AntElements: An Extensible and Scalable Ant Colony Optimization Middleware

Kamil Krynicki
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València
Cami de Vera S/N
46022 Valencia, Spain
Telephone: +34 963-878-3525
kkrynicki@dsic.upv.es

Javier Jaen
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València
Cami de Vera S/N
46022 Valencia, Spain
fjaen@upv.es

ABSTRACT

Ant Colony Optimization (ACO) has become a popular metaheuristic approach for solving hard combinatorial optimization problems. However, most existing ACO software systems are domain-specific, dedicated to concrete problems or non-extensible, non-portable and non-scalable solutions that have been evaluated for problem spaces of limited size. In this context, we present AntElements (AntE), a portable Java-based ACO middleware, designed and implemented with the highest consideration for versatility. The extensibility of the proposed middleware allows its use in virtually any ACO deployment, ranging from experimental to commercial. In this work, the overall object-oriented architecture and the software design patterns of AntE are explained, alongside the main concepts behind them. Furthermore, AntE is analyzed with respect to the computational efficiency, parallelization capacities and memory consumption, which allows to establish its usability and scalability range. In its current implementation, on an average to mid-high workstation, our middleware is capable of processing upwards of 10^5 agents per second, in graphs of the order of 10^5 nodes and sustain a stable, fully logged experiment for over 12 hours. The proposed middleware has already been deployed in several research projects that are outlined in this paper, illustrating the range of possibilities it offers.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*modules and interfaces*

Keywords

Ant Colony Optimization, Middleware, Experimentation

GECCO '15, July 11 - 15, 2015, Madrid, Spain

DOI: http://dx.doi.org/10.1145/2739482.2768464

1. INTRODUCTION

Evolutionary computation is an area of intensive research that requires, due to the internal complexity of the algorithms and the data structures involved, advanced software solutions. This is especially true in the case of multiagent, swarm-related or bio-inspired metaheurisitcs, such as Ant Colony Optimization (ACO) [6]. In this specific case, non-trivial software is generally indispensable in order to validate the soundness and effectiveness of the designed ACO strategies. Such implementations must incorporate, to name a few: advanced multithreading handling, software design patterns, software modularity and low level efficiency optimization. Many of these advanced computational techniques often need to be designed and coded by software engineers, but writing a specialized piece of software anew for each iteration of a problem entails considerable workload.

Software Engineering addresses this matter by introducing the concept of middleware. A middleware is a piece of software that provides a set of black-boxed complex algorithms and encapsulated low-level concepts. It enables researchers to work with higher-level concepts or even entire algorithms as atomic entities, significantly shortening the preparation for the experimental phase and breeching the gap between the theoretical computer science and the experimentation. Standardized middlewares also help to make the experimental results more easily comparable, eliminating the potential for an implementation-bound bias.

In this work we present AntElements ($AntE$), an extensible and highly customizable middleware that facilitates an Ant Colony Optimization (ACO) testbed. The main rationale behind the development of AntE was the creation of a software infrastructure that could be used in three ways. First, for educational purposes, by enabling easy interaction. Second, for experiment execution, due to the extensive logging and configuration abilities. Finally, for software deployment, by achieving very strong scalability, portability and compatibility with mobile devices.

The remainder of this paper is organized as follows. In Section 2 we provide an overview of existing ACO middlewares and simulation environments. In Section 3 we present the AntE middleware. Section 4 studies the efficiency and scalability of the proposed software infrastructure and Section 5 discusses the specific deployments that have been realized. We summarize our findings in Section 6.

2. RELATED WORKS

ACO has received only a moderate attention with respect to simulation environments and middlewares. The existing solutions tend to be devoted to one particular ACO implementation or even just one classical problem. An example of this domain-specific implementation trend is *ACOTSP* [14], a high performing software with C and Java versions, dedicated exclusively to the Traveling Salesman Problem. The configuration of ACOTSP is restricted to the command line interface and experimental data are supplied in external files. On the upside, ACOTSP incorporates a number of ACO algorithms, while most middlewares are based on a single ACO strategy. This is the case of another domain-specific package, *hc-mmas-ubqp* [3] which is restricted to MAX-MIN Ant System (MMAS) for Unconstrained Binary Quadratic Programming (UBQP) in Hypercubes. Similar in nature is *AntClique* [13], which was written for maximum clique problems. The three aforementioned software packages are reasonably efficient implementations of ACO, but offer limited flexibility, as they focus on one specific problem. In addition, the C code developed under Linux provides no guarantee to work in different environments, such as Windows, OSX or Android. No distributed versions of the packages exist and adaptation is done mainly on the code-level.

Gui Ant-Miner [11] and *Myra* [2], are two portable, Java-based implementations of ACO. According to the authors, their implementation is reasonably well performing in terms of CPU time and memory consumption, although only experimental and educational uses should be considered. Again, these two packages work exclusively with two concrete ACO algorithms - Ant Colony-based Data Miners, called Ant-Miner and cAnt-Miner, used for extracting classification rules.

