27]. Other metaheuristics are *Tabu Search* [33], *Simulated Annealing* [47], *Iterated Local Search* [50], and *Genetic Algorithms* [35].

The execution of an ACO algorithm (see Algorithm 1) is composed of cycles. Each ant usually is a constructive method and its behavior can be noted when, in order to decide to where the ant must go next it is used a probability that is calculated based on two factors: pheromone trail and heuristic information [24]. The heuristic information depends on each problem. Once the solutions are constructed by the ants (and eventually improved by a local search method), they are used to update the pheromone trails.

Algorithm 1 ACO metaheuristic.

ACO-METAHEURISTIC

```

1  Initialize parameters, initialize pheromone trails;
2  while stop conditions not found do
3      CONSTRUCT-ANTS-SOLUTIONS();
4      APPLY-LOCAL-SEARCH();           // optional
5      UPDATE-PHEROMONE-TRAILS();
```

Different kinds of methods using colonies of artificial ants were developed for GCP and k -GCP. They are classified in three classes [40]:

- Class 1 is composed by algorithms in which each ant is a constructive method, that reinforces the pheromone trail between pairs of non-adjacent vertices when they assign the same color;
- Class 2 is composed by algorithms in which the ants walk through the graph (not always colored previously) and try to reduce the number of conflicting vertices by modifying the colors of the vertices;

- Class 3 is composed by algorithms in which the ants are local search methods, looking for neighborhood solutions (solutions that are found by changing the color of one or more vertices) and starting from a previously colored graph.

The last two classes are significantly different from the original idea of ACO algorithm, and there are divergences if those algorithms are “based on colonies of artificial ants” [40]. Usually, the algorithms that simulate the pheromone trail do it in a similar way: non-adjacent vertices assigned the same color have their pheromone reinforced. The *ColorAnt-RT* algorithms belong to Class 1.

Some algorithms founded on literature are presented on Section 3.

2.3 Register Allocation

Register allocation is one of the most important compiler optimizations, affecting the performance of compiled code [57]. It determines which of the program values (variables and temporaries) should be in machine registers (or memory), during the execution. In a real machine, registers are usually few and fast to access [59, 68], so the problem addressed here is how to minimize the traffic between registers and memory hierarchy. Therefore, the challenge is to relegate least program values to memory, in other words to minimize the number of spills (the values relegate to memory).

Register allocation can be mapped as GCP [15, 34]. However, there is a slight variation: it is forced to eliminate conflicting vertices, besides coloring the graph with just k colors (registers).

A graph coloring register allocation can be briefly described as follows. First, the register allocator [28] generates a so-called interference graph [57], whose vertices represent program values and real registers and whose edges represent interferences. In this graph, an edge (interference) is added either if two values are simultaneous live or a value cannot be (or should not) allocated to that register. After that, it will color the vertices with k colors, so that any two adjacent vertices have different colors. Finally, the allocator will allocate each value to register that has the same color.

In applications where compilation time is a concern, such as dynamic compilation systems [4, 42, 70], researchers try to balance compilation time and code quality. In this context, they do not choose a register allocation algorithm based on graph coloring, because it is a complex algorithm and a time-consuming register allocator. However, allocators [44, 56, 61, 73] that are considered faster than those based on graph coloring result in code that is not as efficient as that obtained by a graph coloring register allocator (GCRA) [2, 19, 57, 66].

3 Algorithms

There are several heuristic algorithms to find a solution to GCP, many of them based on approaches like Evolutionary Algorithms [30], Tabu-Search [30, 38], and ACO [21]. The literature shows that the application of ACO metaheuristic to GCP has some competitive results. In this context, we are interest in the *ColorAnt-RT* algorithms, an ACO algorithms that use a local search method based on reactive tabu search, in order to improve the results.

We present comparisons among the three variations of *ColorAnt-RT* algorithms: *ColorAnt₁-RT*, *ColorAnt₂-RT* and *ColorAnt₃-RT*. They are different in the way that the pheromone trail is manipulated.

3.1 The Literature

The first algorithm that used ant colonies to color graphs was *ANTCOL* [21]. In this algorithm, each ant tries to find the minimum value of k , using a constructive methods based on *RLF* (Recursive Large First) [12] and *Dsatur* [49]. A matrix $P_{|V| \times |V|}$ keeps the experience founded in the constructions (pheromone). The trail between two non-adjacent vertices, with the same color, is reinforced with the inverse of the number of colors founded by an ant. The results of *ANTCOL* were not the best ones, but they were good enough to encourage new studies. In *ColorAnt-RT* algorithms, the treatment of the pheromone matrix is the same of *ANTCOL*. The difference consists in the use of the probability, which, for both, involves pheromone and heuristic information. In *ANTCOL* it is used to choose a new vertex to be colored, and in *ColorAnt-RT* it is used to choose a color to assign a vertex.

A different approach (for k -GCP) works with each ant moving to a adjacent vertex [18]. On new vertex, the ant changes the current color, trying to minimize the conflicts. All ants work together on one solution, and uses the experience from old events. The results presented just compare the algorithm with *ANTCOL*. This approach fits in Class 2 and does not resemble with what is done by *ColorAnt-RT*.

Another algorithm for k -GCP works with each ant as an iterative procedure, which tries to minimize the number of conflicts [65]. The pheromone trail is updated based on a graph G' , initially equals to G , in which

edges are being added in case of many ants assign different colors to non-adjacent vertices. It belongs to Class 3, so it does not resemble *ColorAnt-RT*.

Another algorithm for k -GCP works with each ant trying to color only one vertex [39]. In this case, the colony finds just one solution. A color, among the k possible ones, is assigned to each ant, and k ants are positioned at each vertex. A procedure based on *Dsatur* chooses a vertex and assign to it the color of an ant. Based on heuristic information and pheromone trail, the ants walk through the graph changing the color of the vertices. It was compared to *Dsatur*, *ANTCOL*, *Tabucol* [30] and *HCA* [30]. The algorithm was only better than *Dsatur* and *ANTCOL*, for some instances. It also belongs to Class 2, not being similar to *ColorAnt-RT*.

A recent algorithm, *ALS-COL* (Ant Local Search) [60], implements each ant as a local search method, derived from tabu search. It works modifying classes of color (C_1, \dots, C_k legal classes, and the class C_{k+1} of non-colored vertices). A neighbor solution is founded by moving a vertex $v \in C_{k+1}$ to any class C_c and moving the neighbors of v that are in C_c to C_{k+1} . The move (v, c) is chosen in two steps: one is based on heuristic information and the other is based on pheromone value (treated as in *ANTCOL*). It was compared with *PartialCol* [10], *Tabucol*, *HCA*, Morgenstern algorithm (*MOR*) [55] and Malaguti, Monacie and Toth algorithm (*MMT*) [53]. It founded the chromatic number or the best known value for several instances. Belonging to Class 3, it is also different from *ColorAnt-RT*.

3.2 The *ColorAnt-RT* Algorithms

The three *ColorAnt-RT* algorithms use as constructive method (for each ant) an algorithm suggested along with *ANTCOL* [21], which tries to color a graph with k fixed colors. Such algorithm will be called here *Ant-Fixed.k*, and it is presented in Algorithm 2.

Algorithm 2 *Ant-Fixed.k*.

```

ANT_FIXED_K( $G = (V, E)$ ,  $k$ )    //  $V$ : vertices;  $E$ : edges
1   $NC = V$ ;                      // set of non-colored vertices
2   $s(i) = 0 \quad \forall i \in V$ ;      //  $s$  maps a vertex to a color
3  while  $NC \neq \{\}$  do
4      choose a vertex  $v$  with the highest degree of saturation in  $NC$ ;
5      choose a color  $c \in 1..k$  with probability  $p$  according to Equation 1;
6       $s(v) = c$ ;
7       $NC = NC \setminus \{v\}$ ;
8  return  $s$ ;    // return solution constructed

```

To construct a solution s , *Ant-Fixed.k* performs two tasks. First, it chooses a vertex v without color with the highest degree of saturation¹, and after choosing a color c to assign v . The color c is chosen based on probability p , as follows:

$$p(s, v, c) = \frac{\tau(s, v, c)^\alpha \cdot \eta(s, v, c)^\beta}{\sum_{i \in \{1, \dots, k\}} \tau(s, v, i)^\alpha \cdot \eta(s, v, i)^\beta} \quad (1)$$

where α and β are parameters of the algorithm that control the influence of the values associated to them.

The pheromone trail τ and heuristic information η are as follows:

$$\tau(s, v, c) = \begin{cases} 1 & \text{if } C_c(s) = \{\} \\ \sum_{u \in C_c(s)} P_{uv} & \text{otherwise} \end{cases} \quad (2)$$

$$\eta(s, v, c) = \frac{1}{|N_{C_c(s)}(v)|} \quad (3)$$

where P_{uv} is the pheromone trail between vertices u and v , $C_c(s)$ is the color class c of solution s (the set of vertices already colored with c), and $N_{C_c(s)}(v)$ are the vertices $x \in C_c(s)$ adjacent to v in s .

¹Degree of saturation is the number of different colors that were already assigned to the adjacent vertices of an uncolored vertex.

The pheromone trail, stored on matrix $P_{|V| \times |V|}$, is initialized with 1 for each edge between non-adjacent vertices and with 0 for each edge between adjacent vertices. Updating the pheromone trail involves the persistence of the current trail by a ρ factor, meaning that $1 - \rho$ is the evaporation rate. Edges between pairs of non-adjacent vertices are reinforced when they receive the same color. The evaporation (Equation 4), and the general form of depositing pheromone (Equation 5) are as follows:

$$P_{uv} = \rho P_{uv} \quad \forall u, v \in V \quad (4)$$

$$P_{uv} = P_{uv} + \frac{1}{f(s)} \quad \forall u, v \in C_c(s) \mid (u, v) \notin E, c = 1..k \quad (5)$$

where $C_c(s)$ is the set of vertices colored with c in solution s and f is the objective function, which returns the number of conflicting vertices of that solution.

The difference between the three versions of *ColorAnt-RT* are:

- *ColorAnt₁-RT*: each ant of the colony is used to reinforce the trail, besides the solution of best ant of the colony in a cycle (s'), and the solution of best ant so far (s^*);
- *ColorAnt₂-RT*: only s' and s^* are used to reinforce the trail;
- *ColorAnt₃-RT*: s' and s^* do not reinforce the pheromone trail simultaneously, initially s' does it more often than s^* . A gradual exchange on this frequency is done based on the maximum number of cycles: at each interval of a fixed number of cycles, the number of cycles in which s^* will reinforce the trail (instead of s') is increased by one.

The three *ColorAnt-RT* algorithms utilize a local search method to improve the results of their solutions: the reactive tabu search *React-Tabucol* (RT) [10]. In *ColorAnt₁-RT* and *ColorAnt₂-RT*, the local search is applied only to the best ant of the colony, at the end of a cycle. In *ColorAnt₃-RT*, the local search is applied to all ants of the colony on every cycle.

The local search *React-Tabucol* is as follows. Given the objective function f , which returns the number of conflicting edges, a solution space S where each solution is a set of k color classes and all the vertices are colored (with or without conflicting vertices), and an initial solution $s_0 \in S$, f must be minimized over S . A *move* consists in changing the color of only one vertex, and it occurs between two *neighbor* solutions. When it is performed, the inverse of that move is stored in a *tabu list*, meaning that for the next tl (*tabu tenure*) iterations that move cannot be performed again. The next solution must be generated by a non-tabu move and it must have the minimum number of conflict vertices between all the possible neighbor solutions. *React-Tabucol* is presented in Algorithm 3.

Algorithm 3 *React-Tabucol* [39].

```

REACT-TABUCOL( $G = (V, E)$ ,  $k$ ,  $s_0 = \{C_1, \dots, C_k\}$ )
1   $s = s_0$ ;
2   $s^* = s$ ;
3   $lista\_tabu = \{\}$ ;
4   $cycles = 0$ ;
5  initialize  $tl$ ;
6  while  $cycles < max\_cycles$  do
7      choose a move  $(v, c) \notin lista\_tabu$  with the minimum value for  $\delta(v, c)$ ;
          // where  $\delta(v, c) = f(s \cup (v, c)) - f(s)$ 
8       $s = (s \cup (v, c)) \setminus (v, s(v))$ ;
9      update  $tl$  according to reactive tabu scheme;
10      $lista\_tabu = lista\_tabu \cup \{(v, s(v))\}$ ; // for  $tl$  iterations
11     if  $f(s) < f(s^*)$  then
12          $s^* = s$ ;
13      $cycles = cycles + 1$ ;
14 return  $s^*$ ;

```

The three *ColorAnt-RT* algorithms are resumed in Algorithm 4.

Algorithm 4 *ColorAnt-RT*.

```

COLORANT-RT( $G = (V, E), k$ )
1   $P_{uv} = 1 \quad \forall (u, v) \notin E$ ;
2   $P_{uv} = 0 \quad \forall (u, v) \in E$ ;
3   $f^* = \infty$ ;      // best value for the objective function so far
4   $cycle = 0$ ;
5   $phero\_var = 0$ ;
6  while ( $cycle < max\_cycles$ ) or ( $CPUtime < max\_cpu\_time$ ) or (a proper solution is founded) do
    // Line 7 exists only in  $CA_1$ -RT
7     $\Delta P_{uv} = 0 \quad \forall u, v \in V$ 
8     $f' = \infty$ ;      // best value function in a cycle
9    for  $a = 1$  to  $nants$  do
10      $s = \text{ANT\_FIXED\_K}(G, k)$ ;
    // Line 11 exists only in  $CA_1$ -RT
11      $\Delta P_{uv} = \Delta P_{uv} + \frac{1}{f(s)} \quad \forall u, v \in C_c(s) \mid (u, v) \notin E, c = 1..k$ ;
    // Line 12 exists only in  $CA_3$ -RT
12      $s = \text{REACT\_TABUCOL}(G, k, s)$ ;
13     if  $f(s) == 0$  or  $f(s) < f'$  then
14          $s' = s$ ;
15          $f' = f(s')$ ;
    // Line 16 exists only in  $CA_1$ -RT and  $CA_2$ -RT
16      $s' = \text{REACT\_TABUCOL}(G, k, s')$ ;
17     if  $f' < f^*$  then
18          $s^* = s'$ ;
19          $f^* = f(s^*)$ ;
20      $P_{uv} = \rho P_{uv} \quad \forall u, v \in V$ ; // according to Equation 4
    // Line 21 exists only in  $CA_1$ -RT
21      $P_{uv} = P_{uv} + \Delta P_{uv} \quad \forall u, v \in V$ ;
    // Lines 22–23 exist only in  $CA_1$ -RT and  $CA_2$ -RT
22      $P_{uv} = P_{uv} + \frac{1}{f(s')}$   $\forall u, v \in C_c(s') \mid (u, v) \notin E, c = 1..k$ ;
23      $P_{uv} = P_{uv} + \frac{1}{f(s^*)}$   $\forall u, v \in C_c(s^*) \mid (u, v) \notin E, c = 1..k$ ;
    // Next lines exist only in  $CA_3$ -RT:
24     if  $cycle \bmod \sqrt{max\_cycles} == 0$  then
25          $phero\_counter = \lfloor cycle \div \sqrt{max\_cycles} \rfloor$ ;
26     if  $phero\_counter > 0$  then
27          $P_{uv} = P_{uv} + \frac{1}{f(s^*)}$ 
             $\forall u, v \in C_c(s^*) \mid (u, v) \notin E, c = 1..k$ ; // according to Equation 5
28     else
29          $P_{uv} = P_{uv} + \frac{1}{f(s')}$ 
             $\forall u, v \in C_c(s') \mid (u, v) \notin E, c = 1..k$ ; // according to Equation 5
30      $cycle = cycle + 1$ ;
31      $phero\_counter = phero\_counter - 1$ ;

```

3.3 The Performance of *ColorAnt-RT* Algorithms

In this section, the results obtained by the three *ColorAnt-RT* algorithms are reported. They are also compared with five other algorithms from the literature.

