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
España

Disponible en: http://www.redalyc.org/articulo.oa?id=92538718007
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 it 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 $NP$-hard combinatorial optimization problem [10]. 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].

$NP$-hard problems demand exact algorithms in superpolynomial time to obtain an optimal solution, unless $P = NP$ [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 [15], bees [8] and
terms. 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 [74], machine learning [16] and the previously referred graph coloring [24].

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 [53, 31, 40, 66, 68].
The experiments with ColorAnt-RT algorithms were performed in forty-nine graphs, of the well known DIMACS challenge [45]. The results indicate that ColorAnts-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, 24] 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 ColorAnts3-RT Register Allocator (CARTRA), which is based on the ColorAnts-RT algorithm. CARTRA extends the Iterated Register Coalescing Allocator (IRA) [2] to use the ACO-based ColorAnts-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 ColorAnts-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 \( k = \chi(G) \) [14].

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
\[
s : V \rightarrow \{1, \ldots, k\} \quad \forall (v_i, v_j) \in E : s(v_i) \neq s(v_j).
\]
Another approach is the partitioning of $V$ in $k$ independent sets or legal classes (classes of colors)

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

$k$-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 a 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 $\chi$, 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 $P = NP$ [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 [20]. 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) [8] [29]. Since then, studies have been conducted to map the behavior of ants to many others optimization problems, like GCP [16] [17] [23] [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 [37], Iterated Local Search [50], and Genetic Algorithms [53].

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. \hspace{1cm} CONSTRUCT-ANTS-SOLUTIONS();
4. \hspace{1cm} APPLY-LOCAL-SEARCH(); \hspace{1cm} // optional
5. \hspace{1cm} 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 [10]:

- 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 [15]. 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. 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 assigns 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 (C1, . . . , Ck legal classes, and the class Ck+1 of non-colored vertices). A neighbor solution is founded by moving a vertex v ∈ Ck+1 to any class Cc and moving the neighbors of v that are in Cc to Ck+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) [59]. 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 ∀i ∈ V;  // s maps a vertex to a color
3 while NC ≠ {} do
4   choose a vertex v with the highest degree of saturation in NC;
5   choose a color c ∈ 1..k with probability p according to Equation 1
6   s(v) = c;
7   NC = NC \{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 saturation and after choosing a color c to assign v. The color c is chosen based on probability p, as follows:

\[
p(s, v, c) = \frac{\tau(s, v, c)^\alpha \cdot \eta(s, v, c)^\beta}{\sum_{i \in \{1, \ldots, k\}} \tau(s, v, i)^\alpha \cdot \eta(s, v, i)^\beta}
\]

(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:

\[
\tau(s, v, c) = \begin{cases} 
1 & \text{if } C_c(s) = \{\}
\sum_{u \in C_c(s)} \frac{P_{uv}}{|C_c|} & \text{otherwise}
\end{cases}
\]

(2)

\[
\eta(s, v, c) = \frac{1}{|N_{C_c(s)}(v)|}
\]

(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.

\(\text{Degree of saturation is the number of different colors that were already assigned to the adjacent vertices of an uncolored vertex.}\)
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 $\rho$ 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:

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

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

where $C_i(s)$ 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:

- **ColorAnt$_1$-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 ($s^*$); 
- **ColorAnt$_2$-RT**: only $s'$ and $s^*$ are used to reinforce the trail; 
- **ColorAnt$_3$-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 ColorAnt$_1$-RT and ColorAnt$_2$-RT, the local search is applied only to the best ant of the colony, at the end of a cycle. In ColorAnt$_3$-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 [9].**

```
Algorithm 3 React-Tabucol [9].

REACT-TABUCOL(G = (V, E), k, s_0 = {C_1, ..., 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 $\delta(v, c)$;
8        // where $\delta(v, c) = f(s \cup (v, c)) - f(s)$
9     s = (s \cup (v, c)) \setminus (v, s(v));
10    update tl according to reactive tabu scheme;
11   lista_tabu = lista_tabu \cup \{(v, s(v))\}; // for tl iterations
12   if $f(s) < f(s^*)$ then
13      s^* = s;
14   cycles = cycles + 1;
15  return s^*;
```