A more general-purpose middleware is *JACSF* [4]. The author presents a centralized, bare-bones ACO middleware and performs no study of efficiency or scalability. Many details of the implementation, such as the results logging, are treated simplistically or are completely absent. On the other hand JACSF is easily extensible and very versatile, it can be reprogrammed for any classical problem and it supports any arbitrary ACO algorithm. The author advises to use his software for experimental purposes only.

Another generic solution, AntLib v1.0 [5] is a promising C++ approach to hybrid ACO middlewares. The authors focus on speed, efficiency and scalability, as well as extensibility, with emphasis on the use of templates and object-orientation. Their solution is not bound to any algorithm nor a specific problem. However, the code is for local and experimental use only. It is reported as a work in progress.

Finally, *AntHill* [1], is a very advanced ACO middleware, written in Java, that provides full support for parallel and distributed execution. It operates with JXTA P2P technology and is suited for both: experimental and deployment applications. Even though the authors fail to comment on execution times, an experiment of 5×10^5 iterations is reported which indicates a moderately high scalability. This promising work, however, has not been in development since 2001 and, as a consequence, the structures and technologies it uses have become obsolete. In addition, the customization of AntHill is questionable, as it seems to be strongly bound to the topic of P2P query propagation.

In Table 1 we summarize the results stemming from this brief discussion. Note that in most cases it is not straightforward to establish the scalability and the maximum working ranges of a given middleware. If the ranges are not explicitly specified by the authors we estimate them using the largest documented and published execution of the software we were able to find. Some of the presented middlewares have a fixed number of algorithms incorporated, in such cases a value is provided in the column *Algorithms*. In the *Problems* column group, *Open* indicates that a given middleware can be applied to any problem, rather than just a preestablished one. Finally, *Dynamic* is a unique feature of our middleware. It denotes middlewares that support the modification of the problem mid-execution.

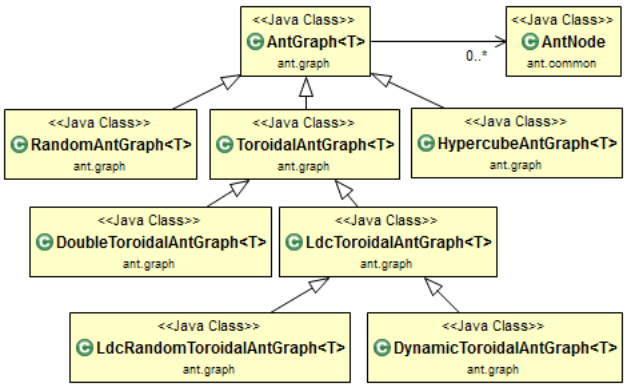


Figure 1: AntE graph family

In general, the middlewares are often not modular nor extensible and the dominating programming language is C under Linux, which limits significantly the deployment possibilities. Thus, the focus tends to be experimental or educational. The authors hardly ever elaborate on the efficiency or the scalability of their software and almost always neglect portability. The efficiency is usually obtained at the cost of flexibility.

Most importantly, the execution environment is always static, i.e, the parameters of the execution have to be provided before it commences and once the algorithm is set in motion no parameters can be changed. Moreover, the evolution of the problem space is universally not permitted.

3. ANTE OVERVIEW

Our solution, AntElements, permits easy creation of complex and compound testing facilities even with limited knowledge of software engineering. Our primary concern, with respect to the software design, was to provide a modular architecture in which each element could be extended and improved upon. It allows to model virtually all ACO-based algorithms and to apply them to an arbitrary problem. In this section we will discuss the high-level architecture of AntE, explain the range of data-logging possibilities, demonstrate snippets of configuration, as well as reveal some interesting low-level optimization techniques.

3.1 Architecture

A very common, practical and justifiable constraint of ACO mathematical models is to represent the contiguous world, in which the real-life ants operate, with a finite graph. To model the problem space in terms of graphs and nodes varies in difficulty. For instance, in the case of the Traveling Salesman Problem the matching between the domain of the

Table 1: Middleware Comparison

						Scalability			Problems		Purpose				
Name	Language	Year	Algorithms	Open	Custom.	Ants	Nodes	Iterations	Open	Dynamic	Edu.	Exp.	Deploy.	Distr.	Multi-thread
ACOTSP	C/Java	2002/4	7	✓	✓*	> 1500	> 2932	N/S			✓	✓			
hc-mmas-ubqp	C++	2004	1	✓		N/S	> 500	N/S			✓	✓			
AntClique	C	2006	1	✓		N/S	> 500	N/S			✓	✓			
Gui Ant-Miner	Java	2002	1	✓	N/S	> 3000	< 5000	N/S			✓	✓			N/S
Myra	Java	2008	1	✓	N/S	> 3000	< 5000	N/S			✓	✓			N/S
JACSF	Java	2009	N/A	✓	✓	> 30	> 50	> 2500	✓		✓	✓			
AntLib v1.0	C++	2008	N/A	✓	✓	N/S	N/S	N/S	✓		✓	✓			
AntHill	Java	2001	N/A	✓	✓	N/S	> 2000	$> 5 \times 10^5$	✓			✓	✓	✓	✓
AntElements	Java	2015	N/A	✓	✓	$> 5 \times 10^6$	> 65536	$> 10^7$	✓	✓	✓	✓	✓	✓	✓