3.3.1 Methodology

The three *ColorAnt-RT* algorithms were implemented in C language and executed in an Intel Xeon E5504 of 2.00 GHz, 24GB RAM running Ubuntu with kernel 3.2.0-24-generic.

The experiments were performed in forty-nine graphs of DIMACS Challenge [45], which are used in many papers in the literature [10, 30, 38, 53, 55, 60]. The instances are:

- **dsjc250.1, dsjc250.5, dsjc500.1, dsjc500.5 and dsjc1000.1**²: standard random graphs **dsjcn.d** have n vertices and any two vertices have a probability d of being adjacent;
- **dsjr500.1, dsjr500.1c and dsjr500.5**²: geometric random graphs **dsjrn.d** are generated by choosing n points uniformly at random in a square, and by setting edges between pairs of vertices situated within a distance less than d . A 'c' letter at the end of the name means that the graph is the complement³ of the respective geometric random graph;
- **miles500, miles750 and miles1000**²: graph instances similar to geometric graphs (**dsjrn.d**), where the vertices are placed in space, and two vertices are connected if they are close enough. These graphs represent real case of cities, and the distances between two vertices are also real case;
- **flat300_26.0, flat300_28.0, flat1000_50.0, flat1000_60.0 and flat1000_76.0**²: **flatn- χ .0** graphs are generated by partitioning n vertices into χ classes (almost of equal size), and by selecting edges between vertices of different classes, in this way they have a chromatic number χ . It is used randomness in the generation of these graphs;
- **le450.15c, le450.15d, le450.25c and le450.25d**²: **le450- χ** graphs always have 450 vertices, and a chromatic number χ . It is used randomness in the generation of these graphs;
- **myciel3, myciel4, myciel5 and myciel6**²: graph instances based on the Mycielski transformation. Their resolution are difficult because they have no triangles, but the coloring number increases in graph instance size;
- **1-insertions_6, 2-insertions_5, 4-insertions_4, 2-fullIns_5, 3-fullIns_4 and 4-fullIns_4**⁴: these graph instances are a generalization of myciel graphs, with inserted nodes to increase graph size but not density;
- **queen6.6, queen7.7, queen8.8 and queen9.9**²: considering a $n \times n$ chessboard, an instance graph of this class has n^2 nodes, each one representing a square of the board. There is an edge between two nodes if the corresponding squares are in the same row, column, or diagonal;
- **ash331gpia, ash608gpia and will199gpia**⁴: graph instances obtained from a matrix partitioning problem, in the segmented columns approach to determine sparse Jacobian matrices;
- **fpsol2.i.1, fpsol2.i.2, fpsol2.i.3, inithx.i.1, inithx.i.2, inithx.i.3, mulsol.i.1, mulsol.i.2, mulsol.i.3, zeroin.i.1, zeroin.i.2 and zeroin.i.3**²: graph instances based on register allocation.

ColorAnt₃-RT has three stop conditions: (1) a proper solution is founded which the k given by parameter; (2) the maximum number of 841 cycles is reached; or (3) a time limit of one hour is reached. *ColorAnt₂-RT* has two stop conditions, which are (1) and (3) mentioned for *ColorAnt₃-RT*. *ColorAnt₁* has the same two stop conditions of *ColorAnt₂-RT*.

The choice of 841 as the maximum number of cycles is due to the way that the reinforcement of the pheromone trail is done. Each interval of cycles has a fixed size ($\sqrt{\text{max_cycles}}$). When an interval of cycles of that size is executed, the number of cycles in which s^* will reinforce the trail (instead of s') is increased by one. In this way, we choose the maximum number of cycles as a perfect square ($\sqrt{841} = 29$). In fact, it is an empirical number. The goal of this value is to give chance to s^* and s' is used to reinforce the pheromone trail.

For each graph instance and k value, each *ColorAnt-RT* algorithm was executed 10 times. In a standard run (execution), the value of k is initialized with the value of $k^* + 5$, and it is decremented by 1, reaching the value k^* . The results reported are the successful runs⁵ with the smallest value of k , for what there were at least one

²Available in <http://mat.gsia.cmu.edu/COLOR/instances.html>, accessed in December 2012.

³The complement of a graph $G = (V, E)$ is a graph $G' = (V, E')$ (with the same vertices of G) in which $e \in E'$ if and only if $e \notin E$.

⁴Available in <http://mat.gsia.cmu.edu/COLOR02/>, accessed in December 2012.

⁵A successful run finds a proper solution.

run without conflicting vertices (see Tables 2 and 3). If there are no successful runs, new runs are done starting the value of k in $k^* + 30$, following the same scheme of the standard runs, and the results are reported in the same way.

An important issue in the execution of heuristic algorithms is the calibration of the parameters. In this way, experiments were done in order to find good values for the parameters number of ants (*nants*), α , β , ρ and number of cycles of the local search (*ls_cycles*). We used the strategy of calibrating each parameter independently of each others. Initially were calibrated α and β , assigning 1, 2, and 3 to the parameter x using the Algorithm 5.

Algorithm 5 Calibrate- α - β

CALIBRATE- α - β ($G = (V, E)$, k^* , x)

```

1  for  $\alpha = 1$  to 20 do
2    for  $\beta = 1$  to 20 do
3       $\rho = 0.5$ ;
4      nants = 50;
5      cycles = 50;
6      ls_cycles = 0;
7      for run = 1 to 3 do
8        COLORANT $x$ -RT( $G$ ,  $k^*$ ,  $\alpha$ ,  $\beta$ ,  $\rho$ , nants, cycles, ls_cycles);
9      Store the average of the quantity of conflicts for this configuration
10 Return the configuration with the smallest quantity of conflicts

```

After the calibration of α and β , the next step was to calibrate ρ . Basically, it was used the same strategy as before as can be seen in the Algorithm 6.

Algorithm 6 Calibrate- ρ

CALIBRATE- ρ ($G = (V, E)$, k^* , x)

```

1  for  $\rho = 0.0$  to 1.0 step 0.1 do
2     $\alpha = 3$ ;
3     $\beta = 5$ ;
4    nants = 50;
5    cycles = 50;
6    ls_cycles = 0;
7    for run = 1 to 3 do
8      COLORANT $x$ -RT( $G$ ,  $k^*$ ,  $\alpha$ ,  $\beta$ ,  $\rho$ , nants, cycles, ls_cycles);
9    Store the average of the quantity of conflicts for this configuration
10 Return the configuration with the smallest quantity of conflicts

```

Finally we calibrated *nants* and *ls_cycles*. The strategy used is the same as presented in Algorithm 6, with the difference that ρ was fixed in 0.5, *nants* varied between 20 and 500, and *ls_cycles* varied between 50 and 2000.

The values obtained by the calibrations are presented in the Table 1. This table presents the characteristics of the graph instances, and the parameters used by the *ColorAnt₁-RT*, *ColorAnt₂-RT* and *ColorAnt₃-RT* algorithms. The characteristics are shown in the group of columns under “graph”. In this group, the first column presents the name of the graph instance. The second, third and fourth column contain the number of vertex ($|V|$), the number of edges ($|E|$) and the density (D) of the graph, respectively. The fifth column shows the pair (χ/k^*) , where a “?” denotes that the value of χ is not known for that instance, and k^* is the value of the best known solution founded and reported in the literature until the moment.

Table 1: Characteristics of the graph instances and parameters of *ColorAnt₁-RT*, *ColorAnt₂-RT* and *ColorAnt₃-RT*.

Graph					<i>ColorAnt₁-RT</i>					<i>ColorAnt₂-RT</i>					<i>ColorAnt₃-RT</i>				
<i>Name</i>	$ V $	$ E $	D	(χ/k^*)	<i>nants</i>	α	β	ρ	<i>ls_cycles</i>	<i>nants</i>	α	β	ρ	<i>ls_cycles</i>	<i>nants</i>	α	β	ρ	<i>ls_cycles</i>
dsjc250.1	250	3218	0.10	(?/8)	20	2	1	0.1	1550	300	6	7	0.6	1900	180	1	2	0.6	650
dsjc250.5	250	15668	0.50	(?/28)	20	2	11	0.9	1600	40	2	1	0.6	1750	460	2	8	0.4	1900
dsjc500.1	500	12458	0.10	(?/12)	360	3	6	0.1	1950	120	20	18	0.8	1800	320	2	14	0.7	1700
dsjc500.5	500	62624	0.50	(?/48)	200	2	10	0.2	1900	140	7	18	0.3	1400	320	2	12	0.0	1800
dsjc1000.1	1000	49629	0.10	(?/20)	20	3	10	0.0	1950	320	2	5	0.1	1850	440	3	11	0.3	1850
dsjr500.1	500	3555	0.03	(12/12)	20	1	1	0.0	50	20	1	1	0.0	50	20	1	1	0.0	50
dsjr500.1c	500	121275	0.97	(?/85)	280	4	8	0.6	1350	160	9	7	1.0	950	80	7	1	1.0	1450
dsjr500.5	500	58862	0.47	(122/122)	180	1	20	1.0	600	420	2	17	0.0	1550	140	1	20	0.9	1750
miles500	128	2340	0.29	(20/20)	20	1	1	0.0	50	20	1	1	0.0	50	20	1	1	0.0	50
miles750	128	4226	0.52	(31/31)	20	1	1	0.0	50	20	8	1	0.0	50	20	1	1	0.0	50
miles1000	128	6432	0.79	(42/42)	20	1	1	0.0	50	20	7	1	0.0	50	20	1	1	0.0	50
flat300_26.0	300	21633	0.48	(26/26)	60	4	5	0.9	1550	60	2	10	0.8	1550	400	1	5	0.6	1950
flat300_28.0	300	21695	0.48	(28/28)	100	2	7	0.4	1550	300	1	2	0.3	1950	460	1	11	0.6	1250
flat1000_50.0	1000	245000	0.49	(50/50)	360	2	6	0.0	1900	460	16	18	0.4	1900	200	2	12	0.7	1800
flat1000_60.0	1000	245830	0.49	(60/60)	240	2	9	0.0	1800	400	8	18	0.2	1800	340	3	19	0.0	1750
flat1000_76.0	1000	246708	0.49	(76/76)	400	2	6	0.4	1950	320	10	13	0.0	1900	480	2	12	0.7	1600
le450_15c	450	16680	0.17	(15/15)	260	19	5	0.5	1950	240	1	11	0.8	1450	160	1	11	0.7	1850
le450_15d	450	16750	0.17	(15/15)	220	12	6	0.0	1700	140	2	14	0.8	1600	460	1	7	0.2	1700
le450_25c	450	17343	0.17	(25/25)	40	2	20	0.3	1250	280	2	14	0.2	1650	220	1	19	0.0	1350
le450_25d	450	17425	0.17	(25/25)	40	2	8	0.7	1750	360	2	19	0.1	1100	420	2	2	0.2	1700
myciel3	11	20	0.36	(4/4)	20	1	1	0.0	50	20	1	1	0.0	50	20	1	1	0.0	50
myciel4	23	71	0.28	(5/5)	20	1	1	0.0	50	20	1	1	0.0	50	20	1	1	0.0	50
myciel5	47	236	0.22	(6/6)	20	1	1	0.0	50	20	1	1	0.0	50	20	1	1	0.0	50
myciel6	95	755	0.17	(7/7)	20	1	1	0.0	50	20	1	1	0.0	50	20	1	1	0.0	50
1-insertions_6	607	6337	0.03	(7/7)	20	1	1	0.0	50	20	1	1	0.0	50	20	1	1	0.0	50
2-insertions_5	597	3936	0.02	(6/6)	20	4	1	0.5	250	200	5	1	0.5	1550	180	2	7	0.0	150
4-insertions_4	475	1795	0.02	(5/5)	480	2	1	0.7	1950	20	2	3	0.1	700	340	1	1	0.2	200
2-fullIns_5	852	12201	0.03	(7/7)	20	3	4	0.7	50	300	1	3	0.0	150	40	2	1	0.0	100
3-fullIns_4	405	3524	0.04	(7/7)	20	1	1	0.0	50	20	1	1	0.0	50	20	1	1	0.0	50
4-fullIns_4	690	6650	0.03	(8/8)	20	1	3	0.0	50	20	1	1	0.0	50	20	1	1	0.0	50
queen6_6	36	580	0.92	(7/7)	20	1	1	0.0	100	20	1	1	0.0	100	20	1	1	0.0	50
queen7_7	49	952	0.81	(7/7)	20	1	1	0.0	200	20	4	1	0.0	50	20	1	1	0.0	50
queen8_8	64	1456	0.72	(9/9)	20	1	1	0.0	100	20	1	1	0.0	200	20	1	1	0.0	50
queen9_9	81	2112	0.65	(10/10)	20	1	1	0.0	300	20	1	1	0.0	150	20	1	1	0.0	50
ash331gpia	662	4185	0.02	(4/4)	20	1	1	0.0	50	20	1	1	0.0	50	20	1	1	0.0	50
ash608gpia	1216	7844	0.01	(4/4)	20	1	1	0.0	50	20	1	1	0.0	50	20	1	1	0.0	50
will199gpia	701	7065	0.03	(7/7)	20	1	1	0.0	50	20	1	1	0.0	50	20	1	1	0.0	50
fpsol2.i.1	496	11654	0.09	(65/65)	20	9	1	0.0	50	20	12	1	0.0	50	20	4	1	0.0	50
fpsol2.i.2	451	8691	0.09	(30/30)	40	20	2	0.0	50	40	20	2	0.8	100	20	17	1	0.0	50
fpsol2.i.3	425	8688	0.10	(30/30)	20	16	2	0.0	100	20	13	2	0.0	50	20	18	1	0.0	50
inithx.i.1	864	18707	0.05	(54/54)	20	19	3	0.0	100	140	9	4	0.0	100	20	16	2	0.0	50
inithx.i.2	645	13979	0.07	(31/31)	60	16	3	0.1	50	40	20	3	0.8	50	20	6	3	0.0	50
inithx.i.3	621	13969	0.07	(31/31)	20	1	4	0.0	50	20	8	4	0.0	100	20	7	3	0.0	50
mulsol.i.1	197	3925	0.20	(49/49)	20	1	1	0.0	50	20	8	1	0.0	50	20	1	1	0.0	50
mulsol.i.2	188	3885	0.22	(31/31)	20	1	1	0.0	50	20	13	1	0.0	50	20	4	1	0.0	50
mulsol.i.3	184	3916	0.23	(31/31)	20	13	1	0.0	50	20	12	1	0.0	50	20	5	1	0.0	50
zeroin.i.1	211	4100	0.19	(49/49)	20	1	1	0.0	50	20	1	1	0.0	50	20	1	1	0.0	50
zeroin.i.2	211	3541	0.16	(30/30)	20	1	1	0.0	50	20	1	1	0.0	50	20	1	1	0.0	50
zeroin.i.3	206	3540	0.17	(30/30)	20	1	1	0.0	50	20	2	1	0.0	50	20	1	1	0.0	50

3.3.2 The Results

Tables 2 and 3 report the results obtained by *ColorAnt₃-RT*, *ColorAnt₂-RT* and *ColorAnt₁-RT*. In this table, the first column indicates the name of the graph instance, and the second column shows the pair (χ/k^*) . The third column reports the value of k . Next, we have three groups, one for each algorithm. Each group has the columns: S/10, where S represents the amount of successful runs in 10 runs for that color; Time(s) is the average CPU-time used (in seconds); and Cfs, which reports the average number of conflicting vertices of each attempt (composed by the 10 runs). The number of 10 runs in each attempt is the same as used in [60].