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

ColorAnt-RT\(\left(G = (V, E), k\right)\)
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 CA1-RT
   \(\Delta P_{uv} = 0 \quad \forall u, v \in V;\)
   \(f' = \infty;\) // best value function in a cycle
   for \(a = 1\) to \(ants\) do
      \(s = \text{ANT\_FIXED\_RT}(G, k);\)
      \(\Delta P_{uv} = \frac{1}{f(s)} \quad \forall u, v \in C_c(s) \mid (u, v) \notin E, c = 1..k;\)
      // Line [12] exists only in CA3-RT
      \(s = \text{REACT\_TABUCOL}(G, k, s);\)
      if \(f(s) = 0\) or \(f(s) < f'\) then
         \(s' = s;\)
         \(f' = f(s);\)
      // Line [10] exists only in CA1-RT and CA2-RT
      \(s' = \text{REACT\_TABUCOL}(G, k, s');\)
      if \(f' < f^*\) then
         \(s^* = s';\)
         \(f^* = f(s^*);\)
      \(P_{uv} = \frac{1}{f(s^*)} \quad \forall u, v \in V;\) // according to Equation [4]
      // Line [21] exists only in CA1-RT
      \(P_{uv} = P_{uv} + \Delta P_{uv} \quad \forall u, v \in V;\)
      // Lines [22] [23] exist only in CA1-RT and CA2-RT
      \(P_{uv} = P_{uv} + \frac{1}{f(s^*)} \quad \forall u, v \in C_c(s') \mid (u, v) \notin E, c = 1..k;\)
      \(P_{uv} = P_{uv} + \frac{1}{f(s^*)} \quad \forall u, v \in C_c(s^*) \mid (u, v) \notin E, c = 1..k;\)
      // Next lines exist only in CA3-RT:
      if \(cycle \mod \sqrt{\text{max\_cycles}} = 0\) then
         \(\text{phero\_counter} = [\text{cycle} \div \sqrt{\text{max\_cycles}}];\)
      if \(\text{phero\_counter} > 0\) then
         \(P_{uv} = P_{uv} + \frac{1}{f(s^*)} \quad \forall u, v \in C_c(s^*) \mid (u, v) \notin E, c = 1..k;\) // according to Equation [5]
      else
         \(P_{uv} = P_{uv} + \frac{1}{f(s^*)} \quad \forall u, v \in C_c(s') \mid (u, v) \notin E, c = 1..k;\) // according to Equation [5]
      \(cycle = cycle + 1;\)
      \(\text{phero\_counter} = \text{phero\_counter} - 1;\)
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 [15], 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 graphs are standard random graphs \( \text{dsjcn}_d \) have \( n \) vertices and any two vertices have a probability \( d \) of being adjacent;
- dsjr500.1, dsjr500.1c and dsjr500.1d geometric random graphs \( \text{dsjrn}_d \) 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 complement of the respective geometric random graph;
- miles500, miles750 and miles1000 graph instances similar to geometric graphs \( \text{dsjrn}_d \), 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 flatn_\( \chi \)0 graphs are generated by partitioning \( n \) vertices into \( \chi \) classes (almost of equal size), and by selecting edges between vertices of different classes, in this way they have a chromatic number \( \chi \). It is used randomness in the generation of theses graphs;
- le450_15c, le450_15d, le450_25c and le450_25d le450_\( \chi \) graphs always have 450 vertices, and a chromatic number \( \chi \). It is used randomness in the generation of theses graphs;
- myciel3, myciel4, myciel5 and myciel6 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 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 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;
- ash331g1ia, ash608g1ia and will1199g1ia graph instances obtained from a matrix partitioning problem, in the segmented columns approach to determine sparse Jacobi 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 graph instances based on register allocation.

ColorAnts-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 ColorAnts-RT. ColorAnt1 has the same two stop conditions of ColorAnts-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 a 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 runs with the smallest value of \( k \), for what there were at least one.

\( ^2 \)Available in \url{http://mat.gsia.cmu.edu/COLOR/instances.html} accessed in December 2012.

\( ^3 \)The 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 \( e \notin E \)).

\( ^4 \)Available in \url{http://mat.gsia.cmu.edu/COLOR002/} accessed in December 2012.

\( ^5 \)A 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 ($nants$, $\alpha$, $\beta$, $\rho$ 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 $\alpha$ and $\beta$, assigning 1, 2, and 3 to the parameter $x$ using the Algorithm 5.

Algorithm 5 Calibrate-$\alpha$-$\beta$

```
CALIBRATE-$\alpha$-$\beta$(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 $\alpha$ and $\beta$, the next step was to calibrate $\rho$. Basically, it was used the same strategy as
before as can be seen in the Algorithm 6.

Algorithm 6 Calibrate-$\rho$

```
CALIBRATE-$\rho$(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 Algorithms 5 with
the difference that $\rho$ 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 ColorAnt$_1$-RT, ColorAnt$_2$-RT and ColorAnt$_3$-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 ($\chi/k^*$),
where a “?” denotes that the value of $\chi$ 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

<table>
<thead>
<tr>
<th>Graph</th>
<th>ColorAnt1-RT</th>
<th>ColorAnt2-RT</th>
<th>ColorAnt3-RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(E)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\chi/k)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\alpha)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\beta)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\rho)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(F)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[\text{Table 2 and 3 report the results obtained by ColorAnt1-RT, ColorAnt2-RT and ColorAnt3-RT. In this table, the first column indicates the name of the graph instance, and the second column shows the pair (\(\chi/k\)). The third column reports the value of } k. \text{ 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 CIs, which reports the average number of conflicting vertices of each attempt (computed by the 10 runs). The number of 10 runs in each attempt is the same as used in [66].}\]

The calibration of the parameters showed in Table 1 had 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”
some information by evaporation of pheromone.

Despite the parameter $p$ has no typical value for ACO algorithms, good solutions were founded as we can see in the instances miles, myciel, fullins, queen, gvia, fpsol, inithx, mulsol 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, gvia, fpsol, inithx, mulsol, and zeroin do not use randomness.

With randomness the ColorAnt-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 dsjr, even in the CPU time.

The ColorAnt-RT shows good performance in dsjc instances, as we can see by the fact that in dsjc250.5 it founded $k^*$, while the ColorAnt-RT did not. On the other hand, ColorAnt-RT founded only the best $k$ (founded by ColorAnt-RT) plus 2 for the dsjc500.5. In general, in dsjc ColorAnt-RT loses in CPU time to the others. However, it is not precise to compare the times between algorithms that found different values of $k$.