N/A not applicable, N/S not specified, * command line only

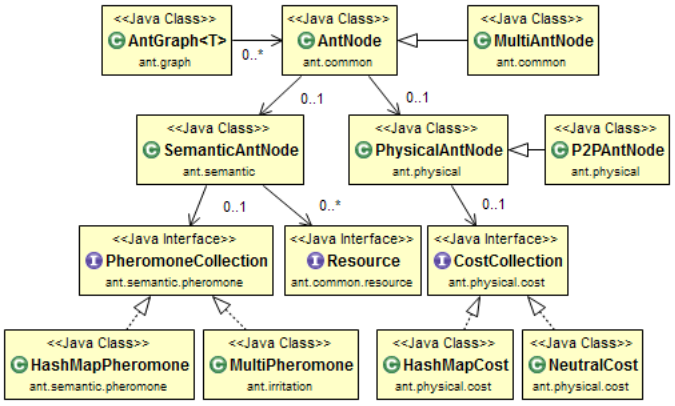


Figure 2: AntNode class diagram

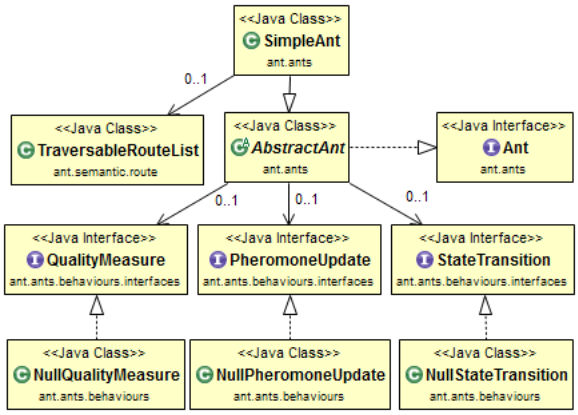


Figure 3: Ant architecture

problem and the graph is straightforward. The nodes would model cities, while the edges - the possible transitions between them. Regardless of the complexity of the modeling process, this approach is considered not to affect the generality of the solutions obtained.

The main component of our architecture is the problem graph (problem-space, Figure 1) which is defined in our model by the *AntGraph* class. *AntGraph* provides a basic interface for manipulating a graph, the graph creation and a set of useful additional methods. We produced a number of *AntGraph* implementations, such as *ToroidalAntGraph*, *HyperCubeAntGraph*, *RandomAntGraph* and more. Furthermore, any custom graph is possible, as well as an extension of any of the preexisting ones.

As shown in Figure 1 *AntGraph* is composed of problem nodes, represented by the *AntNode* class (Figure 2), which are the backbone of the system. The *AntNode* is divided into the *SemanticAntNode* and *PhysicalAntNode*, which correspond to the semantic and the physical levels of the system.

The semantic level is in charge of controlling the pheromone values, as well as maintaining local resources and resolving resource queries. The pheromone manipulation is performed via the *PheromoneCollection* interface. We provide two implementations of the *PheromoneCollection*: *HashMapPheromone* and *MultiPheromone*, which differ in efficiency, scalability and versatility. The choice of the correct implementation of the *PheromoneCollection* is crucial for the performance of the whole model.

The physical level is responsible for maintaining intra-node communication and the cost thereof. AntE comes with two existing implementations: *PhysicalAntNode*, which is optimized for local execution and *P2PAntNode*, which enables deployment in a P2P network. We based our P2P implementation on the PastryRing middleware. If the cost of the communication is either not a factor or an invariant the use of the *NeutralCost* class is recommended, in other cases the *HashMapCost* is made available. The physical level must obligatorily enable access to methods for linking and unlinking nodes and offer the possibility to obtain or modify the costs of all the links present.