The calibration of the parameters showed in Table 1 lead several graph instances to have the parameter $\rho = 0.0$. Among the 49 instances, this occurs 33 times for *ColorAnt₁-RT*, 31 times for *ColorAnt₂-RT* and 35 times to *ColorAnt₃-RT*. In these cases, the algorithm never “forget” the pheromone deposits, so that, the ants take all the history of pheromone updating into account when deciding where to go. For that instances, the calibration process has founded better solutions when considering all the history, instead of the ones that “forget”

Table 2: Results and CPU time obtained by *ColorAnt₁-RT*, *ColorAnt₂-RT* and *ColorAnt₃-RT*

Graph	(χ/k^*)	k	<i>ColorAnt₁-RT</i>			<i>ColorAnt₂-RT</i>			<i>ColorAnt₃-RT</i>		
			S/10	Time(s)	Cfs	S/10	Time(s)	Cfs	S/10	Time(s)	Cfs
dsjc250.1	(?/8)	8	10	9.731	0	10	144.111	0	10	6.426	0
dsjc250.5	(28/28)	29	10	54.241	0	9	664.322	0.2	10	50.220	0
		28	3	2772.877	1.8	-	-	-	2	629.074	2.6
dsjc500.1	(?/12)	13	10	526.874	0	10	11.930	0	10	44.110	0
dsjc500.5	(48/48)	53	4	3075.580	2.1	10	515.037	0	10	297.618	0
		52	-	-	-	5	2465.660	1.9	9	682.204	0.4
		51	-	-	-	-	-	-	3	1690.425	2.8
dsjc1000.1	(?/20)	22	10	63.622	0	10	552.081	0	9	426.408	0.3
dsjc				1289.737			767.621			559.289	
dsjr500.1	(12/12)	12	10	226.186	0	10	13.389	0	10	3.057	0
dsjr500.1c	(?/85)	85	2	3369.691	1.7	6	1719.053	0.6	9	29.494	0.1
dsjr500.5	(122/122)	123	10	17.696	0	10	26.192	0	9	164.488	0.1
		122	8	1709.114	0.2	-	-	-	1	625.379	1.5
dsjr				1768.33			586.211			219.31	
miles500	(20/20)	20	10	0.065	0	9	390.794	0.2	10	0.014	0
miles750	(31/31)	31	10	0.560	0	7	1105.952	0.6	6	0.791	0.8
miles1000	(42/42)	42	10	0.287	0	10	0.028	0	7	0.680	0.6
miles				0.304			498.925			0.495	
flat300_26_0	(26/26)	30	-	-	-	-	-	-	1	1043.777	8.3
flat300_28_0	(28/28)	33	9	708.358	0.3	10	63.858	0	10	126.352	0
		32	-	-	-	5	2259.892	1.5	6	434.151	1.1
flat1000_50_0	(50/50)	55	-	-	-	-	-	-	1	3638.892	651.5
flat1000_60_0	(60/60)	90	-	-	-	-	-	-	-	-	-
		65	-	-	-	-	-	-	-	-	-
flat1000_76_0	(76/76)	95	-	-	-	5	2910.029	2.1	5	3430.387	1.5
		81	-	-	-	-	-	-	-	-	-
flat				708.358			2584.961			2136.802	
le450_15c	(15/15)	15	-	-	-	5	2149.167	3.5	8	222.446	1.4
le450_15d	(15/15)	20	2	3241.958	8.2	10	28.641	0	10	60.347	0
		19	-	-	-	10	26.470	0	10	81.990	0
		18	-	-	-	10	28.419	0	10	94.763	0
		17	-	-	-	10	25.655	0	10	110.225	0
		16	-	-	-	10	49.456	0	10	129.024	0
		15	-	-	-	3	2811.537	2.1	10	168.655	0
le450_25c	(25/25)	27	10	4.883	0	10	148.077	0	10	15.591	0
		26	7	1235.019	1.3	-	-	-	2	456.198	3.4
le450_25d	(25/25)	26	3	2690.312	2.8	1	3270.295	4.6	2	724.874	3.4
le450				2389.096			2094.769			393.043	
myciel3	(4/4)	4	10	0.0003	0	10	0.0003	0	10	0.0003	0
myciel4	(5/5)	5	10	0.0006	0	10	0.0006	0	10	0.0006	0
myciel5	(6/6)	6	10	0.002	0	10	0.001	0	10	0.002	0
myciel6	(7/7)	7	10	0.005	0	10	0.004	0	10	0.005	0
myciel				0.002			0.001			0.002	

some information by evaporation of pheromone.

Despite the parameter ρ has no typical value for ACO algorithms, good solutions were founded as we can see in the instances **miles**, **myciel**, **fullIns**, **queen**, **gpia**, **fpsol**, **inithx**, **multsol** and **zeroIn**, where we can highlight that k^* was founded in all of these cases.

An interesting aspect of the classes that should be taken into account is the use of randomness in the creation of the graph. The ones with randomness tend to be more difficult to solve, than the ones that not use randomness. The classes **dsjc**, **dsjr**, **flat** and **le450** use randomness, while the classes **miles**, **myciel**, **insertions**, **fullIns**, **queen**, **gpia**, **fpsol**, **inithx**, **multsol**, and **zeroIn** do not use randomness.

With randomness the *ColorAnts-RT* do not found k^* in several instances, namely the ones of kind **dsjc** with 500 and 1000 vertices, all the **flat** instances, and the **le450s** with chromatic number 25. On the other hand, without randomness, k^* was reached for all the instances. In this context, we can oppose the **dsjr** instances, that use randomness, and the **miles** instances, that are similar to the formers, but without the use of randomness: the **miles** were easier to solve than the **dsjrs**, even in the CPU time.

The *ColorAnt₁-RT* shows good performance in **dsjc** instances, as we can see by the fact that in **dsjc250.5** it founded k^* , while the *ColorAnt₂-RT* did not. On the other hand, *ColorAnt₁-RT* founded only the best k (founded by *ColorAnt₃-RT*) plus 2 for the **dsjc500.5**. In general, in **dsjcs** *ColorAnt₁-RT* looses in CPU time to the others. However, it is not precise to compare the times between algorithms that found different values of k . In **dsjrs** *ColorAnt₁-RT* has a better performance, finding k^* for **dsjr500.5** when *ColorAnt₂-RT* did not, with the advantage that there is no case that *ColorAnt₁-RT* founds worst k than the other algorithms. The CPU time of **dsjrs** follows the **dsjcs**.

Table 3: Results and CPU time obtained by *ColorAnt₁-RT*, *ColorAnt₂-RT* and *ColorAnt₃-RT*

Graph	(χ/k^*)	k	<i>ColorAnt₁-RT</i>			<i>ColorAnt₂-RT</i>			<i>ColorAnt₃-RT</i>		
			S/10	Time(s)	Cfs	S/10	Time(s)	Cfs	S/10	Time(s)	Cfs
1-insertions.6	(7/7)	7	-	-	-	10	3.272	0	10	2.997	0
2-insertions.5	(6/6)	6	10	0.145	0	10	0.796	0	10	0.680	0
4-insertions.4	(5/5)	5	10	1.986	0	10	0.049	0	10	0.952	0
insertions				1.066			1.372			1.543	
2-fullIns.5	(7/7)	7	10	0.321	0	10	2.468	0	10	7.033	0
3-fullIns.4	(7/7)	7	10	3.068	0	10	0.653	0	10	0.905	0
4-fullIns.4	(8/8)	8	10	0.395	0	10	4.783	0	10	3.922	0
fullIns				1.261			2.635			3.953	
queen6.6	(7/7)	7	10	0.060	0	10	0.433	0	10	0.004	0
queen7.7	(7/7)	7	10	0.018	0	7	1214.182	3.8	10	0.015	0
queen8.8	(9/9)	9	10	6.973	0	10	0.278	0	7	0.147	0.6
queen9.9	(10/10)	10	10	0.081	0	10	0.145	0	7	0.249	0.6
queen				1.783			303.76			0.104	
ash331gpia	(4/4)	4	10	169.577	0	10	3.884	0	10	4.039	0
ash608gpia	(4/4)	4	9	2060.892	80	10	25.786	0	10	18.389	0
will199gpia	(7/7)	9	4	2401.957	99.1	10	7.064	0	10	5.573	0
		8	-	-	-	10	7.368	0	10	5.869	0
		7	-	-	-	10	9.478	0	10	6.388	0
gpia				1544.142			13.049			5.943	
fpsol2.i.1	(65/65)	65	6	1515.915	0.4	8	979.765	0.2	6	9.436	0.4
fpsol2.i.2	(30/30)	30	8	730.834	0.2	7	1085.836	0.4	2	10.614	0.8
fpsol2.i.3	(30/30)	30	3	2526.135	0.7	4	2168.797	0.7	8	2.315	0.2
fpsol				1590.961			1411.466			7.455	
inithx.i.1	(54/54)	54	7	1084.534	0.3	6	1831.237	0.5	9	4.799	0.1
inithx.i.2	(31/31)	31	2	2887.077	0.9	2	2890.418	1.1	7	7.340	0.3
inithx.i.3	(31/31)	31	10	8.903	0	6	1443.429	0.4	6	8.989	0.4
inithx				1326.838			2055.028			7.043	
multsol.i.1	(49/49)	49	10	0.991	0	9	365.755	0.1	9	0.541	0.1
multsol.i.2	(31/31)	31	9	432.421	0.1	5	1804.698	0.6	7	1.000	0.3
multsol.i.3	(31/31)	31	6	1443.411	0.4	4	2164.696	0.6	6	1.278	0.4
multsol				625.608			1445.05			0.94	
zeroin.i.1	(49/49)	49	10	0.048	0	10	0.046	0	10	0.031	0
zeroin.i.2	(30/30)	30	10	0.187	0	10	0.124	0	10	0.059	0
zeroin.i.3	(30/30)	30	10	0.093	0	10	0.165	0	10	0.050	0
zeroin				0.109			0.112			0.047	

In *dsjcs* and *dsjrs* the best choice is *ColorAnt₃-RT*, for the main reasons: (1) it founded the best values of k among the algorithms in several instances, even finding various k^* , and (2) it has significant small average CPU time.

For *miles*, a highlight: *ColorAnt₁-RT* has the best times with average smaller than one second, and every run founded the k^* . *ColorAnt₃-RT* has a similar performance, but it was not able to have 10 succeeded runs in *miles750* and *miles1000* instances. The worst performance was with *ColorAnt₂-RT*, that was not able to produce 10 succeeded runs.

In *flat* *ColorAnt₁-RT* founded just one k (that is 5 colors worst from k^*). *ColorAnt₂-RT* founded two values k and *ColorAnt₃-RT* has four (in a total of five). The *flat* revealed to be difficult for *ColorAnts-RT*. It was necessary to use runs starting from $k^* + 30$, as we can see in *flat1000.60.0* and *flat1000.76.0*. No *ColorAnt-RT* founded a solution to *flat1000.60.0* instance, while *ColorAnt₂-RT* and *ColorAnt₃-RT* find $k^* + 19$ (95) solutions for the *flat1000.76.0*. It is difficult to compare CPU time, but considering *ColorAnt₃-RT*, the average of 2136.802s is higher than all other classes of graph instances.

Focusing on solutions founded by the *ColorAnt₃-RT* to instances of 1000 or higher number of vertices, we can compare: (1) the average time of 3430.387s of *flat1000.76.0* (with a $k^* + 19$ solution founded, 25% far from the k^* solution); (2) the *dsjc1000.1*, that has time of 426.408s with a solution $k^* + 2$ (10% far from the k^* solution); and (3) *ash608gpia* that has 1216 vertices, with the time of 18.389s, and a solution k^* in all the runs tried. Randomness was used in the creation of *flat1000.76.0* and *dsjc1000.1*, which can give an explanation of their difficulty compared to *ash608gpia*. Between *flat1000.76.0* and *dsjc1000.1*, we can highlight that *flat* is in general more difficult than *dsjc* as the results indicate.

In *1e450s*, *ColorAnt₁-RT* has not too bad results as in *flats*, but the numbers are not so good, taking into account that the others founded two k^* , and *ColorAnt₁-RT* not. However, it is interesting to note that *ColorAnt₁-RT* founded a better k than *ColorAnt₂-RT* in *1e450.25c*. The *ColorAnt₃-RT* founded the better values of k with the best average CPU time of 393.043s (against very worst CPU time of the others). For the *1e450.15c* and *1e450.15d*, it founded the values of k^* , and for the other two instances, it founded $k^* + 1$.

The class *myciel*, and its derivated *insertions* and *fullIns*, have a good situation: all *ColorAnts-RT* founded

k^* for all instances in all runs with the exceptions of just one case, which is *ColorAnt₁-RT* to **1-insertions.6**. The CPU time of *ColorAnt₁-RT* is the better, but the average times of all the algorithms are not higher than four seconds. For these classes, the best choice is *ColorAnt₂-RT*, because all the runs finding k^* with better average CPU time than *ColorAnt₃-RT*. However, as *ColorAnt₃-RT* also finds all possible k^* in all runs, with average CPU time no higher than 4s.

In **queen** all the algorithms find all k^* , with advantage of *ColorAnt₁-RT* that is well succeeded in all the runs. It has a good average CPU time of 1.783 seconds.

ColorAnt₁-RT is worst in **gpia** class, while *ColorAnt₂-RT* and *ColorAnt₃-RT* founded the best possible quality of solutions. *ColorAnt₃-RT* has the advantage of a better average CPU time: 5.943 seconds, about a half than the *ColorAnt₂-RT*.