In dsjr ColorAnt-RT has a better performance, finding $k^*$ for dsjr500.5 when ColorAnt-RT did not, with the advantage that there is no case that ColorAnt-RT founds worst $k$ than the other algorithms. The CPU time of dsjr follows the dsjcs.
In In $\text{dsjc}$s and $\text{dsjr}$s the best choice is $\text{ColorAnt}_{3}-\text{RT}$, 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 $\text{miles}$, a highlight: $\text{ColorAnt}_{3}-\text{RT}$ has the best times with average smaller than one second, and every run founded the $k^*$. $\text{ColorAnt}_{3}-\text{RT}$ has a similar performance, but it was not able to have 10 succeeded runs in $\text{miles750}$ and $\text{miles1000}$ instances. The worst performance was with $\text{ColorAnt}_{2}-\text{RT}$, that was not able to produce 10 succeeded runs.

In $\text{flat}$ $\text{ColorAnt}_{3}-\text{RT}$ founded just one $k$ (that is 5 colors worst from $k^*$). $\text{ColorAnt}_{2}-\text{RT}$ founded two values $k$ and $\text{ColorAnt}_{3}-\text{RT}$ has four (in total of five). The $\text{flat}$ revealed to be difficult for $\text{ColorAnt}_{3}-\text{RT}$. It was necessary to use runs starting from $k^*+30$, as we can see in $\text{flat1000}_{60}$ and $\text{flat1000}_{760}$. $\text{No ColorAnt-RT}$ founded a solution to $\text{flat1000}_{60}$ instance, while $\text{ColorAnt}_{2}-\text{RT}$ and $\text{ColorAnt}_{3}-\text{RT}$ found $k^*+19$ (95) solutions for the $\text{flat1000}_{760}$. It is difficult to compare CPU time, but considering $\text{ColorAnt}_{3}-\text{RT}$, the average of 2136.802s is higher than all other graph instances.

Focusing on solutions founded by the $\text{ColorAnt}_{3}-\text{RT}$ to instances of 1000 or higher number of vertices, we can compare: (1) the average time of 3430.387s of $\text{flat1000}_{760}$ (with a $k^*+19$ solution founded, 25% far from the $k^*$ solution); (2) the $\text{dsjc1000.1}$, that has time of 426.408s with a solution $k^*+2$ (10% far from the $k^*$ solution); and (3) $\text{ash608gpi}$ 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 $\text{flat1000}_{760}$ and $\text{dsjc1000.1}$, which can give an explanation of their difficulty compared to $\text{ash608gpi}$. Between $\text{flat1000}_{760}$ and $\text{dsjc1000.1}$, we can highlight that $\text{flat}$ is in general more difficult than $\text{dsjc}$ as the results indicate.

In $\text{le450s}$, $\text{ColorAnt}_{3}-\text{RT}$ has not too bad results as in $\text{flats}$, but the numbers are not so good, taking into account that the others founded two $k^*$, and $\text{ColorAnt}_{2}-\text{RT}$ not. However, it is interesting to note that $\text{ColorAnt}_{3}-\text{RT}$ founded a better $k$ than $\text{ColorAnt}_{3}-\text{RT}$ in $\text{le450}_{25c}$. The $\text{ColorAnt}_{3}-\text{RT}$ founded the better values of $k$ with the best average CPU time of 239.043s (against very worst CPU time of the others). For the $\text{le450}_{15c}$ and $\text{le450}_{15d}$, it founded the values of $k^*$, and for the other two instances, it founded $k^*+1$.

The class $\text{mycie}$, and its derived $\text{insertions}$ and $\text{fullins}$, have a good situation: all $\text{ColorAnts-RT}$ founded
for all instances in all runs with the exceptions of just one case, which is ColorAnt$_1$-RT to 1-insertions. The CPU time of ColorAnt$_1$-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 ColorAnt$_2$-RT, because all the runs finding $k^*$ with better average CPU time than ColorAnt$_3$-RT. However, as ColorAnt$_3$-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 ColorAnt$_1$-RT that is well succeeded in all the runs. It has a good average CPU time of 1.783 seconds.

ColorAnt$_1$-RT is worst in gpiia class, while ColorAnt$_2$-RT and ColorAnt$_3$-RT founded the best possible quality of solutions. ColorAnt$_3$-RT has the advantage of a better average CPU time: 5.943 seconds, about a half than the ColorAnt$_2$-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, 1415.05s, 0.94s). An average of these respective values results in 1181.136s, 1637.181s, 5.146s, respectively. It shows a good CPU time for ColorAnt$_3$-RT, which is about 229 times smaller than the second better (ColorAnt$_2$-RT).

The ColorAnt$_3$-RT is the best choice in the most of classes. However, different situations can be seen: in miles ColorAnt$_1$-RT is the best, but ColorAnt$_2$-RT is close to it; in myciel, insertions and fullIns the better choice is ColorAnt$_3$-RT, but it is safe to use ColorAnt$_3$-RT as we have already pointed; and the instances of king queen are better solved with ColorAnt$_1$-RT, but ColorAnt$_3$-RT is acceptable. With this in mind, ColorAnt$_3$-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 [50], TabuCOL [25, 35], MMT [55], HCA [30, 38], 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 $k^*$ are better solved with ColorAnt$_1$-RT that is well succeeded in all the runs. It has a good average CPU time of 1.783 seconds.