The central piece of any ACO-related middleware is the concept of an ant. In AntE the abstraction of the ACO-ant is the *Ant* interface (see Figure 3), which contains, according to the *strategy* software design pattern, three encapsulated and self-explanatory behaviors: *PheromoneUpdate*, *StateTransition* and *QualityMeasure*. This approach allows the code to be hot-pluggable and the behavior to be changed during

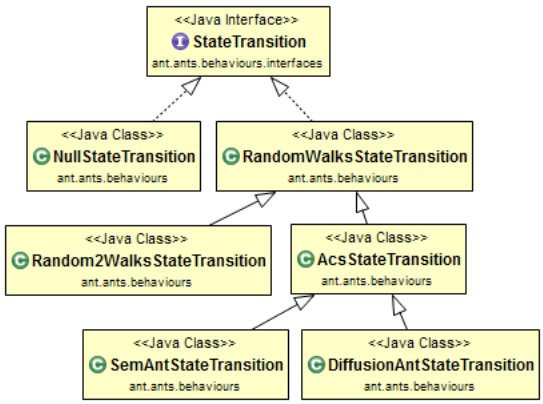


Figure 4: State transition implementations

the algorithm's execution. All three come with a base *null implementation*, as recommended by the *nullobject* design pattern.

```
1 public interface StateTransition
2 {
3   public SemanticId[]
        performStateTransition(SemanticId[] links,
        float[] ph, float[] cs, RouteList visited);
4 }
```

Code Snippet 1: State Transition interface

```
1  public SemanticId[] performStateTransition(SemanticId[]
       links, float[] ph, float[] cs, RouteList visited)
2  {
3    float[] weights = new float[links.length];
4    float   sum = 0;
5
6    float weight;
7    for(int i = 0; i < links.length; i++)
8    {
9      weight = calculateWeight(ph[i], cs[i], beta);
10     sum += weight;
11     weights[i] = weight;
12   }
13
14   double f = this.sum * (new Random().nextFloat());
15
16   int i = -1;
17
18   do
19   {
20       f -= this.weights[++i];
21   }
22   while (f > 0.00001d);
23
24   return links[i];
25 }
```

Code Snippet 2: Example State Transition (ACS)

The state transition interface (Code Snippet 1) takes in four parameters: *links*, an array of links to choose from; *ph* and *cs*, arrays of pheromones and costs corresponding to the links provided; and an optional *visited* parameter: the list of visited nodes, which may or may not influence the state transition. The method returns an array of links to travel to. We chose the return value to be an array, rather than a single value to preserve maximum generality of the code. Some algorithms, such as SemAnt [12], permit ant cloning and splitting, which requires multiple results from a single state transition step. A family of existing state transition rules is provided (Figure 4). See Code Snippet 2 for a simple example of a state transition rule.

```
1 public interface PheromoneUpdate
2 {
3   float[] performLocalUpdate(float[] ph);
4   float[] performGlobalUpdate(float[] ph, int
        rewardedLink, float solutionQuality);
5 }
```

Code Snippet 3: Pheromone Update interface

The pheromone update interface (Code Snippet 3) defines two methods: *performLocalUpdate* and *performGlobalUpdate*, which represent pheromone evaporation and deposition respectively. Both methods take a pheromone array as input, however the deposition requires an additional specification of the link that participates in the deposition process, as well as the quality of the solution obtained.

```
1 public interface QualityMeasure
2 {
3   public float getQuality(Ant a);
4 }
```

Code Snippet 4: Quality Measure interface

The quality measure interface (Code Snippet 4) is trivial. It is designed to enable a simplified access to the quality of the solution the ant created. The quality must be returned as a float value and should be positive.

3.2 Output and data logging

One of the challenges of every middleware is an efficient data logging module. A badly designed one can slow down the system beyond usability, even when it is not in operation. Due to this reason we paid special attention to it, making sure that it would not render our middleware non-responsive under any circumstances.

The most adequate approach for handling a detached process, such as the logger, is the *short-circuit* design pattern. The short circuit pattern is a multithread software design pattern, which consists of splitting the threads into two groups: worker threads wt and handler threads ht. Worker threads produce results, which are later processed by handler threads. Depending on the computation load, the ratio of wt/ht must be adjusted. In our case the handler thread (the logger) is far less CPU-consuming. Thus, we have only one ht and an adjustable amount of wt. As we empirically established, in order to maximize CPU usage it is recommended to have $ht = 1$ and $wt = 2 \times cores - 1$, where $cores$ is the amount of physical CPU cores available.

The *LogWriter* interface (Figure 5), alongside its implementation of *AbstractDataLogger*, are the base elements of the logging module. They provide methods for logging all the primitive values, common Java collections, char strings and numerals. In the current version of the middleware three implementations of the LogWriter interface are provided:

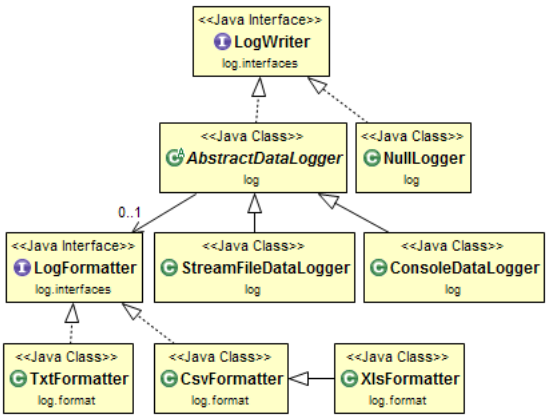


Figure 5: Logger structure

NullLogger (nullobject), *ConsoleDataLogger* (on-screen data display) and *StreamFileDataLogger* (saving data via FileStream).