All classes taken from the register allocation context (**fpsol**, **inithx**, **mulsol**, and **zeroin**) let all the *ColorAnts-RT* find the k^* . In **zeroin**, all the runs were succeeded, and the average CPU time is under 0.2 seconds. For the other classes, the total of succeeded runs varied from 14 to 22 (in a total of 30). What separates these classes, by the algorithm, is the average CPU time; namely: **fpsol** (1590.961s, 1411.466s, 7.455s), **inithx** (1326.838s, 2055.028s, 7.043s), and **mulsol** (625.608s, 1445.05s, 0.94s). An average of these respective values results in 1181.136s, 1637.181s, 5.146s, respectively. It shows a good CPU time for *ColorAnt₃-RT*, which is about 229 times smaller than the second better (*ColorAnt₁-RT*).

The *ColorAnt₃-RT* is the best choice in the most of classes. However, different situations can be seen: in **miles** *ColorAnt₁-RT* is the best, but *ColorAnt₃-RT* is close to it; in **myciel**, **insertions** and **fullIns** the better choice is *ColorAnt₂-RT*, but it is safe to use *ColorAnt₃-RT* as we have already pointed; and the instances of king **queen** are better solved with *ColorAnt₁-RT*, but *ColorAnt₃-RT* is acceptable. With this in mind, *ColorAnt₃-RT* is the best choice overall.

3.3.3 Comparison with other Algorithms

The results of the three *ColorAnts-RT* were also compared with the results obtained for the following heuristic algorithms: *ALS-COL* [60], *Tabucol* [30, 38], *MMT* [53], *HCA* [30], *MOR* [55], and *PartialCol* [10]. It must be clear that such algorithms were not implemented for this paper. Although the conditions for the execution are distinct, it is possible to realize some comparisons about the quality of the solutions in relation to k^* .

Table 4 presents the results obtained by the three *ColorAnt-RT*, and the other heuristics for just some graph instances. The values on this table are bold when k^* is reached.

Table 4: Values of k founded by *ColorAnt₃-RT* (*CA₃-RT*), *ColorAnt₂-RT* (*CA₂-RT*), *ColorAnt₁* (*CA₁-RT*), *ALS-COL* (*ALS*), *Tabucol* (*TC*), *MMT*, *HCA*, *MOR*, and *PartialCol* (*PC*).

Grafo	(χ/k^*)	<i>CA₁-RT</i>	<i>CA₂-RT</i>	<i>CA₃-RT</i>	<i>ALS</i>	<i>TC</i>	<i>MMT</i>	<i>HCA</i>	<i>MOR</i>	<i>PC</i>
dsjc500.1	(?/12)	13	13	13	12	12	12	12	12	12
dsjc500.5	(?/48)	53	52	51	48	49	48	48	49	49
dsjc1000.1	(?/20)	22	22	22	20	20	20	20	21	21
dsjr500.1c	(?/85)	85	85	85	85	85	85	-	85	85
dsjr500.5	(122/122)	122	123	122	125	126	122	-	123	126
flat300.28.0	(28/28)	33	32	32	29	31	31	31	31	28
flat1000.76.0	(76/76)	-	95	95	85	88	82	83	89	88
le450.15c	(15/15)	-	15	15	15	16	15	15	15	15
le450.15d	(15/15)	20	15	15	15	15	15	15	15	15
le450.25c	(25/25)	26	27	26	26	26	25	26	25	27
le450.25d	(25/25)	26	26	26	26	26	25	26	25	27

In the compared instances of Table 4, *ColorAnt₃-RT* always has better solutions than the other algorithms. Comparing *ColorAnt₃-RT* with the other algorithms, we can see that in the **le450** instances the algorithms have similar behavior. For the other instances the scenario is different.

Flat300.28.0 and **flat1000.76.0** are hard to all the algorithms. Among all of them, **flat300.28.0** has $k = 29$ (k^* is 28) as the better solution and **flat1000.76.0** has $k = 82$ (k^* is 76). Each of these values is reached just once, and by distinct algorithms. *ColorAnt₃-RT* presents the worst results for these instances, 32 for the first and 95 for the second.

The algorithms *ALS*, *MMT* and *HCA* have founded k^* for all **dsjc500** instances. *TC*, *PC* and *MOR* have the same results, except that they have founded $k^* + 1$ to **dsjc500.5**, and that *MOR* reported $k^* + 1$ to **dsjc1000.1**. The *ColorAnt₃-RT* presents worst results: $k^* + 1$, $k^* + 3$, and $k^* + 2$, for **dsjc500.1**, **dsjc500.5**, and **dsjc1000.1**, respectively.

In **dsjr500.1c** and **dsjr500.5**, *ColorAnt₃-RT* has a very nice performance: it founded k^* for the two instances, so it has the same behavior as *MMT* and is better than all others. Although the literature presents efficient

algorithms that outperform *ColorAnt-RT* in some instances, the next section will show that *ColorAnt-RT* is an efficient algorithm for the context of register allocation.

3.3.4 Revisiting The ColorAnt₃-RT

The results presented in the previous sections can raise the issue of robustness, because can be desirable that the same group of parameters covers several instances. Besides, in the previous sections are not clear whether or not is necessary to use local search. Based on these two considerations new experiments were conducted, but in a small set of instances. The methodology of these experiments is:

ColorAnt-RT The previous results indicate that *ColorAnt₃-RT* outperforms its predecessors. As a result, these experiments use the *ColorAnt₃-RT*.

Instances It is essential to analyze those instances, whose the best color founded by *ColorAnt₃-RT* is not the best known coloring.

Parameters The results on Table 1 indicate that heuristic information provided better results, than the pheromone trial. Based on this observation the parameters used by all instances are $\alpha = 1$, $\beta = 9$, and $\rho = 0.1$. Besides, the number of ants ranges from 10 to 100, and the number of local search cycles ranges from 10 to 1000000. The number of ants has a high computational cost, due to this fact the maximum number of ants is only 100. With these ranges, it is possible to identify whether or not local search improves the results.

The Figure 1 shows the best coloring founded by *ColorAnt₃-RT* and the CPU time.

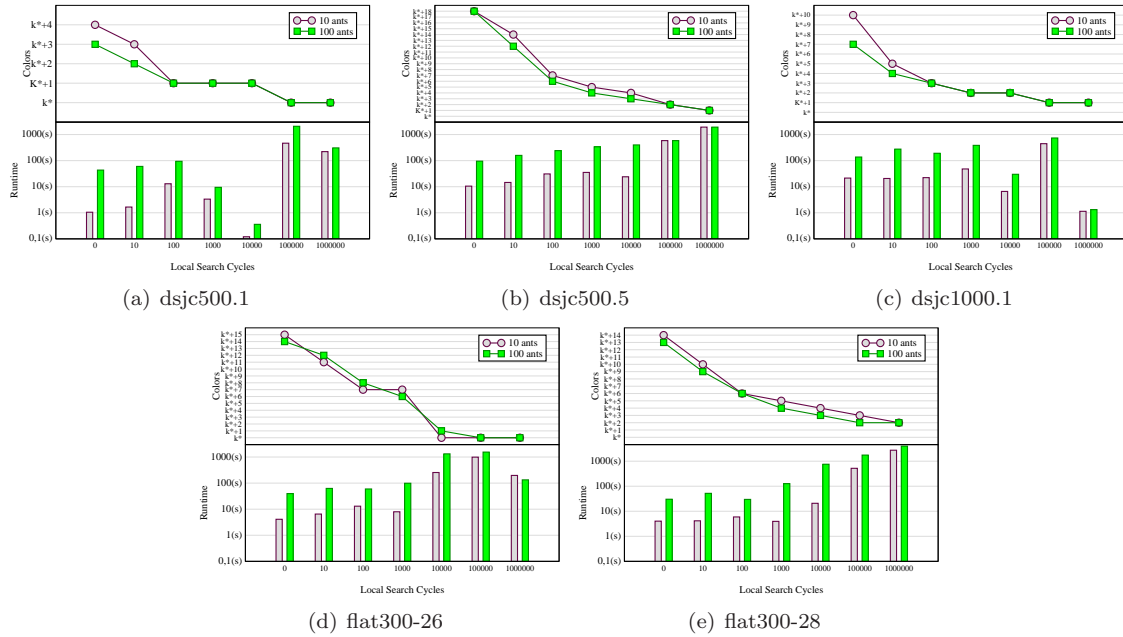


Figure 1: New results obtained by *ColorAnt₃-RT*

These results indicate that the colony size influences the CPU time, and the number of local search influences both the quality of the results and the successful attempts in terms of coloring. Therefore, it is not a good choice to use only ant colony. At least in the context of *ColorAnt-RT*, the performance gain is related to use an algorithm that combines a constructive strategy with an improvement one.

The best configuration is one that has a small colony, but uses a considerable amount of local search cycles, namely 100000. It is important to note that even though it seems a huge number, *ColorAnt-RT* does not use necessarily all these cycles, due to the validation of results in each cycle.

The *ColorAnt-RT* was not able to find the best known coloring to some instances. In some cases searching for a better solution leads to a termination of the algorithm, because it exceeded the maximum time or it does not converge to a better solution. This fact evidences that it is necessary to propose new strategies to *ColorAnt-RT* algorithms, even though in the context of register allocation *ColorAnt₃-RT* is a promising algorithm.

3.4 Summary

The *ColorAnts-RT* algorithms became a good choice for solving the k-GCP.

Analysing some characteristics of the graph instances, we could understand something about what these influence the performance of the *ColorAnts-RT* algorithms. An important perception is that some graph instances, in which randomness was used in its creation, tends to be more difficult to the algorithms find good solutions in an acceptable CPU time. If there is no randomness, the *ColorAnts-RT* tends to find better solutions.

The *ColorAnt₃-RT* is the best choice among the *ColorAnts-RT*.

4 Application

The goal of register allocation [28] is to allocate an unbounded number of program values to a finite number of machine registers, a problem that can be mapped as a GCP.

To solve this problem several works [52, 64, 71, 75] proposed the use of metaheuristics, more specifically, evolutionary algorithms. This paper proposes a different approach to solve the register allocation problem: a new algorithm for intraprocedural register allocation called **CARTRA**, an algorithm that extends a classic graph coloring register allocator (**IRA**) based on ACO.

4.1 The Literature

Chaitin *et al.* proposed a graph coloring register allocator (GCRA) [14, 15], an allocator used by the IBM 370 PL/I compiler. Currently, mainstream compilers uses an allocator derived from its. Subsequently, several works added improvements to Chaitin's allocator [7, 9]. Briggs *et al.* developed the most successful design for GCRA [13]. Their work redesigned the Chaitin *et al.* allocator to delay spill decisions, until later on in the allocation process. Runeson and Nyström proposed a generalization of Chaitin's allocator, which allows it to be used for irregular architectures [63]. This work is an interesting framework for a retargetable graph-coloring allocator.

George and Appel [2, 32] designed a GCRA that interleaves Chaitin-style simplification steps with Briggs-style conservative coalescing. They ensure that this approach eliminates more move instructions than Briggs's register allocator, while still guaranteeing not to introduce spills.

Daveou *et al.* [22] presented a register allocation framework designed to address the embedded processor specifications, such as a smaller number of registers, irregular and constrained register sets, and instructions operating on short or long data types. This allocator is based on Briggs's one, with two new components developed to improve performance, namely: a spill manager that optimizes spill operations, and a code manager that optimizes the move operations inserted by the allocator.

A problem with some GCRA approaches is the fact that some of them apply simple heuristic methods, resulting often in a poor allocation. In this case, there will be constant data traffic between the processor and the memory, causing a performance loss. To address this issue, several works proposed the use of metaheuristics with the goal of using a more aggressive strategy of graph coloring [52, 64, 71, 75].

Mahajan and Ali [52] developed a heuristic algorithm for GCRA for embedded processors, based on hybrid evolutionary algorithm that uses a new crossover operator and a new local search. The assumption of the authors is that traditional register allocators are developed to homogenous register set, but embedded processors need special attention due to its irregularities.

The work of Shamizi and Lotfi [64], and Topcuoglu *et al.* [71] also developed a register allocator based on hybrid evolutionary algorithm. Similar to work of Mahajan and Ali, these works also proposed crossover operators, and the use of local search. And Wu and Li [75] proposed a hybrid metaheuristic algorithm for GCRA that combines several ideas from classic GCRA algorithms, besides evolutionary algorithms [5] and Tabu Search [10, 38]. The main idea of this approach is to exploit the interplay between intensification and diversification of the solution space. The authors argue that it is a good solution to prevent searching processes from cycling, i.e., from endlessly revisiting the same solutions set, besides it can impart additional robustness to the search.

4.2 The Iterated Register Coalescing Allocator

Based on the observation that a good GCRA should not only assign different colors to interfering program values, but also trying to assign the same color to temporaries related by copies, George and Appel developed the Iterated Register Coalescing Allocator (**IRA**) [2, 32]. This algorithm iterates until there are no spills. The results show how to interleave coloring reductions with coalescing heuristic, leading to an algorithm that is safe and aggressive. The assumption in this approach is that the compiler is free to generate new temporaries and copies, because almost all copies will be coalesced. Figure 2 shows the phases of this register allocator, besides its organization.

The goal of each phase is as follows.

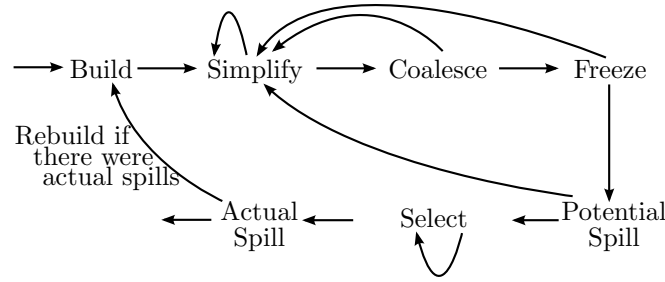


Figure 2: The Iterative Register Allocation Algorithm [2].

Build This phase constructs the interference graph using dataflow analysis, which nodes are categorized as either related or not related to moves. A move instruction means that the node is either the source or the destination of that move.

Simplify IRA uses a simple heuristic to simplify the graph. If the graph G contains a node n with less than k (number of registers) neighbors, then G' is built by doing $G' = G - \{n\}$. Then, if G' can be colored with k colors, G can be, as well. This phase repeatedly removes the non-move-related nodes from the graph if they have a low degree ($< k$), by pushing them on a stack.

Coalesce This phase tries to find moves to coalesce in the reduced graph obtained in *Simplify* phase. If two temporaries $T1$ and $T2$ do not interfere, it is desirable that these temporaries are allocated into the same register. This phase eliminates all possible move instructions by coalescing source and destination into a new node. If it is possible, this phase also removes the redundant instruction from the target program. *Simplify* and *Coalesce* phases are repeated while the graph contains non-move-related nodes or nodes of low degree.