**Table 4: Values of $k$ founded by ColorAnt$_3$-RT (CA$_3$-RT), ColorAnt$_2$-RT (CA$_2$-RT), ColorAnt$_1$ (CA$_1$-RT), ALS-COL (ALS), TabuCOL (TC), MMT, HCA, MOR, and PartialCol (PC).**

<table>
<thead>
<tr>
<th>Graph</th>
<th>ALS-COL</th>
<th>ColorAnt$_1$-RT</th>
<th>ColorAnt$_2$-RT</th>
<th>ColorAnt$_3$-RT</th>
<th>ALS-COL</th>
<th>ColorAnt$_1$-RT</th>
<th>ColorAnt$_2$-RT</th>
<th>ColorAnt$_3$-RT</th>
<th>ALS-COL</th>
<th>ColorAnt$_1$-RT</th>
<th>ColorAnt$_2$-RT</th>
<th>ColorAnt$_3$-RT</th>
<th>ALS-COL</th>
<th>ColorAnt$_1$-RT</th>
<th>ColorAnt$_2$-RT</th>
<th>ColorAnt$_3$-RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>dsjc500.1</td>
<td>7/12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>dsjc500.5</td>
<td>7/48</td>
<td>53</td>
<td>52</td>
<td>51</td>
<td>48</td>
<td>49</td>
<td>48</td>
<td>48</td>
<td>49</td>
<td>49</td>
<td>49</td>
<td>49</td>
<td>49</td>
<td>49</td>
<td>49</td>
<td>49</td>
</tr>
<tr>
<td>dsjc1000.1</td>
<td>7/20</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>dsjr500.1c</td>
<td>7/85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
</tr>
<tr>
<td>dsjc500.5</td>
<td>12/122</td>
<td>122</td>
<td>123</td>
<td>122</td>
<td>123</td>
<td>123</td>
<td>123</td>
<td>123</td>
<td>123</td>
<td>123</td>
<td>123</td>
<td>123</td>
<td>123</td>
<td>123</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>flat300.28</td>
<td>28/28</td>
<td>83</td>
<td>82</td>
<td>82</td>
<td>82</td>
<td>82</td>
<td>82</td>
<td>82</td>
<td>82</td>
<td>82</td>
<td>82</td>
<td>82</td>
<td>82</td>
<td>82</td>
<td>82</td>
<td>82</td>
</tr>
<tr>
<td>flat1000.76</td>
<td>76/76</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
</tr>
<tr>
<td>le450.1c</td>
<td>15/15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>le450.1sd</td>
<td>15/15</td>
<td>20</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

In the compared instances of Table 4, ColorAnt$_3$-RT always has better solutions than the other algorithms. Comparing ColorAnt$_1$-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 and flat1000.76 are hard to all the algorithms. Among all of them, flat300.28 has $k = 29$ ($k^* = 28$) as the better solution and flat1000.76 has $k = 82$ ($k^* = 76$). Each of these values is reached just once, and by distinct algorithms. ColorAnt$_3$-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 ColorAnt$_3$-RT presents worst results: $k^* + 1$, $k^* + 3$, and $k^* + 2$ for dsjc500, dsjc500.5, and dsjc1000.1, respectively.

In dsjr500.1c and dsjr500.5, ColorAnt$_3$-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 I 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 100000. 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](image)

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 thought 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 ColorAnts-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 [16, 17]. Briggs et al. developed the most successful design for GCRA [19]. 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 [25]. 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 specificities, 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 [65], 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, 65]. 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.
**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 $T_1$ and $T_2$ 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 ColorAnt$_3$-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 ColorAnt$_3$-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 ColorAnt$_3$-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 ColorAnt$_3$-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
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 vertex, as follows:

\[
P_v = \frac{(\text{uses}_{\text{out}} + \text{defs}_{\text{out}}) + 10 \times (\text{uses}_{\text{in}} + \text{defs}_{\text{in}})}{\text{degree}}\]

where \(\text{uses}_{\text{out}}\) is the set of temporaries that the node uses outside a loop, \(\text{defs}_{\text{out}}\) is the set of temporaries that it defines outside a loop, \(\text{uses}_{\text{in}}\) is the set of temporaries that it uses within a loop, \(\text{defs}_{\text{in}}\) is the set of temporaries that it defines within a loop, and \(\text{degree}\) is the number of edges incident to the node.

The node that has the lowest priority will be selected to be spilled first. IRA's approach is an optimistic approximation: the node removed from the graph does not interfere with any of the other 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 were conducted. Such experiments were 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, INTX, 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 \(nants = 10, \alpha = 1.0, \beta = 1.0, \rho = 0.0, \text{max}\_\text{cycles} = 50\), and \(\text{tabu}\_\text{search}\_\text{cycles} = 50\). CARTRA's coloring phase (ColorAnt3-RT) stops if there is no improvement in reducing the number of conflicting edges for more than \(\text{max}\_\text{cycles}/4\).