LogWriter enables to select what data is to be logged, while the *LogFormatter* establishes the format of the data. Three classes of LogFormatter have been made available by default: *TxtFormatter*, *CsvFormatter* and *XlsFormatter*, which generate data compatible with *.txt* files, *.csv* files and *.xls* files respectively. This part of the system is, again, fully customizable.

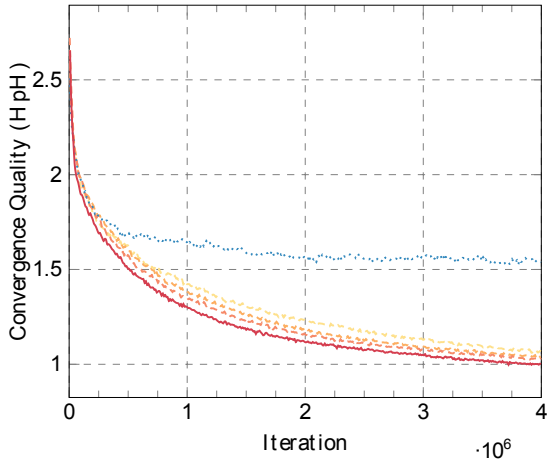
Another important aspect of the logging module is the data post-processing. Raw data from long executions can attain very large sizes, measured in hundreds of gigabytes. AntE offers an extensive spectrum of post-processing tools that include, but are not limited to: data sorting, data compacting by rolling average, merging independent executions, standard deviation extraction as well as lineal, non-lineal and arbitrary data bucketing.

In Figure 6 we present some of the possible outputs our middleware can produce. The linear plots (Figure 6a) are the most basic form of output. Upon indicating the source of the x-axis from the raw data file we can plot the evolution of any of the logged values or a combination thereof. In this case it is the classic hop per hit (*HpH*) measure in function of algorithm's iteration.

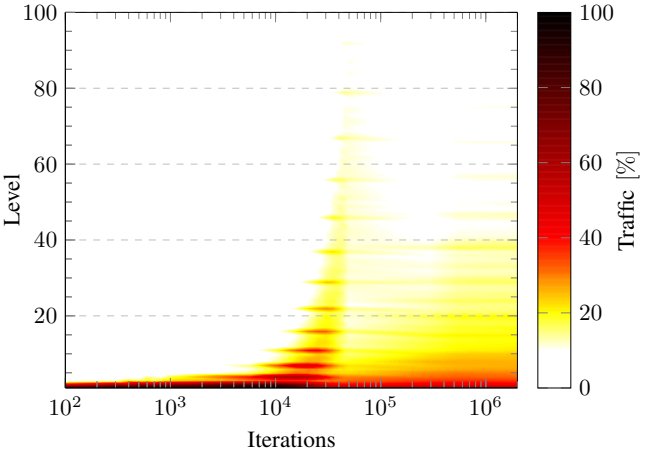
If data of higher dimensionality need to be visualized, the use of heatmaps is recommended (Figure 6b and Figure 6c). Our middleware can prepare the data for a visualization with the Ti*k*Z LaTeX package for traditional heatmaps (in the shown case - ant traffic density per pheromone level in a multi-pheromone implementation as a function of algorithm's iteration), or it can produce a custom *evolving* heatmap, which we called *DNA-heatmaps*. DNA-heatmaps are a gray-scale 2D variant of 3D plots, useful for plotting data comprised of several independent values that evolve in time. Figure 6c shows an evolution of the response quality (color intensity) per problem (y-axis) in time (x-axis). The scripts that produce DNA-heatmaps are bundled-in with our post-processing module.

3.3 Extensibility and Configuration

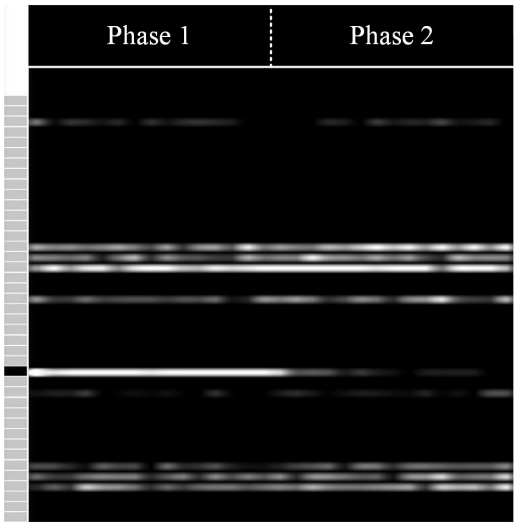
AntE achieves a high degree of customizability due to the widespread use of the strategy design pattern. In Code Snippet 5 we present a fragment of a configuration of the algorithm. Note lines (1), (2) and (5), where a class-encapsulated