Freeze Sometimes, neither *Simplify* nor *Coalesce* can be applied. In this case, the algorithm *freezes* a move-instruction node of low degree by considering it a non-move-related, and enabling more simplification. After this, *Simplify* and *Coalesce* are resumed.

Potential Spill If the graph, at some point, has only nodes of degree $\geq k$, these nodes are marked for spilling (they probably will be represented in memory). At this point, they are just removed from the graph and pushed on the stack.

Select *Select* remove the nodes from the stack, and tries to color them by rebuilding the original graph. This process does not guarantee that the graph will be k -colorable. If the adjacent nodes were already colored with k colors, the current node cannot be colored and will be an *actual spill*. This process will continue until there are not nodes in the stack.

Actual Spill In case of *Select* phase identifies an *actual spill*, the program is rewritten to fetch the spilled node from memory before each use, and store it after each definition. Now, the algorithm needs to be repeated on this new program.

4.3 The *ColorAnt₃-RT* Register Allocator

CARTRA modifies IRA in order to add an ACO metaheuristic phase. Two modifications were made, namely:

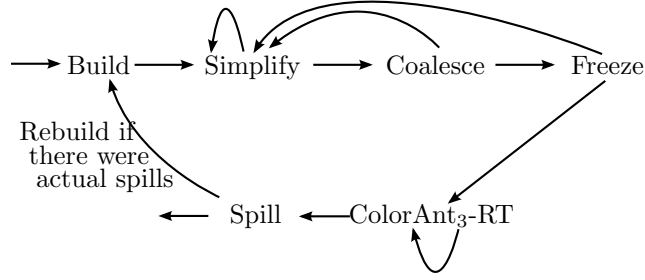
1. The *Select* phase was substituted by *ColorAnt₃-RT* algorithm, now it is a more aggressive phase than IRA's optimistic coloring; and
2. The strategy used for selecting spill is not based on node degree, but based on conflicting vertices.

Figure 3 shows the phases of the CARTRA algorithm.

Firstly, the IRA's classic phases construct an interference graph and reduces the graph. After, the *ColorAnt₃-RT* algorithm colors the interference graph. Finally, the new *Spill* phase selects an appropriate node to be represented in memory. These two modifications are as follows.

The ColorAnt₃-RT Phase The proposed approach is to use a heuristic algorithm based on artificial colonies of ants with local search designed for the GCRA problem, which can reduce the amount of spills.

The Spill Phase George and Appel showed that the Briggs *et al.* conservative coalescing criteria could be relaxed to allow more aggressive coalescing without introducing extra spilling. Besides, they describe an algorithm that preserves coalesced nodes founded before the potential spill was discovered. CARTRA uses

Figure 3: The Iterative Register Allocation with *ColorAnt₃-RT*.

the same strategy for coalescing, but a different approach to choose the nodes in the graph that will be represented in memory.

In IRA's algorithm, if there is no opportunity for *Simplify* or *Freeze*, the node will be spilled. In this case, the *Potential Spill* phase will calculate spill priorities for each vertice, as follows:

$$P_n = \frac{(uses_{out} + defs_{out}) + 10 \times (uses_{in} + defs_{in})}{degree} \quad (6)$$

where $uses_{out}$ is the set of temporaries that the node uses outside a loop, $defs_{out}$ is the set of temporaries that it defines outside a loop, $uses_{in}$ is the set of temporaries that it uses within a loop, $defs_{in}$ is the set of temporaries that it defines within a loop, and $degree$ is the number of edges incident to the node.

The node that has the lowest priority will be select to be spilled first. IRA's approach is an optimistic approximation: the node removed from the graph does not interfere with any of the others nodes in the graph.

CARTRA uses a different approach to select a spill node. Since the resulting graph given by *ColorAnt₃-RT* phase may have conflicting edges, the *Spill* phase selects the node with more frequency in the set of conflicting ones. In other words, considering each color c , the node colored with c , which has the biggest number of incident conflicting vertices, is removed from the graph and considered as an *actual spill*. If there is *actual spill* the program will be rewritten as IRA's algorithm, and a new iteration will take place. Therefore, the algorithm finishes when there are no more conflicting vertices in the graph.

4.4 The Performance of CARTRA

To analyze the performance of CARTRA several experiments was conducted. Such experiments was based on a compiler research framework that implements IRA [2, 32], and generates code to Intel's IA32 architecture. The compilers were executed in an Intel Xeon E5504 of 2.00 GHz, 24GB RAM running Ubuntu with kernel 3.2.0-24-generic.

4.4.1 Methodology

The benchmark used in the experiments consists of fifteen programs from SNU-RT [36], that are outlines on Table 5. For each program, we run the allocators ten times to measure the performance.

The programs, outline in Table 5, consist in general of small interference graphs. There are some exceptions, but in these cases the interference graphs have low density.

In general, the best results of an ACO algorithm are obtained after calibrating it for each instance that will be evaluated. This task is impractical in register allocation context because the characteristics of the interference graph change dynamically. CARTRA was not calibrated, due to this fact.

A strategy to calibrate CARTRA was to use the same parameters that calibrate the instances based on register allocation, such as FPSOL, INITX, MULSOL, or ZEROIN, except the number of ants which was based on the results of the Section 3.3. Therefore, the parameters of CARTRA are $nants = 10$, $\alpha = 1.0$, $\beta = 1.0$, $\rho = 0.0$, $max_cycles = 50$, and $tabu_search_cycles = 50$. CARTRA's coloring phase (*ColorAnt₃-RT*) stops if there is no improvement in reducing the number of conflicting edges for more than $max_cycles/4$.

In a register allocation context several questions need to be discussed, such as:

- Does the new register allocator decrease the number of spills?
- Is the new register allocator fast?

Table 5: Programs

Name (Data size)	Characteristics			
	Function	Interference Graph		
		$ V $	$ E $	D
Adpcm (16khz sample rate data) (test_data[2000])	rad	50	376	0.31
	encode	396	3632	0.05
	decode	401	4105	0.05
	reset	2457	16911	0.01
	filtez	53	485	0.35
	filtep	21	130	0.62
	quantl	55	466	0.31
	logscl	31	185	0.40
	scalel	30	183	0.42
	upzero	149	1518	0.14
	uppol1	31	204	0.44
	uppol2	38	265	0.38
	invqah	22	130	0.56
	logsch	30	179	0.41
	sin	51	391	0.31
	main	180	2302	0.14
Binary Search (15×10^7)	bs	60	581	0.33
	main	173	1029	0.07
FFT (1024 elements)	sin	50	382	0.31
	fft	306	5112	0.11
	main	38	288	0.41
FFT Complex (1024 elements)	sin	49	370	0.31
	init_w	50	395	0.32
	fft	138	2303	0.24
	main	52	377	0.28
Fibonacci 30_i element	fib	20	120	0.63
	main	11	40	0.73
FIR 35 points	sin	52	461	0.35
	sqrt	40	376	0.48
	fir_filter	53	498	0.36
	gaussian	53	455	0.33
	main	585	7456	0.04
Insertion Sort (10^8)	main	149	1285	0.12
Jfdctint (64 elements)	fdct	652	9555	0.05
	main	28	186	0.49
LMS (64 sine wave length)	sqrt	40	376	0.48
	sin	52	461	0.35
	gaussian	53	455	0.33
	lms	145	2076	0.20
	main	70	956	0.40
Matmul $A[5 \times 10^7][5 \times 10^7]$ $B[5 \times 10^7][5 \times 10^7]$	alloc	56	559	0.36
	matmul	120	1709	0.24
	main	83	919	0.27
Minver $A[3 \times 10^6][3 \times 10^6]$	alloc	82	669	0.20
	mmul	104	1375	0.26
	minver	496	9464	0.08
	main	250	2456	0.08
Quick Sort 19×10^7	sort	462	5197	0.05
	main	163	1101	0.08
Qurt 2×10^7	sqrt	38	329	0.47
	qurt	254	2081	0.06
	main	95	603	0.14
Select 2×10^8	select	416	5339	0.06
	main	154	1041	0.09
Sqrt $N = 1234$	sqrt	38	329	0.47
	main	11	40	0.73

- What is the impact of the new register allocator on code quality, in terms of runtime, code size, and memory hierarchy accesses?
- Is there any tradeoff in using the new register allocator?

4.4.2 Spill and Fetch

The implementation of both algorithms attempts to minimize the number of spills (the values relegated to memory), and therefore the number of fetches (the loads necessary to fetch the spills). The number of spills will influence on the several aspects of the code quality, such as code size, memory hierarchy accesses, and runtime.

The Figure 4 shows the number of spills for each program compiled for both allocators. This figure presents four bars for each program. The first bar represents the best case of **CARTRA**, in other words the least number of spills. The second represents the average number of spills. The third represents the largest number of spills. And the last represents the number of spills obtained by **IRA**. Note that, **IRA** is a deterministic algorithm, therefore, it generates the same final code in each execution. On the other hand, **CARTRA** is a heuristic algorithm. Consequently, **CARTRA** can generate in each execution a different final code. The Figure 4 shows three bars for each program compiled using **CARTRA**, due to this fact.

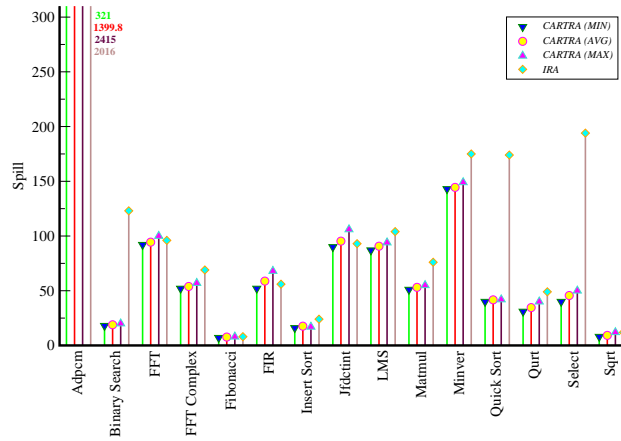


Figure 4: The number of spills.

As it can be seen in Figure 4, **CARTRA** outperforms **IRA**. **CARTRA** tends to spill fewer temporaries than **IRA**, because the former tries to find the best approach to color the graph, consequently the number of conflicting edges is zero. In this case, the allocator uses few registers per function. **CARTRA** minimizes the function cost by reducing the number of memory access instructions, instructions that typically have a higher cost when compared to other instructions classes. **CARTRA** spills few temporaries and uses few registers in the allocation, therefore, it finds more opportunities for coalescing.

In the worst case (third bar) **CARTRA** outperforms **IRA** in almost all programs, except for ADPCM, FFT, and JFDCTINT. It situation changes for the average case (second bar), and also for the best case (first bar). In the average **CARTRA** do not outperform **IRA** for FIR and JFDCTINT. However, in the best case **CARTRA** outperforms **IRA** in all programs.

The best choice is to execute **CARTRA** several times because it can improve the results (decrease the number of spills) up to 50%. It was the case of SQRT, in the worst case **CARTRA** generate for this program 12 spills, and in the best case only 8 spills. Executing **CARTRA** 10 times indicated that this number of times was a good choice to generate good results.

In average, **CARTRA** is decreases the number of spills from 1,65% (ADPCM) to 44,02% (adpcm), excluding two cases: (1) the case in which **CARTRA** increases the number of spills (FIR, and JFDCTINT); and (2) the case in which the improvement is upper than 300% (BINARY SEARCH, QUICK SORT, and SELECT). These two cases deserve a detailed analysis. The Table 6 shows the detail results for the instances: FIR, JFDCTINT, BINARY SEARCH, QUICK SORT, and SELECT.

The performance loss of FIR, and JFDCTINT is due the functions *fir_filter*, and *fdct*, respectively. In both cases, **CARTRA** was not able to choose the best nodes for spilling. Note that **CARTRA** tries to color the interference graph based on probabilities. Besides, heuristic algorithms are based on pseudo-random numbers, and it is not certain that these algorithms will find a property solution in each execution. Consequently, some executions can generate poor results.

Table 6: Results obtained by **CARTRA** and **IRA**. The runtime (R) presents the best CPU time, the average CPU time, and the worst CPU time for the program.

Program		CARTRA								R	IRA		
Name	Function	Spill/Fetch									Spill	Fetch	R
		Min		Average		Max		S. Deviation					
		Spill	Fetch	Spill	Fetch	Spill	Fetch	Spill	Fetch				
FIR	sin	11	22	11.22	21.44	12	21	1.86	3.36	1.10s 1.11s 1.12s	11	22	1.21s 1.22s 1.23s
	sqrt	9	11	9	11.44	9	13	0.0	0.88		12	19	
	fir_filter	17	25	18.89	26.56	27	34	4.51	4.45		12	18	
	gaussian	6	11	9	12.56	10	13	1.62	2.40		12	17	
	main	9	48	9.89	49.54	10	50	0.60	12.61		9	47	
	Total	52	117	58.78	127.44	68	131	5.26	10.93		56	123	
Jfcdctint	fdct	83	191	88.70	181.90	101	198	6.85	8.46	7.87s	79	157	7.09s
	main	7	7	6.60	7.20	5	7	0.84	0.42	7.93s	14	15	7.14s
	Total	90	198	95.30	189.10	106	205	6.72	8.28	8.00s	93	172	7.20s
Binary Search	bs	15	16	16.90	16.10	17	17	0.88	0.88	0.59s	28	27	0.73s
	main	3	3	3	3	3	3	0.00	0.00	0.61s	95	126	0.75s
	Total	18	19	19.90	19.10	20	20	0.88	0.88	0.62s	123	153	0.76s
Quick Sort	sort	38	100	39.70	100.70	40	103	1.34	1.70	3.98s	50	116	4.30s
	main	2	2	2	2	2	2	0.00	0.00	4.01s	124	165	4.33s
	Total	40	102	41.70	102.70	42	105	1.34	1.70	4.05s	174	281	4.37s
Select	select	38	77	43.60	83.90	48	88	5.40	4.75	8.27s	70	104	8.87s
	main	2	2	2	2	2	2	0.00	0.00	8.44s	124	165	8.91s
	Total	40	79	45.60	85.90	50	90	5.40	4.75	8.61s	194	269	8.97s

Though *Fir_filter* and *fdct* have low density, in these cases the decisions made by **CARTRA** occasioned a low convergency. For these functions, the allocator needed to execute several times to obtained an interference graph which all nodes could be mapped to machine registers.

The performance loss of **BINARY SEARCH**, **QUICK SORT**, and **SELECT** when executed by **IRA** is due the function *main*. The results presented in Table 6 show that **IRA** generated a high number of spills (and fetches) for this function. Consequently, **CARTRA** obtained an excellent performance over **IRA**. The **IRA** performance loss occurred due to the calibration of each program to increase the register pressure. It was done to evaluate both allocators under high register pressure. Each program that had values defined like this:

```
struct DATA data[15] = {{1,100}, ..., {18,10}};
```

was transformed in:

```
struct DATA data[15];
```

```
void main() {
...
data[0].key = 1;
data[0].value = 100;
...
data[14].key = 18;
data[14].value = 10;
...
}
```

Besides, code like:

```
void main() {
...
binary_search(3);
...
}
```

was also transformed in:

```
void main() {
int a = 3;
...
binary_search(a);
...
}
```

These transformations occasioned a high performance loss for these three programs in *IRA*. In fact, the same problem occurred for the function *reset* of the program *ADPCM* (in average *CARTRA* outperforms *IRA* in 1784,69% for this function). In this case, the problem was minimized because *IRA* outperforms *CARTRA* in nine functions of this program. Consequently, in average *CARTRA* outperforms *IRA* in 62,87% for *ADPCM*.