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Function</td>
</tr>
<tr>
<td>real</td>
<td>50</td>
</tr>
<tr>
<td>encode</td>
<td>396</td>
</tr>
<tr>
<td>decode</td>
<td>401</td>
</tr>
<tr>
<td>reset</td>
<td>2457</td>
</tr>
<tr>
<td>filter</td>
<td>64</td>
</tr>
<tr>
<td>fftep</td>
<td>21</td>
</tr>
<tr>
<td>quantl</td>
<td>55</td>
</tr>
<tr>
<td>logft</td>
<td>41</td>
</tr>
<tr>
<td>scalel</td>
<td>30</td>
</tr>
<tr>
<td>usize</td>
<td>129</td>
</tr>
<tr>
<td>upper</td>
<td>11</td>
</tr>
<tr>
<td>uppol2</td>
<td>38</td>
</tr>
<tr>
<td>invqah</td>
<td>22</td>
</tr>
<tr>
<td>logct</td>
<td>30</td>
</tr>
<tr>
<td>sin</td>
<td>94</td>
</tr>
<tr>
<td>main</td>
<td>180</td>
</tr>
<tr>
<td>Binary Search</td>
<td>60</td>
</tr>
<tr>
<td>main</td>
<td>173</td>
</tr>
<tr>
<td>FFT</td>
<td>50</td>
</tr>
<tr>
<td>main</td>
<td>38</td>
</tr>
<tr>
<td>FFT Complex</td>
<td>49</td>
</tr>
<tr>
<td>main</td>
<td>38</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>20</td>
</tr>
<tr>
<td>main</td>
<td>11</td>
</tr>
<tr>
<td>FIR</td>
<td>40</td>
</tr>
<tr>
<td>35 points</td>
<td>34</td>
</tr>
<tr>
<td>main</td>
<td>53</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>149</td>
</tr>
<tr>
<td>Jfdctint</td>
<td>622</td>
</tr>
<tr>
<td>(64 elements)</td>
<td>main</td>
</tr>
<tr>
<td>LMS</td>
<td>49</td>
</tr>
<tr>
<td>(64 sine wave length)</td>
<td>main</td>
</tr>
<tr>
<td>Matmul</td>
<td>56</td>
</tr>
<tr>
<td>A[5x10^3][5x10^3]</td>
<td>main</td>
</tr>
<tr>
<td>B[5x10^3][5x10^3]</td>
<td>main</td>
</tr>
<tr>
<td>Minver</td>
<td>82</td>
</tr>
<tr>
<td>A[3x10^6][3x10^6]</td>
<td>main</td>
</tr>
<tr>
<td>Select</td>
<td>416</td>
</tr>
<tr>
<td>main</td>
<td>154</td>
</tr>
<tr>
<td>Sort</td>
<td>38</td>
</tr>
<tr>
<td>N = 1234</td>
<td>main</td>
</tr>
</tbody>
</table>
• 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 relegate 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.

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 jfdct, 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.

<table>
<thead>
<tr>
<th>Program</th>
<th>Function</th>
<th>Min</th>
<th>Average</th>
<th>Max</th>
<th>S. Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Split</td>
<td>Fetch</td>
<td>Split</td>
<td>Fetch</td>
</tr>
<tr>
<td>SIN</td>
<td></td>
<td>11</td>
<td>22.21</td>
<td>22</td>
<td>4.44</td>
</tr>
<tr>
<td>SQRT</td>
<td></td>
<td>8</td>
<td>11.22</td>
<td>9</td>
<td>11.44</td>
</tr>
<tr>
<td>MAIN</td>
<td></td>
<td>5</td>
<td>11.89</td>
<td>27</td>
<td>34</td>
</tr>
<tr>
<td>FIR</td>
<td></td>
<td>10</td>
<td>26.56</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>GAUSSIAN</td>
<td></td>
<td>9</td>
<td>10.34</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>12</td>
<td>21.44</td>
<td>12</td>
<td>21</td>
</tr>
</tbody>
</table>

Though FIR filter and SQRT 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 obtain 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 to 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:

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

was transformed in:

```c
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:

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

was also transformed in:

```c
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
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