(a) Linear plot (with Ti*k*Z LaTeX)



(b) Heat-map plot (with Ti*k*Z LaTeX)



(c) DNA-like heat-map (proprietary)

Figure 6: Examples of AntE logging outputs

behavior is passed as parameter. In line (4) we establish a numerical parameter and in line (7) a boolean parameter.

```
1 antConfig.setParameter(AntBehaviourConfig.ANT_ST,
      AcsStateTransition.class);
2 antConfig.setParameter(AntBehaviourConfig.SOLUTION_QM,
      AcsQualityMeasure.class);
3 antConfig.setParameter(AntBehaviourConfig.TTL_MAX, 8);
4
5 phConfig.setParameter(PheromoneConfig.PHEROMONE_COLLECTION,
      HashMapPheromone.class);
6
7 logConfig.setParameter(Logger.LOG_ENABLED, true);
```

Code Snippet 5: An example of configuration (1)

This configuration technique, coupled with the modular approach allows defining arbitrary behaviors. Consider the following example (Code Snippet 6). Our objective is to create a quality measure based on the square on the well-known hop per hit (HpH) metric. First we create a behavior class (line 1) as an extension of the predefined HpHQuality class, which encapsulates the calculation of the quality measure. Next, we introduce it into the configuration as the chosen behavior of the algorithm (line 12).

```
1  class SquareRootHpHQuality extends HpHQuality
       implements QualityMeasure
2  {
3    @Override
4    public float getQuality(Ant a)
5    {
6      float hphQuality = super.getQuality(a);
7      float squareRootHphQuality = Math.sqrt(hphQuality);
8      return squareRootHphQuality;
9    }
10 }
11
12 antConfig.setParameter(AntBehaviourConfig.SOLUTION_QM,
       SquareRootHpHQuality.class);
```

Code Snippet 6: An example of configuration (2) - behavior class definition

Note that the change of configuration can be done during the execution of the algorithm, allowing interesting and unusual observations and experiments.

3.4 Low Level Optimization Techniques

The choice between the programming languages in which to write the middleware is a trade-off. The main benefits of Java are obvious: portability and hardware and software abstraction. Java, however, produces slower code than the corresponding solutions in C++. To counteract this we were forced to design and apply a series of low level optimization techniques. Here we list some of the most notable ones.

The most basic step is not to permit Java to perform, so called, boxing and unboxing of numerical values. In Java, numerical values are both object-based and primitives. Boxing and unboxing are the names given to the conversions between the two types. The conversions are, typically, transparent from the programmer's point of view and unnoticeably quick. However, in our case they were very frequent and consumed an important portion of the CPU time.

This prompted us to eliminate completely one of the two types of the numerals. We decided to exclude the object-based numerical values from our code, which benefited us thrice. First, objects are larger in terms of memory than their primitive counterparts (64 bytes versus 2 - 8 bytes). Second, as mentioned, the unboxing is avoided completely. And third, the creation and destruction of objects and, in consequence, the garbage collector usage are reduced significantly.

As a result, we were forced to rewrite the generic Java collections, such as *ArrayList* and *HashSet* (which use object-based numerals) into their corresponding, primitive-based counterparts. This was an opportunity to incorporate in the newly created classes (*PrimitiveIntArrayList* and *PrimitiveFloatArrayList*) low-level methods, which accelerated common operations, such as *max*, *min*, *sum* etc. This change alone helped to reduce the memory consumption by a margin of 75% and the CPU consumption by 90%.

The biggest challenge in ACO related middlewares is the pheromone container. It is very often accessed and it is read and updated with similar frequency, which excludes a write- or read-focus optimization. Our solution, alongside the aforementioned primitive-based computation, was to handle the pheromone writes and reads in batches, via the low level method *System.arraycopy* (Code Snippet 7, from *HashMapPheromone* class). In order to benefit from this technique, the code must be redesigned with batch operations in mind. The difference of the execution time of the batch-approach with respect to the value-by-value-approach is above 71% in favor of the former.

```
1  public void setAll(float[] pheromone)
2  {
3    System.arraycopy(pheromone, 0, elementData, 0,
         pheromone.length);
4    size = pheromone.length;
5  }
6
7  public float[] getAll()
8  {
9    float[] pheromone = new float[size];
10   System.arraycopy(elementData, 0, pheromone, 0, size);
11
12   return pheromone;
13 }
```

Code Snippet 7: Batch operations

Another common efficiency bottleneck is the method that establishes if a given element is present or absent from a data collection, typically called *.contains*. It is, at best, of O(log(n)) complexity. Quite often however, as in the case of unsorted lists, it is O(n). We designed a technique which guarantees a numerical complexity of O(1). Our method is possible if the elements stored in the collection are labeled with an index, and it is especially effective if the range of the values of the indices is known, limited and relatively small.