The results for these three programs (*BINARY SEARCH*, *QUICK SORT*, and *SELECT*) indicate that *CARTRA* handles a high register pressure, and it handles this situation better than *IRA*.

When the program need to be rewritten due to spill decision, each spilled node need to be fetched from memory before each use. For this reason, it is also important to investigate the number of fetches produced by the register allocator.

The Figure 5 shows the number of fetches for each program compiled for both allocators. This figure has the same configuration of the Figure 4. Consequently, Figure 5 also presents four bars for each program. The first bar represents the best case of *CARTRA*, in other words the least number of spills. The second represents the average number of spills. The third represents the largest number of spills, and the last represents the number of spills obtained by *IRA*.

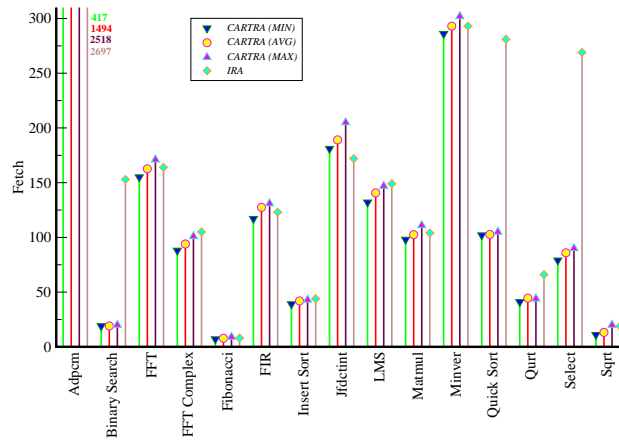


Figure 5: The number of fetches.

The results showed in Figure 5 indicate that both allocators have a different pattern for fetches (considering the number and not the performance loss). While for spills, only five programs have the number of spills upper than 100 (*ADPCM*, *FFT*, *JFDCTINT*, *LMS*, and *MINVER*), the number of fetches is upper than this mark for nine programs (*ADPCM*, *FFT*, *FFT COMPLEX*, *FIR*, *JFDCTINT*, *LMS*, *MATMUL*, and *QUICK SORT*). It is essential to note that the same programs belong to these two groups.

In the worst case (third bar) *CARTRA* outperforms *IRA* for half programs, namely: *ADPCM*, *BINARY SEARCH*, *FFT COMPLEX*, *INSERT SORT*, *LMS*, *QUICK SORT*, *QURT*, and *SELECT*. This situation changes for the average case (second bar), in which *CARTRA* is not able to outperform *IRA* only in two programs (*FIR*, and *JFDCTINT*). It also changes for the best case (first bar), in which *CARTRA* is not able to outperform *IRA* for *JFDCTINT*. This results show that concerning performance loss *CARTRA* has the same performance for the number of spills and fetches.

Besides, there is the same problem with the programs *BINARY SEARCH*, *QUICK SORT*, and *SELECT* in *IRA*, even thought in different proportion. Because in *BINARY SEARCH* the improvement is upper than 700% for *IRA*, and for the other two programs this improvement is lesser than 300%. *CARTRA* decreases in average the number of fetches from 0.82% to 80%, excluding these three programs, and that in which *CARTRA* does not outperform *IRA*.

We can conclude based on these results that *CARTRA* is a good option to decrease the constant data traffic between the processor and memory.

4.4.3 Compile Time

Other aspect that is important to analyze in the context of register allocation is the compile time. The Figure 6 shows the compile time of both allocators. This figure shows the compile time in logarithm scale because *CARTRA* compile time is greater than *IRA*.

IRA is faster than *CARTRA* from 2.58 (*FIBONACCI*) to 40.0 (*FFT*) times. It is a problem when the compile time should be address, for example, in dynamic systems. On the other hand, in a standalone compilation system, the compile time should not be a problem. These results also indicate the instability of *ACO* algorithm. In fact, the standard deviation ranges from 0.11 (*BINARY SEARCH*) to 5.24 (*QUICK SORT*). The increase of the graph density

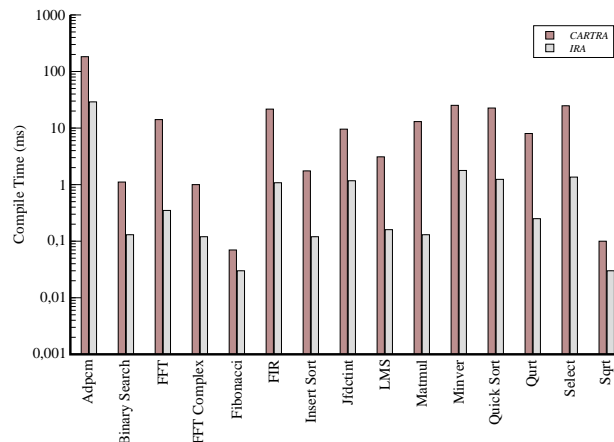


Figure 6: The compile time.

and/or the number of nodes tends to increase the compile time. In fact analyzing the Figure 6 and the Table 5 this tendency occurs in **CARTRA**.

A relatively high runtime is usually a problem in ACO algorithms. Although these algorithms are able to find satisfactory solutions for several problem, the runtime is a cost that must be paid. Consequently, many researchers use different approaches avoiding ACO algorithms.

There is a tradeoff in using **CARTRA**, compile time versus code quality. **CARTRA** has a high runtime when compared with **IRA**, but the code quality in terms of the number of spill is better than that generated by **IRA**.

It is essential to note that the reduction of the number of spill tends to eliminates clock cycles and assembly instructions, which impacts the runtime and code size. These issues can not be so critical in desktop applications, but it is highly significant in other contexts, such as wireless sensor networks (WSN) [1, 37, 41].

An important issue in the WSN context is the energy consumption, due to sensor nodes are particularly simple in terms of their components.

WSN usually consists of a microcontroller with limited computational power, limited memory storage, among other components. Minimizing clock cycles and addressing the storage constraints have been a design goal for WSN, due to the limited computational power. Both goals can be addressed reducing the number of spills because this reduction decreases the number of clock cycles, and also the number of assembly instructions. Consequently, this reduction can also deal with other issue, namely: energy consumption.

In WSN, energy consumption is a key factor for the network lifetime and accuracy of information. In this way, it is essential to develop techniques that are able to save energy.

In this context, it is of note that the energy dissipated by an application during data-processing depends on the compiler.

The efficiency of the compiler affects the instruction count and average cycles per instructions because the compiler determines the translation of the source language instructions into hardware instructions. The strategy used by the compiler for register allocation, therefore, can improve the application performance, in other words register allocator can save energy.

4.4.4 Convergence

Both algorithms are iterative, i.e., the register allocator ends when there are no spills (see Figures 2 and 3 – *rebuild if there were actual spills*).

The Table 7 shows the convergence for both allocators. The results showed for **CARTRA** is the average case, but the results for **IRA** are deterministic. For each interference graph is presented a list, which contains the number of spills in each iteration, and the list size.

CARTRA finds coloring that eliminates the number of spills in lesser iterations (rebuilding) than **IRA**. In general, **CARTRA** does not need more than three rounds to finish while **IRA** needs in many cases more than five rounds. Besides, some rounds do not minimize the number of spills, resulting in more iterations.

The approach based on *defs-uses* used by **IRA** causes a gradual decrease in the number of spills until this number reaches zero. On the other hand, the approach based on conflicts leads to a faster convergence than *defs-uses*.

Table 7: Convergence

Program		Allocator	
Name	Function	CARTRA	IRA
Adpcm	rad	[5,2,1,0](4)	[5,0](2)
	encode	[16,0](2)	[16,3,3,3,2,1,0](7)
	decode	[16,2,0](3)	[14,0](2)
	reset	[56,0](2)	[1046,294,60,60,0](5)
	filtez	[8,0](2)	[8,2,1,1,0](5)
	filtep	[3,1,0](3)	[3,1,1,0](4)
	qunt1	[8,3,0](3)	[8,1,0](3)
	logsc1	[3,0](2)	[3,0](2)
	scale1	[3,0](2)	[4,1,1,1,2,0](6)
	upzero	[4,3,2,2,2,2,1,1,1,1,1,0](13)	[10,5,5,4,3,4,1,0](9)
	uppol1	[9,5,1,1,1,0](6)	[3,2,1,0](4)
	uppol2	[3,0](2)	[4,5,3,1,0](5)
	invqah	[3,1,1,0](4)	[3,0](2)
	logsch	[3,0](2)	[3,0](2)
	sin	[5,0](2)	[5,0](2)
	main	[15,0](2)	[15,3,2,2,1,1,0](7)
Binary Search	bs	[9,0](2)	[9,3,1,1,1,2,1,1,0](10)
	main	[3,0](2)	[33,31,16,0](4)
FFT	sin	[5,0](2)	[5,0](2)
	fft	[38,2,0](3)	[32,2,1,1,0](5)
	main	[9,0](2)	[8,3,2,0](4)
FFT Complex	sin	[5,0](2)	[5,0](2)
	init_w	[5,0](2)	[6,2,6,1,4,0](6)
	fft	[22,0](2)	[18,3,0](3)
	main	[5,0](2)	[5,0](2)
Fibonacci	fib	[2,1,0](3)	[2,1,1,0](4)
	main	[3,0](2)	[3,0](1)
FIR	sin	[5,0](2)	[5,0](2)
	sqrt	[7,0](2)	[7,0](4)
	fir_filter	[8,2,0](3)	[8,1,0](3)
	gaussian	[5,0](2)	[6,0](7)
	main	[9,0](2)	[8,0](2)
Insertion	main	[11,0](2)	[10,2,3,1,0](5)
Jfdctint	fdct	[36,2,0](3)	[23,0](2)
	main	[5,0](2)	[6,3,1,1,0](5)
LMS	sqrt	[7,0](2)	[7,0](2)
	sin	[5,0](2)	[5,0](2)
	gaussian	[6,1,0](3)	[6,0](2)
	lms	[22,2,2,0](4)	[21,2,3,2,2,1,0](7)
	main	[20,1,0](3)	[19,2,1,1,0](5)
Matmul	alloc	[7,2,0](3)	[9,2,2,1,2,4,0](7)
	matmul	[18,4,0](3)	[18,0](2)
	main	[10,3,1,0](4)	[11,2,4,3,7,1,0](7)
Minver	alloc	[5,0](2)	[4,0](2)
	mmul	[15,3,3,0](4)	[17,0](2)
	minver	[38,12,3,1,0](5)	[39,8,8,8,7,0](6)
	main	[14,6,2,0](4)	[14,6,4,4,6,4,0](8)
Quick Sort	sort	[15,1,1,1,0](5)	[15,2,2,2,2,2,0](8)
	main	[2,0](2)	[23,21,20,20,20,0](6)
Qurt	sqrt	[7,0](2)	[7,0](2)
	qurt	[12,2,1,0](4)	[12,3,4,3,2,0](6)
	main	[4,0](2)	[4,0](2)
Select	select	[16,1,1,0](4)	[19,1,6,4,5,4,3,2,0](9)
	main	[2,0](2)	[23,21,20,20,20,0](6)
Sqrt	sqrt	[7,0](2)	[7,0](2)
	main	[0](1)	[0](1)

4.4.5 Runtime

It is essential to evaluate the impact of the reduction of the traffic between the processor and memory in runtime. As a result of the number of spills and fetches reduction, it is expected that there is a reduction in the program runtime.

The Figure 7 shows the runtime obtained by each program compiled using both allocators. This figure also presents four bars for each program. The first bar represents the runtime of the best code generated by **CARTRA**, in other words the runtime of the code that has the least number of spills and fetches. The second represents the average runtime. The third represents the runtime of the worst code. The last represents the runtime of the code generated by **IRA**. The runtime is the average of 10 executions. The average runtime (second bar) is the average

of 100 executions because there are 10 different final codes generated by **CARTRA**. Besides, the runtime is showed in logarithm scale because the programs have short runtime.

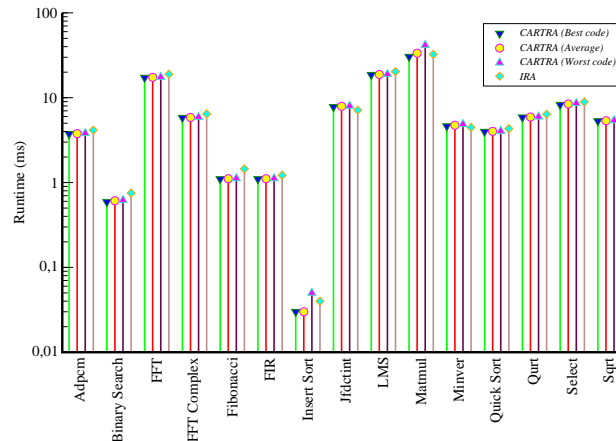


Figure 7: The runtime.

The worst final code generated by **CARTRA** is not able to outperform **IRA** in five programs, namely: INSERT, FIR, JFDCTINT, MATMUL, and MINVER. For these programs **IRA** outperforms **CARTRA** in 9.76%, 11.11%, 13%, 28%, and 4.01%, respectively. The worst final code outperforms **IRA** from 4.04% (SELECT) to 23.37% (FIBONACCI).

The average code has the same performance than the worst final code for JFDCTINT, decreases the performance gain for MATMUL in 25.14% (the runtime of the program generated by **IRA** outperforms **CARTRA** in 2.86%) and in 3% for **IRA**, improves the performance of **CARTRA** for INSERT in 22.16% over **IRA**, and outperforms **IRA** in a performance gain that ranges from 5.28% (SELECT) to 23.94% (FIBONACCI). The average code increases the performance gain for MINVER in 2.14% for **IRA** over **CARTRA**.

The best final code do not outperform **IRA** for JFDCTINT, and MINVER. For these programs, the performance gap is as similar as that achieved by average code. Using this code, the performance gain ranges from 6.76% SELECT to 28.21% (INSERT).

These results are consistent with the results showed in previous sections, except for MINVER. They corroborate the need of executing the compiler several times to improve the performance, but until now they do not explain the MINVER performance loss.

CARTRA decreases the number of spills generated by **IRA** in 20%, but **CARTRA** does not decrease the number of fetches (for an average case). These results would suggest that, in the worst case, the runtime obtained by **CARTRA** and **IRA** were the same. There can be a side effect of using a heuristic algorithm besides choosing spill based on the number of conflicting edges. However, this situation is an exception. A wrong decision in which node should be spilled can generate a final code that does not achieve a good performance for the cache hierarchy of the underlying hardware.