<table>
<thead>
<tr>
<th>Program</th>
<th>Function</th>
<th>CARTRA</th>
<th>IRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>rad</td>
<td><a href="4">5,2,1,0</a></td>
<td><a href="2">5,0</a></td>
<td></td>
</tr>
<tr>
<td>encode</td>
<td><a href="2">16,0</a></td>
<td><a href="7">16,3,3,2,1,0</a></td>
<td></td>
</tr>
<tr>
<td>decode</td>
<td><a href="3">16,0</a></td>
<td><a href="4">10,4,5,1,0</a></td>
<td></td>
</tr>
<tr>
<td>rect</td>
<td><a href="4">16,0</a></td>
<td><a href="4">10,4,5,1,0</a></td>
<td></td>
</tr>
<tr>
<td>filter</td>
<td><a href="2">8,0</a></td>
<td><a href="5">8,2,1,1,0</a></td>
<td></td>
</tr>
<tr>
<td>ipplep</td>
<td><a href="4">8,1,0</a></td>
<td><a href="4">8,1,1,0</a></td>
<td></td>
</tr>
<tr>
<td>quad1</td>
<td><a href="5">8,3,0</a></td>
<td><a href="4">8,1,0</a></td>
<td></td>
</tr>
<tr>
<td>logcel</td>
<td><a href="2">4,0</a></td>
<td><a href="2">3,0</a></td>
<td></td>
</tr>
<tr>
<td>scale</td>
<td><a href="2">4,0</a></td>
<td><a href="6">4,1,1,2,0</a></td>
<td></td>
</tr>
<tr>
<td>snumcro</td>
<td><a href="14">4,3,2,2,2,2,1,1,0</a></td>
<td><a href="19">10,9,8,4,3,2,1,0</a></td>
<td></td>
</tr>
<tr>
<td>uppo1</td>
<td><a href="6">9,5,1,1,0</a></td>
<td><a href="4">3,2,1,0</a></td>
<td></td>
</tr>
<tr>
<td>uppo2</td>
<td><a href="2">3,0</a></td>
<td><a href="5">4,5,3,1,0</a></td>
<td></td>
</tr>
<tr>
<td>uvqah</td>
<td><a href="4">3,1,1,0</a></td>
<td><a href="2">3,0</a></td>
<td></td>
</tr>
<tr>
<td>logsch</td>
<td><a href="2">3,0</a></td>
<td><a href="2">3,0</a></td>
<td></td>
</tr>
<tr>
<td>sim</td>
<td><a href="2">5,0</a></td>
<td><a href="2">5,0</a></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td><a href="2">10,0</a></td>
<td><a href="14">10,8,2,2,1,1,0</a></td>
<td></td>
</tr>
<tr>
<td>Binary</td>
<td>bs</td>
<td><a href="4">9,0</a></td>
<td><a href="10">9,8,1,1,2,1,1,0</a></td>
</tr>
<tr>
<td>Search</td>
<td>main</td>
<td><a href="2">3,0</a></td>
<td><a href="4">33,3,1,6,0</a></td>
</tr>
<tr>
<td>FFT</td>
<td>sin</td>
<td><a href="2">5,0</a></td>
<td><a href="2">5,0</a></td>
</tr>
<tr>
<td></td>
<td>ft</td>
<td><a href="3">38,2,0</a></td>
<td><a href="5">32,2,1,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="4">9,0</a></td>
<td><a href="4">8,3,2,0</a></td>
</tr>
<tr>
<td>FFT</td>
<td>sim</td>
<td><a href="2">5,0</a></td>
<td><a href="2">5,0</a></td>
</tr>
<tr>
<td>Complex</td>
<td>main_w</td>
<td><a href="4">0,0</a></td>
<td><a href="6">6,2,6,1,0</a></td>
</tr>
<tr>
<td></td>
<td>ft</td>
<td><a href="2">22,0</a></td>
<td><a href="3">18,3,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="4">0,0</a></td>
<td><a href="4">6,5,0</a></td>
</tr>
<tr>
<td>Fibonaci</td>
<td>fib</td>
<td><a href="4">2,1,0</a></td>
<td><a href="4">2,1,1,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="2">3,0</a></td>
<td><a href="1">3,0</a></td>
</tr>
<tr>
<td>FIR</td>
<td>sin</td>
<td><a href="2">5,0</a></td>
<td><a href="2">5,0</a></td>
</tr>
<tr>
<td></td>
<td>sqrt</td>
<td><a href="4">7,0</a></td>
<td><a href="4">7,0</a></td>
</tr>
<tr>
<td></td>
<td>if_filter</td>
<td><a href="3">8,3,0</a></td>
<td><a href="3">8,5,0</a></td>
</tr>
<tr>
<td></td>
<td>gaussian</td>
<td><a href="4">6,0</a></td>
<td><a href="4">8,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="2">9,0</a></td>
<td><a href="2">8,0</a></td>
</tr>
<tr>
<td>Insert</td>
<td>main</td>
<td><a href="2">11,0</a></td>
<td><a href="3">10,8,3,1,0</a></td>
</tr>
<tr>
<td>Jfdctint</td>
<td>mix</td>
<td><a href="3">16,4,0</a></td>
<td><a href="2">23,9,2</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="2">6,0</a></td>
<td><a href="5">6,3,1,1,0</a></td>
</tr>
<tr>
<td>LMS</td>
<td>sqrt</td>
<td><a href="2">7,0</a></td>
<td><a href="2">7,0</a></td>
</tr>
<tr>
<td></td>
<td>sim</td>
<td><a href="2">5,0</a></td>
<td><a href="2">5,0</a></td>
</tr>
<tr>
<td></td>
<td>gaussian</td>
<td><a href="4">6,1,0</a></td>
<td><a href="4">8,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="4">22,2,2,0</a></td>
<td><a href="7">21,2,2,2,1,0</a></td>
</tr>
<tr>
<td></td>
<td><a href="3">20,1,0</a></td>
<td><a href="5">19,2,1,1,0</a></td>
<td></td>
</tr>
<tr>
<td>Matmul</td>
<td>alloc</td>
<td><a href="3">7,2,0</a></td>
<td><a href="7">9,2,2,1,2,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="4">18,1,0</a></td>
<td><a href="4">18,1,0</a></td>
</tr>
<tr>
<td></td>
<td><a href="4">10,3,1,0</a></td>
<td><a href="7">11,2,4,3,7,1,0</a></td>
<td></td>
</tr>
<tr>
<td>Minver</td>
<td>alloc</td>
<td><a href="4">0,0</a></td>
<td><a href="2">4,0</a></td>
</tr>
<tr>
<td></td>
<td>minin</td>
<td><a href="14">16,3,3,0</a></td>
<td><a href="2">17,0</a></td>
</tr>
<tr>
<td></td>
<td>minver</td>
<td><a href="5">38,12,3,1,0</a></td>
<td><a href="6">39,8,8,9,7</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="4">14,6,2,0</a></td>
<td><a href="8">14,6,4,6,4,3,0</a></td>
</tr>
<tr>
<td>Quick</td>
<td>sort</td>
<td><a href="5">15,1,1,0</a></td>
<td><a href="8">15,2,2,2,2,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="2">2,0</a></td>
<td><a href="6">23,2,1,20,20,0</a></td>
</tr>
<tr>
<td>Sort</td>
<td>sqrt</td>
<td><a href="2">7,0</a></td>
<td><a href="2">7,0</a></td>
</tr>
<tr>
<td></td>
<td>sqrt</td>
<td><a href="4">12,2,1,0</a></td>
<td><a href="10">12,2,4,3,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="2">4,0</a></td>
<td><a href="2">4,0</a></td>
</tr>
<tr>
<td>Select</td>
<td>select</td>
<td><a href="4">16,1,1,0</a></td>
<td><a href="9">19,1,6,4,3,2,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="2">2,0</a></td>
<td><a href="10">23,2,1,20,20,0</a></td>
</tr>
<tr>
<td>Sqrt</td>
<td>sqrt</td>
<td><a href="2">7,0</a></td>
<td><a href="2">7,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="2">10,1</a></td>
<td><a href="2">10,1</a></td>
</tr>
</tbody>
</table>