First we instantiate a bit array of size *max_index*. Each time an element is inserted into the collection, the field in the bit array that corresponds to the index of the inserted element is set to *true*. The opposite happens when the last element of a given index is removed. This way the *.contains* check is reduced to reading the bit corresponding to the examined element. The memory increase is modest. For max_index of 32768 the bit array only uses $4kB$ of memory. See Code Snippet 8 for a simplified example. This approach has reduced the overall execution time by 16.7%. We believe

Table 2: Execution Times, Intel Xeon X3430

Worker Threads	1	2	3	4	5	6	7
Logger Threads	1	1	1	1	1	1	1
Physical Cores	2	2	2	2	2	2	2
Time [s]	8.4	6.7	6.4	6.9	7.0	7.3	8.0
Ants$\times s^{-1}$ [$\times 10^4$]	11.9	14.9	15.6	14.3	14.3	13.6	12.5
Ant-steps$\times s^{-1}$ [$\times 10^6$]	2.4	2.9	3.1	2.8	2.8	2.7	2.4

Table 3: Memory Usage, Intel Core i5 540m

Graph size	256	1024	4096	16384	65536
Total memory [MB]	2.6	9.5	37.1	147.4	588.8
Memory per node [kB]	10.2	9.3	9.0	8.9	8.9

that this technique has a wide range of uses, well beyond our middleware.

```
1  class MonitoredCollection<T> extends Collection<T>
2  {
3  private BitSet cMonitor = new BitSet(max_index);
4
5  (...)
6
7  @Override
8  public void add(int index, T data)
9  {
10   super.add(index, data);
11   cMonitor.set(index);
12 }
13
14 @Override
15 public void remove(int index)
16 {
17   super.remove(index);
18   cMonitor.clear(index);
19 }
20
21 @Override
22 public boolean contains(int index)
23 {
24   return cMonitor.get(index);
25 }
26
27 (...)
```

Code Snippet 8: *contains* operation

4. EFFICIENCY AND SCALABILITY

Having described some of the most notable elements of our implementation we proceed to the analysis of the efficiency and scalability. We start with, arguably, the most crucial parameter, which is the overall execution time. We executed our middleware under the following conditions: a graph with 1024 nodes with 10^6 ants released simultaneously onto it. Each ant performs a full search of $TTL = 20$ steps and saves the search results in a file. The hardware configuration is: Intel Xeon X3430 (8MB Cache, 2.40 GHz), limited to 2 of the 4 physical cores and 2GB Ram. In Table 2 we present the execution times in function of processing threads used. We conclude that in the best case of 3 worker threads, the task is terminated in under 6.5 seconds. At the peak, the efficiency of ant processing was about 15.6×10^4 ants per second and 3.1×10^6 *ant-steps* per second. By ant-step we understand a full processing of one ant in one node. It includes: the resource query, the pheromone read and write as well as the state transition.

The same experiment on a far slower Intel Core i5 540m takes roughly 43 seconds. In both cases the CPU resources consumption reached 98% per core on average, which demonstrates an above-average effectiveness of our short-circuit approach. This suggests that the CPU-related scalability of AntE is high, allowing to benefit from multicore processors in a satisfying degree.

Another common limitation of other existing middlewares is the size of the graph they can produce and process. Classical graph-related problems tend to be rather small, mostly under 10^4 nodes. This means that a trivial (non-memory efficient) implementation would possibly suffice. Still, we attempted to reduce the memory consumption to a relative minimum, keeping in mind that the shorter CPU time, within reasonable bounds, is favored. In Table 3 we summarize the memory consumption in function of graph size. It can be observed that the property is very weakly sublineal, and therefore, scalable. On average one node occupies $8.9kB$ of memory. Under these conditions, with a 32-bit version of Java the upper limit of the graph size is situated at around 10^5 nodes, on a 64-bit version it is virtually unlimited.

The stability of the middleware is also quite important. In our extensive experiments we were able to sustain a constant, uninterrupted flow of ants during 12 hours, generating roughly $7GB$ of log files. We claim that, with AntE, experiments of the order of magnitude of 10^{10} ants are feasible.

5. EXISTING IMPLEMENTATIONS

We have used our middleware in several experimental setups and one possible commercial deployment.

The early versions of the software have been used to examine ACO algorithms under the condition of the dynamism of the problem space [8]. This is a largely unexplored area of ACO, however, it was essential in our study of the applicability of ACO to P2P networks. The unique property of supporting modifications of the graph while the algorithm is under execution allowed us to observe insufficiencies of ACO in this regard [9].