4.4.6 Code Size

The Figure 8 shows the code size obtained by both allocators. This figure has the same configuration of Figure 7. The first bar represents the code size of the best final code. The second represents the average code size. The third represents the code size the worst final code, and the last represents the code size of the final code generated by **IRA**.

The codes generated by **CARTRA** have similar performance to **IRA**. From the worst final code to the best final code the improvement gain does not have a growth up to 0.69% (LMS). This growth is negligible in the used context (desktop) because, it corresponds a reduction of only 136 bytes.

Based on code size reduction obtained by **CARTRA** the programs can be classified in two groups, namely: similar performance, and performance gain.

The programs that belong to similar performance are FFT, FFT COMPLEX, FIBONACCI, FIR, INSERT, LMS, MATMUL, MINVER, QURT, and SQRT. In this case the performance lost (or gain) of **CARTRA** ranges from -0.90% to 0.50% that is also a negligible range in a desktop context. The programs that belong to the second group are JFDCTINT, ADPCM, BINARY SEARCH, QUICK SORT, and SELECT. For these programs, the **CARTRA** performance gain over **IRA** is 26.45%, 26.34%, 18.07%, 4.59%, and 4.26%, respectively in the average case. **CARTRA**, therefore, decreases in some cases the code size.

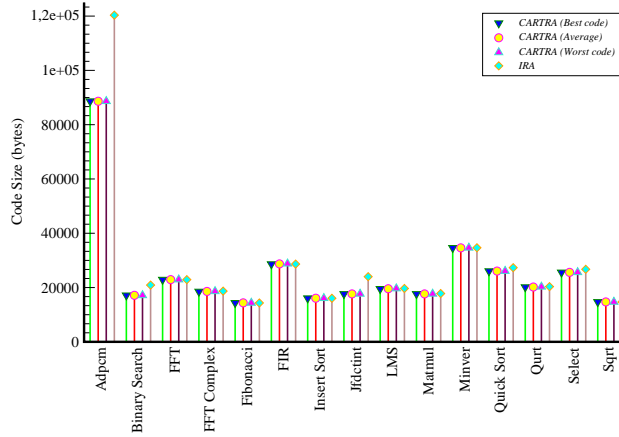


Figure 8: The code size.

4.4.7 Another Register Allocator

An important research is to investigate a different strategy to color the interference graph that is able to reduce the compile time. Although, the results with **CARTRA** demonstrated that it is able to reduce the amount of spills, we need to pay the price of a high compile time.

For this purpose, we developed the Hybrid Evolutionary Coloring Register Allocator (**HECRA**). Like **CARTRA**, **HECRA** modifies **IRA** in order to add a metaheuristic phase. While **CARTRA** is based on ACO algorithm, **HECRA** is based on Hybrid Evolutionary Algorithm (HCA) [1]. Both allocators use the same strategy to select spills. Therefore, they have the same structure but with a different coloring phase, while **HECRA** uses HCA as coloring phase, **CARTRA** uses *ColorAnt₃-RT*.

The HCA phase begins with a population and an iterative process is repeated for a number of generations. In the members of the population (*parents*) a crossover operator is applied to generate a new configuration (*child*), in which a local search method will be applied to improve it. The HCA is described in details in [30].

A difference between **CARTRA** and **HECRA** is in the use of local search. The former use a reactive scheme, and the later a dynamic scheme.

Results We conducted a series of experiments to evaluate **HECRA**. To perform such experiments, we add **HECRA** in a research compiler framework, as we implemented **CARTRA**. It compiler framework generates code to Intel's IA32, and was executed in a Intel Xeon E5504 of 2.00 GHz, 24GB RAM running Ubuntu with kernel 3.2.0-24-generic.

The benchmark consists of eleven programs from SNU-RT [36]. For each program, we run the allocators ten times to measure the performance. The parameters of **CARTRA** are as described in Section 4.4.7. And, the parameters of **HECRA** are: $p = 10$, $L = 2000$, $max_cycles = 50$, $diversity = 20$, and the tabu search was limited by a maximum of 2000 cycles.

We conduct several experiments to measure the performance of our algorithm. The experiments have the following goals: (1) measure the number of spills; (2) analyse the compilation time; and (3) analyse the convergence.

Spill and Fetch The implementation of both allocators attempts to minimize the number of spills. As it can be seen in Table 8, our allocators outperform **IRA**. Our proposed allocators tend to spill less temporaries, because they try to find the best approach to color the graph, so that the number of conflicting vertices is zero. In this case, they are able to use less registers per function. They minimize the function cost by reducing the number of memory access instructions, instructions that typically have a high cost when compared to other instruction classes. Also, because our allocators tend to spill few temporaries and use few registers in the allocation, they are able to find more opportunities for coalescing.

In ten applications, our allocators achieve reductions from 0% to 85.58% on number of spills in the average, when they are compared with **IRA**. Only for one application the **IRA** obtained better results, namely: *Jfdctint*. Besides, our allocators achieve reductions from 3.64% to 86.27% on number of fetches. However, for fetches, the **IRA** obtained best results for *FIR* and *Jfdctint*. In summary, only for one application our allocators did not achieve a better performance than **IRA**. It demonstrated that the strategy for coloring the interference graph and selecting spill, used by our allocators are a better approach to minimize the number of spill.

In some cases, **CARTRA** is able to get better results than **HECRA** and **IRA**. On the other hand, it is necessary to run our allocators several times to get the best result. This does not occur with the **IRA**, because it has no

Table 8: Results obtained by HECRA, CARTRA and IRA.

Name	HECRA						CARTRA						IRA	
	Min		Average		Max		Min		Average		Max		Spill	Fetch
	Spill	Fetch	Spill	Fetch	Spill	Fetch	Spill	Fetch	Spill	Fetch	Spill	Fetch		
Binary	17	18	18.3	19.5	20	21	18	19	19	19.5	20	20	123	153
FFT	53	86	56.1	84.6	63	103	52	88	54.5	94.5	57	101	96	164
Fibonacci	4	3	5.5	4.3	9	8	7	7	7.5	8	8	9	8	8
FIR	43	97	54.3	144.4	73	175	52	117	53	135	54	153	56	123
Insert	13	32	15.6	35.0	19	38	16	36	16.5	39.5	17	43	24	44
Jfdctint	91	181	98.5	193.1	110	202	90	181	98	193	106	205	93	172
LMS	75	121	108.2	156.4	293	343	87	132	90.5	139.5	94	147	104	149
Quick	39	103	43.7	105.4	48	109	40	102	41	103.5	42	105	174	281
Qurt	27	40	31.3	44.0	36	52	31	41	35.5	42.5	40	44	49	66
Select	42	85	48.1	90.6	56	98	40	79	45	84.5	50	90	194	269
Sqrt	8	10	9.7	14.0	12	20	8	11	10	15.5	12	20	12	19

random feature like HECRA and CARTRA. In other words, IRA is deterministic, while HECRA and CARTRA provides a different solution for each run (nondeterministic). The ideal is to run our allocators as many times as possible to ensure that good results will be obtained.

The analysis of the interference graphs does not give some insight about the performance of our allocators. Neither the number of vertices nor the number of edges influenced the performance, except for Binary Search. All benchmarks spill some temporaries. Besides, the number of store instructions is almost equal to the number of fetch instructions, suggesting that the vertices that have been spilled may have few definitions and uses.

Compile Time Table 9 shows the compilation time of all allocators. IRA is faster than CARTRA from 5.43 to 606.52 times. IRA is also faster than HECRA, but in this case from 0.19 to 3.08 times. In the context which compilation time should be address our allocators can be a problem, for example, in dynamic systems. On the other hand, in a standalone compilation system, the compilation time should not be a problem.

Table 9: Compile Time

Program	HECRA		CARTRA		IRA	
	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation
Binary Search	0.272	0.005	26.059	0.322	0.168	0.001
FFT	0.657	0.037	43.351	4.461	0.213	0.008
Fibonacci	0.087	0.020	2.431	0.483	0.448	0.004
FIR	1.421	0.034	235.817	18.697	1.317	0.053
Insert Sort	0.285	0.106	22.495	7.287	0.110	0.000
Jfdctint	1.705	0.360	634.870	386.634	1.402	0.045
LMS	0.810	0.089	84.079	11.072	0.358	0.002
Quick sort	1.326	0.247	327.153	242.891	1.180	0.000
Qurt	0.674	0.059	144.351	45.332	0.238	0.006
Select	1.327	0.079	363.318	180.627	1.510	0.065
Sqrt	0.079	0.000	3.413	0.019	0.047	0.008

These results also demonstrated the instability of ACO algorithm. Note that the standard deviation is very high. A relatively high runtime is usually a problem on ACO algorithms. Although these algorithms are able to find satisfactory solutions to many problems, the runtime is a cost that must be paid in some cases.

Note that CARTRA is able to reduce the number of spills. This reduction eliminates clock cycles and code size. Although CARTRA has a very high runtime, it is able to address several goals, such as: reduce code size, reduce the number of memory accesses, and consequently reduce the amount of energy needed.

It is very important to note that HECRA is able to address the goals that CARTRA addresses, besides minimizing the compilation time. HECRA is faster than CARTRA from 27.94 to 372.36 times. These results demonstrated that changing the strategy for coloring the interference graph, the new allocator was able to maintain the code quality but in a low compilation time.

Convergence The Table 10 shows the convergence in average. For each interference graph is presented a list containing the number of spills at each iteration and the list size.

The results demonstrated that both HECRA and CARTRA find a coloring that eliminates the number of spills in fewer iterations (rebuilding) than IRA. In general, the number of iterations required by IRA is greater than that required by our allocators.

Table 10: Convergence

Program		Allocator		
Name	Function	HECRA	CARTRA	IRA
Binary Search	bs	[9,0](2)	[9,0](2)	[9,3,1,1,1,2,1,1,1,0](10)
	main	[3,0](2)	[3,0](2)	[33,31,16,0](4)
FFT	sin	[5,0](2)	[5,0](2)	[5,0](2)
	init_w	[6,1,0](3)	[38,2,0](3)	[32,2,1,1,0](5)
	fft	[21,2,0](3)	[9,0](2)	[8,3,2,0](4)
	main	[4,2,0](3)	[9,0](3)	[8,3,2,0](5)
	fib	[2,1,0](3)	[2,1,0](3)	[2,1,1,0](4)
Fibonacci	main	[0](1)	[3,0](2)	[3,0](2)
	sin	[5,0](2)	[6,0](2)	[5,0](2)
FIR	sqrt	[7,0](2)	[8,1,0](3)	[7,0](2)
	fir_filter	[8,2,0](3)	[8,2,0](3)	[8,1,0](3)
	gaussian	[5,0](2)	[5,0](2)	[6,0](2)
	main	[7,0](2)	[9,0](2)	[8,0](2)
	main	[9,0](2)	[11,0](2)	[10,2,3,1,0](5)
Jfdctint	fdct	[39,0](2)	[3,2,0](3)	[23,0](2)
LMS	main	[5,2,1,1,1,1,0](7)	[5,0](2)	[6,3,1,1,0](5)
	sqrt	[7,0](2)	[7,0](2)	[7,0](2)
	sin	[5,0](2)	[5,0](2)	[5,0](2)
	gaussian	[5,0](2)	[6,1,0](3)	[6,0](2)
	lms	[23,4,1,0](4)	[22,2,2,0](4)	[21,2,3,2,2,1,0](7)
	main	[16,0](2)	[20,1,0](3)	[19,2,1,1,0](5)
Quick Sort	sort	[16,2,1,0](4)	[15,1,1,1,0](5)	[15,2,2,2,2,2,0](8)
	main	[2,0](2)	[2,0](2)	[23,21,20,20,20,0](6)
Qurt	sqrt	[7,0](2)	[7,0](2)	[7,0](2)
	qurt	[11,4,1,0](4)	[12,2,1,0](4)	[12,3,4,3,2,0](6)
	main	[2,0](2)	[4,0](2)	[4,0](5)
Select	select	[19,1,1,0](4)	[16,1,1,0](4)	[19,1,6,4,5,4,3,2,0](9)
	main	[2,0](2)	[2,0](2)	[23,21,20,20,20,0](6)
Sqrt	sqrt	[7,0](2)	[7,0](2)	[7,0](2)
	main	[0](1)	[0](1)	[0](1)

The approach based on *defs-uses* used by IRA causes a gradual decrease in the number of spills. On the other hand, the approach based on conflicts leads to a fast convergence.

HECRA performance is similar to CARTRA. They do not need more than three rounds to finish, while IRA needs in many cases more than five rounds. Besides, some rounds do not minimize the number of spills, resulting in more iterations.

4.4.8 Discussion

The experiments with CARTRA and the results obtained by this allocator can be summarized as follows.

The Traffic Between the Processor and Memory CARTRA is a good register allocator based on graph coloring that decreases the traffic between the processor and memory. In fact, CARTRA outperforms a traditional graph coloring register allocator: IRA. It is due to CARTRA maximizes the program values into machine registers. Consequently, CARTRA improves the final code runtime.

Calibration The good results are due to a good calibration of CARTRA. In fact, ACO algorithms depend on choices made to estimate the values of its parameters. Besides, the results are improved if the algorithm uses local search. In general, researches with ACO algorithms calibrate the parameters of each instance that will be evaluated. It is impractical in the register allocation context because the characteristics of the interference graph changes during the execution of the register allocator. The estimation of CARTRA's parameters was based on an experimental evaluation of traditional graph coloring instances.

Compile time In CARTRA there is a tradeoff: compile time versus code quality. Although, CARTRA be able to minimize the number of spills and fetches, there is the cost of a high compile time. CARTRA is not a choice for dynamic systems, but for standalone compiler CARTRA improves the compiler performance. This tradeoff indicates that CARTRA is a good choice in contexts which compile time is not a concern.

Runtime As a result of minimizing the number of spills and fetches, it is expected that the program has a reduction in its runtime. The results obtained by CARTRA indicate that it occurs.

Convergence CARTRA is based on the framework used by IRA, as a result, CARTRA is also an iterative register allocation, but due to the strategy used for coloring the interference graph, CARTRA needs less iterations than IRA. The aggressive strategy used by CARTRA, an ACO algorithm, was an excellent strategy to improve the framework performance.

Code Size The impact of CARTRA in the code size is not an influential fact because in a desktop context the memory size in general is not a restriction. In other contexts such as WSN, the storage restriction should be addressed. CARTRA can probably address this restriction.

Good Results There is no guarantee that an ACO algorithm will find the same results in different executions. It was showed in experiments with CARTRA. In fact, the best thing to do is to execute ACO algorithms several times, and CARTRA either.

4.5 Summary

Register allocation determines what values in a program must reside in registers, due to instructions involving register operands are faster than those involving memory access. Therefore, register allocation is a very significant compiler optimization technique and can be mapped as a graph coloring problem.

Due the nature of this problem, register allocators based on graph coloring algorithm apply some heuristic method to find a good coloring. These allocators do not guarantee that the coloring is the best. CARTRA indicated that is possible a register allocator based on graph coloring provides good solutions, even though at a cost of a high compile time.