### 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 runtime.
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.

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.
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) [30]. 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
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>
<thead>
<tr>
<th>Program</th>
<th>Average</th>
<th>Standard Deviation</th>
<th>Average</th>
<th>Standard Deviation</th>
<th>Average</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary Search</td>
<td>0.472</td>
<td>0.005</td>
<td>26.059</td>
<td>0.322</td>
<td>0.168</td>
<td>0.001</td>
</tr>
<tr>
<td>FFT</td>
<td>0.067</td>
<td>0.037</td>
<td>43.301</td>
<td>4.461</td>
<td>0.213</td>
<td>0.008</td>
</tr>
<tr>
<td>FIR</td>
<td>1.421</td>
<td>0.034</td>
<td>235.817</td>
<td>18.697</td>
<td>1.317</td>
<td>0.054</td>
</tr>
<tr>
<td>Insert Sort</td>
<td>0.285</td>
<td>0.106</td>
<td>22.495</td>
<td>7.287</td>
<td>0.110</td>
<td>0.000</td>
</tr>
<tr>
<td>Jdcttun</td>
<td>1.705</td>
<td>0.360</td>
<td>644.870</td>
<td>386.634</td>
<td>1.402</td>
<td>0.045</td>
</tr>
<tr>
<td>LMS</td>
<td>0.810</td>
<td>0.089</td>
<td>81.079</td>
<td>11.072</td>
<td>0.358</td>
<td>0.002</td>
</tr>
<tr>
<td>Quicksort</td>
<td>1.326</td>
<td>0.247</td>
<td>347.165</td>
<td>224.391</td>
<td>1.180</td>
<td>0.000</td>
</tr>
<tr>
<td>Qurt</td>
<td>0.674</td>
<td>0.059</td>
<td>144.351</td>
<td>45.332</td>
<td>0.238</td>
<td>0.006</td>
</tr>
<tr>
<td>Select</td>
<td>1.327</td>
<td>0.079</td>
<td>363.318</td>
<td>180.627</td>
<td>1.510</td>
<td>0.065</td>
</tr>
<tr>
<td>Sqrt</td>
<td>0.079</td>
<td>0.000</td>
<td>4.414</td>
<td>0.013</td>
<td>0.047</td>
<td>0.008</td>
</tr>
</tbody>
</table>

These results also demonstrated the instability of ACO algorithms. 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.