5 Concluding Remarks

The *ColorAnts-RT* algorithms are used to find solutions to the GCP. This paper presents the algorithms *ColorAnt₁-RT*, *ColorAnt₂-RT*, and *ColorAnt₃-RT*, which are based on the Ant Colony Optimization metaheuristic, beyond using a React-Tabucol local search. The main difference is the way the pheromone trails are updated by the ants, and how often the local search is used. Among them, the *ColorAnt₃-RT* obtained the best performance.

An important calibration process was done to adjust the parameters of the *ColorAnts-RT*, using the strategy of calibrating each parameter independently of each others. However, new researches should treat the adjustment of the parameters considering some relationship between the parameters values.

In a general manner, the performance of the *ColorAnts-RT* were influenced by an interesting characteristic of the graph instances: if it was (or not) used some kind of randomness in the creation of the instance. The randomness tends to lead a worst performance, than in the instances without randomness.

ColorAnt₃-RT has presented an excellent performance in the graph instances derived from the register allocation problems, namely the ones of classes *fpsol*, *inithx*, *mulsol* and *zeroin*. This indicates that *ColorAnt₃-RT* could be well suitable to this kind of application.

Due to the good results obtained by our *ColorAnt₃-RT*, our team proposed a different approach to solve the register allocation problem: a new ACO-based algorithm for intraprocedural register allocation called CARTRA.

CARTRA modifies a traditional graph coloring register allocator in order to use the *ColorAnt₃-RT* algorithm. This modification enables the use of a different strategy to select spill. In CARTRA this selection is based on conflicting vertices, and not in spill cost as in other allocators. The experiments with CARTRA demonstrated that it is able to decrease the traffic between processor and memory, consequently decreasing the runtime.

Acknowledgements

The first author would like to thank São Paulo Research Foundation, FAPESP (grant 2013/01172-0).

References

- [1] Ian Fuat Akyildiz and Mehmet Can Vuran. *Wireless Sensor Networks*. John Wiley & Sons Inc, 2010.
- [2] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, New York, NY, EUA, 1998.
- [3] David L. Applegate, Robert E. Bixby, Vasek Chvátal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2007.
- [4] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeno JVM. *SIGPLAN Notices*, 46:65–83, May 2011.
- [5] Daniel Ashlock. *Evolutionary Computation for Modeling and Optimization*. Springer, 2005.
- [6] Leonid Barenboim and Michael Elkin. Combinatorial algorithms for distributed graph coloring. In *Proceedings of the 25th International Conference on Distributed Computing, DISC'11*, pages 66–81, Berlin, Heidelberg, 2011. Springer-Verlag.

- [7] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O’Keefe. Spill Code Minimization via Interference Region Spilling. *SIGPLAN Notices*, 32:287–295, May 1997.
- [8] Anabela Moreira Bernardino, Eugénia Moreira Bernardino, Juan Manuel Sánchez-Pérez, Juan Antonio Gómez-Pulido, and Miguel Angel Vega-Rodríguez. Efficient load balancing for a resilient packet ring using artificial bee colony. In *Proceedings of the 2010 International Conference on Applications of Evolutionary Computation - Volume Part II*, EvoCOMNET’10, pages 61–70, Berlin, Heidelberg, 2010. Springer-Verlag.
- [9] D. Bernstein, M. Golumbic, y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill Code Minimization Techniques for Optimizing Compilers. *SIGPLAN Notices*, 24:258–263, June 1989.
- [10] Ivo Blöchliger and Nicolas Zufferey. A Graph Coloring Heuristic Using Partial Solutions and A Reactive Tabu Scheme. *Computers & Operations Research*, 35(3):960–975, 2008.
- [11] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph Theory*, volume 244 of *Graduate Texts in Mathematics*. Springer, New York, NY, EUA, 2008.
- [12] Daniel Brélaz. New Methods to color the Vertices of a Graph. *Communications of the ACM*, 22(4):251–256, 1979.
- [13] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transaction on Programming Languages and Systems*, 16:428–455, May 1994.
- [14] Gregory J. Chaitin. Register Allocation & Spilling via Graph Coloring. *SIGPLAN Notices*, 17:98–101, 1982.
- [15] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register Allocation via Coloring. *Computer Languages*, 6(1):47 – 57, 1981.
- [16] Daniil Chivilikhin and Vladimir Ulyantsev. Muacosm: A new mutation-based ant colony optimization algorithm for learning finite-state machines. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO ’13, pages 511–518, New York, NY, USA, 2013. ACM.
- [17] Dario Coltorti and Andrea E. Rizzoli. Ant colony optimization for real-world vehicle routing problems. *SIGEVolution*, 2(2):2–9, July 2007.
- [18] Francesc Comellas and Javier Ozón. An Ant Algorithm for the Graph Colouring Problem. In *Proceedings of the First International Workshop on Ant Colony Optimization*, pages 151–158, Heidelberg, Berlin, 1998. Springer.
- [19] Keith D. Cooper and Anshuman Dasgupta. Tailoring Graph-coloring Register Allocation For Runtime Compilation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 39–49, Washington, DC, USA, 2006. IEEE Computer Society.
- [20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 3rd edition, 2009.
- [21] D. Costa and Alain Hertz. Ants Can Colour Graphs. *The Journal of the Operational Research Society*, 48(3):295–305, 1997.
- [22] Jean-Marc Daveau, Thomas Thery, Thierry Lepley, and Miguel Santana. A Retargetable Register Allocation Framework for Embedded Processors. *SIGPLAN Notices*, 39:202–210, June 2004.
- [23] Marco Dorigo and Luca Maria Gambardella. Ant Colony System: A Cooperative Learning Approach to The Traveling Salesman Problem. *IEEE Transaction on Evolutionary Computation*, 1(1):53–66, 1997.
- [24] Marco Dorigo and Socha Krzysztof. An Introduction to Ant Colony Optimization. *IRIDIA Technical Report Series*, 2006.
- [25] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant System: An Autocatalytic Optimizing Process. Technical Report TR91-016, Politecnico di Milano, Politecnico di Milano, Italia, 1991.
- [26] Marco Dorigo, Vittorio Maniezzo, and Alberto 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.
- [27] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Books. MIT Press, Cambridge, Massachusetts, 2004.
- [28] C. N. Fischer, R. K. Cytron, and R. J. LeBlanc. *Crafting a Compiler*. Addison Wesley, 2010.
- [29] Hanna Furmańczyk, Adrian Kosowski, and PawełŻyliński. Scheduling with precedence constraints: Mixed graph coloring in series-parallel graphs. In *Proceedings of the 7th International Conference on Parallel Processing and Applied Mathematics*, PPAM’07, pages 1001–1008, Berlin, Heidelberg, 2008. Springer-Verlag.
- [30] Philippe Galinier and Jin-Kao Hao. Hybrid Evolutionary Algorithms for Graph Coloring. *Journal of Combinatorial Optimization*, 3(4):379–397, 1999.

- [31] Philippe Galinier and Alain Hertz. A Survey of Local Search Methods for Graph Coloring. *Computers & Operations Research*, 33(9):2547–2562, 2006.
- [32] Lal George and Andrew W. Appel. Iterated Register Coalescing. *ACM Transactions on Programming Languages and Systems*, 18:300–324, May 1996.
- [33] Fred Glover. Tabu Search - Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.
- [34] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.
- [35] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.
- [36] FV Group. Snu real-time benchmarks. <http://www.cprover.org/goto-cc/examples/snu.html>.
- [37] Anna Hać. *Wireless Sensor Network Designs*. John Wiley, San Francisco, CA, USA, 2003.
- [38] A. Hertz and D. Werra. Using Tabu Search Techniques for Graph Coloring. *Computing*, 39:345–351, 1987.
- [39] Alain Hertz and Nicolas Zufferey. A New Ant Algorithm for Graph Coloring. In *Proceedings of the Workshop on Nature Inspired Cooperative Strategies for Optimization NICSO*, pages 51–60, Granada, Espanha, 2006. David Alejandro Pelta and Natalio Krasnogor.
- [40] Alain Hertz and Nicolas Zufferey. Vertex coloring using ant colonies. In N. Monmarché, F. Guinand, and P. Siarry, editors, *Artificial Ants: From Collective Intelligence to Real-life Optimization and Beyond*, France, 2010. Wiley.
- [41] Mohammad Ilyas and Imad Mahgoub, editors. *Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems*. CRC PRESS, 2004.
- [42] Kazuaki Ishizaki, Mikio Takeuchi, Kiyokuni Kawachiya, Toshio Suganuma, Osamu Gohda, Tatsushi Inagaki, Akira Koseki, Kazunori Ogata, Motohiro Kawahito, Toshiaki Yasue, Takeshi Ogasawara, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Effectiveness of Cross-platform Optimizations for a Java Just-in-Time Compiler. *SIGPLAN Notices*, 38:187–204, October 2003.
- [43] Junzhong Ji, Ning Zhang, Chunnian Liu, and Ning Zhong. An ant colony optimization algorithm for learning classification rules. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, WI '06, pages 1034–1037, Washington, DC, USA, 2006. IEEE Computer Society.
- [44] Erik Johansson and Konstantinos F. Sagonas. Linear Scan Register Allocation in a High-Performance Erlang Compiler. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, pages 101–119, London, UK, UK, 2002. Springer-Verlag.
- [45] D.S. Johnson and M.A. Trick. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation challenge*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, Providence, RI, EUA, 1996.
- [46] Richard Manning Karp. Reducibility Among Combinatorial Problems. In R.E. Miller and J.M. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, NY, EUA, 1972.
- [47] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.
- [48] Manuel Laguna and Rafael Martí. A GRASP for Coloring Sparse Graphs. *Computational Optimization and Applications*, 19(2):165–178, 2001.
- [49] F. T. Leighton. A Graph Coloring Algorithm for Large Scheduling Problems. *Journal of Research of the National Bureau of Standards*, 84(6):489–506, 1979.
- [50] Helena Lourenço, Olivier Martin, and Thomas Stützle. Iterated local search. In Fred Glover and Gary Kochenberger, editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*, pages 320–353. Springer, New York, NY, EUA, 2003.
- [51] Francisco Luna, Christian Blum, Enrique Alba, and Antonio J. Nebro. Aco vs eas for solving a real-world frequency assignment problem in gsm networks. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 94–101, New York, NY, USA, 2007. ACM.
- [52] Anjali Mahajan and M. S. Ali. Hybrid Evolutionary Algorithm Based Solution for Register Allocation for Embedded Systems. *Journal of Computers*, 3(6), 2008.
- [53] Enrico Malaguti, Michele Monaci, and Paolo Toth. A Metaheuristic Approach for the Vertex Coloring Problem. *INFORMS Journal on Computing*, 20(2):302–316, 2008.
- [54] Peter Marwedel. *Embedded System Design*. Springer Verlag, 2010.

- [55] Craig Morgenstern. Distributed Coloration Neighborhood Search. In David S. Johnson and Michael Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 335–357. American Mathematical Society, Providence, RI, EUA, 1996.
- [56] Hanspeter Mössenböck and Michael Pfeiffer. Linear Scan Register Allocation in the Context of SSA Form and Register Constraints. In *Proceedings of the International Conference on Compiler Construction*, pages 229–246, London, UK, 2002. Springer-Verlag.
- [57] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.
- [58] Gabriela Ochoa, Rong Qu, and Edmund K. Burke. Analyzing the landscape of a graph based hyper-heuristic for timetabling problems. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 341–348, New York, NY, USA, 2009. ACM.
- [59] D. A. Patterson and J. L. Hennessy. *Computer Organization And Design: The Hardware/software Interface*. The Morgan Kaufmann, 2008.
- [60] Matthieu Plumettaz, David Schindl, and Nicolas Zufferey. Ant Local Search and Its Efficient Adaptation to Graph Colouring. *Journal of the Operational Research Society*, 61(5):819–826, 2010.
- [61] Massimiliano Poletto and Vivek Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems*, 21:895–913, September 1999.
- [62] Zhigang Ren and Zuren Feng. An ant colony optimization approach to the multiple-choice multidimensional knapsack problem. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pages 281–288, New York, NY, USA, 2010. ACM.
- [63] Johan Runeson and Sven-Olof Nyström. Retargetable Graph-Coloring Register Allocation for Irregular Architectures. In *Proceedings of the Software and Compilers for Embedded Systems*, pages 22–8. Springer, 2003.
- [64] Sevin Shamizi and Shahriar Lotfi. Register Allocation via Graph Coloring Using an Evolutionary Algorithm. In *Proceedings of the Second international conference on Swarm, Evolutionary, and Memetic Computing - Volume Part II*, pages 1–8, Berlin, Heidelberg, 2011. Springer-Verlag.
- [65] John Shawe-Taylor and Janez Zerovnik. Ants and Graph Coloring. In *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms*, pages 276–279, Berlin, Heidelberg, 2001. Springer.
- [66] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A Generalized Algorithm for Graph-coloring Register Allocation. *SIGPLAN Notices*, 39:277–288, June 2004.
- [67] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. *SIGPLAN Not.*, 39(6):277–288, June 2004.
- [68] W. Stallings. *Computer Organization and Architecture*. Prentice Hall, 2010.
- [69] T. Stützle and HH Hoos. Improvements on the Ant System: Introducing the MAX-MIN Ant System. *Artificial Neural Networks and Genetic Algorithms*, Wien New York: Springer Verlag, 1995.
- [70] T. Suganuma, T. Ogasawara, K. Kawachiya, M. Takeuchi, K. Ishizaki, A. Koseki, T. Inagaki, T. Yasue, M. Kawahito, T. Onodera, H. Komatsu, and T. Nakatani. Evolution of a Java Just-in-Time Compiler for IA-32 Platforms. *IBM Journal of Research and Development*, 48:767–795, September 2004.
- [71] H. R. Topcuoglu, B. Demiroz, and M. Kandemir. Solving the register allocation problem for embedded systems using a hybrid evolutionary algorithm. *IEEE Transactions on Evolutionary Computation*, 11(5):620–634, October 2007.
- [72] Marcos Villagra and Benjamín Barán. Ant colony optimization with adaptive fitness function for satisfiability testing. In *Proceedings of the 14th International Conference on Logic, Language, Information and Computation*, WoLLIC'07, pages 352–361, Berlin, Heidelberg, 2007. Springer-Verlag.
- [73] Christian Wimmer and Hanspeter Mössenböck. Optimized Interval Splitting in a Linear Scan Register Allocator. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 132–141, New York, NY, USA, 2005. ACM.
- [74] Michael Wolfe. How Compilers and Tools Differ for Embedded Systems. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 1–1, New York, NY, USA, 2005. ACM.
- [75] Shengning Wu and Sikun Li. Extending Traditional Graph-Coloring Register Allocation Exploiting Metaheuristics for Embedded Systems. In *Proceedings of the Third International Conference on Natural Computation*, pages 324–329, Washington, DC, USA, 2007. IEEE Computer Society.