<table>
<thead>
<tr>
<th>Program</th>
<th>HECRA</th>
<th>Average</th>
<th>Standard Deviation</th>
<th>Average</th>
<th>Standard Deviation</th>
<th>Average</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary Search</td>
<td>17</td>
<td>18</td>
<td>18.3</td>
<td>19.5</td>
<td>20</td>
<td>21</td>
<td>18</td>
</tr>
<tr>
<td>FFT</td>
<td>54</td>
<td>36</td>
<td>56.1</td>
<td>84.6</td>
<td>63</td>
<td>101</td>
<td>92</td>
</tr>
<tr>
<td>FIR</td>
<td>4</td>
<td>4</td>
<td>5.5</td>
<td>4.3</td>
<td>9</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>Insert Sort</td>
<td>13</td>
<td>32</td>
<td>15.6</td>
<td>35.0</td>
<td>19</td>
<td>38</td>
<td>16</td>
</tr>
<tr>
<td>LMS</td>
<td>75</td>
<td>121</td>
<td>108.2</td>
<td>156.4</td>
<td>293</td>
<td>343</td>
<td>87</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>39</td>
<td>103</td>
<td>34.7</td>
<td>105.4</td>
<td>48</td>
<td>109</td>
<td>40</td>
</tr>
<tr>
<td>Select</td>
<td>42</td>
<td>85</td>
<td>48.1</td>
<td>90.6</td>
<td>56</td>
<td>98</td>
<td>40</td>
</tr>
<tr>
<td>Sqrt</td>
<td>8</td>
<td>10</td>
<td>9.7</td>
<td>14.0</td>
<td>12</td>
<td>20</td>
<td>8</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Program</th>
<th>Function</th>
<th>HECRA</th>
<th>CARTRA</th>
<th>IRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary Search</td>
<td>bs</td>
<td><a href="2">9,0</a></td>
<td><a href="2">9,0</a></td>
<td><a href="10">9,1,1,1,1,1,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="2">3,0</a></td>
<td><a href="2">3,0</a></td>
<td><a href="4">33,1,6,0</a></td>
</tr>
<tr>
<td>FFT</td>
<td>sin</td>
<td><a href="2">9,0</a></td>
<td><a href="2">9,0</a></td>
<td><a href="5">32,2,1,0</a></td>
</tr>
<tr>
<td></td>
<td>init</td>
<td><a href="4">6,1,0</a></td>
<td><a href="4">38,2,0</a></td>
<td><a href="5">32,2,1,0</a></td>
</tr>
<tr>
<td></td>
<td>ff</td>
<td><a href="4">24,2,0</a></td>
<td><a href="2">9,0</a></td>
<td><a href="4">8,4,2,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="4">9,2,0</a></td>
<td><a href="3">9,0</a></td>
<td><a href="4">8,4,2,0</a></td>
</tr>
<tr>
<td>Fibonacci</td>
<td>fib</td>
<td><a href="3">2,1,0</a></td>
<td><a href="3">2,1,0</a></td>
<td><a href="4">2,1,1,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="1">0</a></td>
<td><a href="2">3,0</a></td>
<td><a href="2">3,0</a></td>
</tr>
<tr>
<td>FIR</td>
<td>sin</td>
<td><a href="2">5,0</a></td>
<td><a href="2">5,0</a></td>
<td><a href="2">5,0</a></td>
</tr>
<tr>
<td></td>
<td>init</td>
<td><a href="3">6,1,0</a></td>
<td><a href="3">38,2,0</a></td>
<td><a href="5">32,2,1,0</a></td>
</tr>
<tr>
<td></td>
<td>fft</td>
<td><a href="3">21,2,0</a></td>
<td><a href="2">9,0</a></td>
<td><a href="4">8,4,2,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="3">4,2,0</a></td>
<td><a href="3">9,0</a></td>
<td><a href="4">8,4,2,0</a></td>
</tr>
<tr>
<td>Fibonacci</td>
<td>sin</td>
<td><a href="2">5,0</a></td>
<td><a href="2">5,0</a></td>
<td><a href="2">5,0</a></td>
</tr>
<tr>
<td></td>
<td>sqrt</td>
<td><a href="2">7,0</a></td>
<td><a href="2">7,0</a></td>
<td><a href="2">7,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="2">7,0</a></td>
<td><a href="2">9,0</a></td>
<td><a href="2">9,0</a></td>
</tr>
<tr>
<td>Insertion</td>
<td>main</td>
<td><a href="2">9,0</a></td>
<td><a href="2">11,0</a></td>
<td><a href="5">10,2,3,1,0</a></td>
</tr>
<tr>
<td>Jfdctint</td>
<td>fdct</td>
<td><a href="2">39,0</a></td>
<td><a href="3">3,2,0</a></td>
<td><a href="2">23,0</a></td>
</tr>
<tr>
<td>LMS</td>
<td>sqrt</td>
<td><a href="2">7,0</a></td>
<td><a href="2">7,0</a></td>
<td><a href="2">7,0</a></td>
</tr>
<tr>
<td></td>
<td>sin</td>
<td><a href="2">5,0</a></td>
<td><a href="2">5,0</a></td>
<td><a href="2">5,0</a></td>
</tr>
<tr>
<td></td>
<td>gaussian</td>
<td><a href="2">5,0</a></td>
<td><a href="2">5,0</a></td>
<td><a href="2">5,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="7">24,4,1,1,0</a></td>
<td><a href="4">22,4,2,0</a></td>
<td><a href="7">21,2,3,2,1,0</a></td>
</tr>
<tr>
<td>Quick Sort</td>
<td>sort</td>
<td><a href="4">16,2,1,0</a></td>
<td><a href="5">15,1,1,0</a></td>
<td><a href="8">15,2,2,2,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="2">2,0</a></td>
<td><a href="2">2,0</a></td>
<td><a href="6">23,2,0,20,20,0</a></td>
</tr>
<tr>
<td>Qurt</td>
<td>sqrt</td>
<td><a href="2">7,0</a></td>
<td><a href="2">7,0</a></td>
<td><a href="2">7,0</a></td>
</tr>
<tr>
<td></td>
<td>qurt</td>
<td><a href="4">11,4,1,0</a></td>
<td><a href="4">12,4,1,0</a></td>
<td><a href="6">12,4,3,2,0</a></td>
</tr>
<tr>
<td>Select</td>
<td>select</td>
<td><a href="4">19,1,1,1,0</a></td>
<td><a href="4">19,1,1,1,0</a></td>
<td><a href="4">19,1,1,1,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="2">2,0</a></td>
<td><a href="2">2,0</a></td>
<td><a href="6">23,2,1,20,20,0</a></td>
</tr>
<tr>
<td>Sqrt</td>
<td>sqrt</td>
<td><a href="2">7,0</a></td>
<td><a href="2">7,0</a></td>
<td><a href="2">7,0</a></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td><a href="1">0</a></td>
<td><a href="1">0</a></td>
<td><a href="1">0</a></td>
</tr>
</tbody>
</table>

The approach based on def-use 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 is 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.
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, mulso1 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