After having defined and demonstrated the problem, a proposal of a P2P-compatible ACO was deployed as an experimental module, formulated completely within AntE [7]. We opted for counteracting the problems associated to the dynamism of the graph by correcting the pheromone paths with the help of a new type of ants, the *graph structure diffusion* ants. The coding effort involved in the aforementioned line of research, albeit seemingly complex, was in fact minimal and could have been completed by researchers with limited knowledge of programming. In this line of research the graph sizes ranged between 1024 and 32768 and the experiment lengths were of 10^5 iterations.

Our middleware has also been used to implement a deployable piece of software that served as a recommendation service for rehabilitation tasks for people with Acquired Brain Injuries [10]. This extensive software can be used as both: an experimental module and a deployable application. In its essence it was an example of a practical application of ACO algorithms. Here, we treated the graph nodes as rehabilitation tasks, the pheromone as a similarity measure between them, and the ants as a representation of a query issued for

a patient. All of the changes were possible strictly within what the configuration of our middleware permits.

Our most recent research is centered on the question of resolving multi-class queries in P2P networks with ACO. We extended our software with multi-pheromone capacities in order to experiment with the effects of multiple types of ants and how they impact the overall efficiency of ACO. We also increased the length of the experiments to 4×10^6 iterations. Working with our middleware enabled us to put the formulated ideas to a test quickly and obtain a very early experimental validation, which, in turn, was a way to avoid or abandon unpromising concepts.

6. CONCLUSIONS AND FUTURE WORKS

In this paper we presented a customizable ACO middleware, AntE. We elaborated on the novelty of our software, as well as outlined some of its more interesting features. We discussed its performance and scalability, and described its architectural design alongside some selected optimization techniques, which, we hope, will benefit the ACO community.

In the direct future we will focus on the release of our software as open source and we will bundle it with a documentation allowing users to use it to its full potential. In addition, we continually rework and update our source code, improving safety, stability, as well as the key aspects: scalability, speed and customizability. In this respect, we are currently considering new and promising ACO-hibridization techniques that will be included as optional components of the next release of the AntE software infrastructure.

We would also like to compare the effectiveness of our approach with general purpose environments, such as *HeuristicLab* [15]. HeuristicsLab has been in development since 2002 and it has become a very extensive and configurable environment for broadly understood algorithmic experimentation, with a modern GUI, visual algorithm and experiment designers, as well as analytical tools. Due the scope of the software, HeuristicsLab bares no characteristic of a middleware, but is, in fact, a meta-level environment. Therefore, we argue that AntE should not be perceived as a competing, but rather complementary approach.

A possible continuation of our work could focus on the conversion of AntE into a plug-in of HeuristicsLab. This would enable us to benefit from the powerful framework HeuristicsLab is. Such a plug-in, supporting multiple pheromone levels, graph structure diffusion ants and resolution of multi-class queries in P2P networks with ACO, has, to our best knowledge, not been implemented.

7. ACKNOWLEDGMENTS

Kamil Krynicki is a FPI fellow of Universitat Politècnica de València, number 3117. This work received support from the Ministry of Science and Innovation under the National Strategic Program of Research, Project TIN2010-20488 and TIN2014-60077-R, as well as the National Institute of Informatics, Tokyo, Japan.

8. REFERENCES

[1] O. Babaoglu, H. Meling, and A. Montresor. Anthill: a Framework for the Development of Agent-Based Peer-to-Peer Systemse, 2001.

[2] F. E. Barril Otero. Myra, 2008. Available at: http://sourceforge.net/projects/myra.

[3] C. Blum. hc-mmas-ubqp, 2004. Available at: http://iridia.ulb.ac.be/ mdorigo/ACO/downloads/hc-mmas-ubqp.tar.gz.

[4] U. Chirico. JACSF. 2009. Available at: http://www.ugosweb.com/Documents/jacs.aspx.

[5] F. J. Diego Martín, J. A. González Manteca, R. Carrasco-Gallego, and J. Carrasco Arias. AntLib v1.0: A generic C++ framework for ant colony optimization. In *Ant Colony Optimization and Swarm Intelligence*, volume 5217 LNCS, pages 397–398, 2008.

[6] M. Dorigo, M. Birattari, and T. Stützle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1, 2006.

[7] K. Krynicki, J. Jaen, and A. Catala. A diffusion-based ACO resource discovery framework for dynamic p2p networks. In *2013 IEEE Congress on Evolutionary Computation*, pages 860–867. IEEE, June 2013.

[8] K. Krynicki, J. Jaen, and J. A. Mocholí. On the performance of ACO-based methods in p2p resource discovery. *Applied Soft Computing Journal*, 13:4813–4831, 2013.

[9] K. Krynicki, J. Jaen, and J. A. Mocholí. Ant colony optimisation for resource searching in dynamic peer-to-peer grids. *International Journal of Bio-Inspired Computation*, 6(3):153, 2014.

[10] K. Krynicki, J. Jaen, and E. Navarro. An aco-based personalized learning technique in support of people with acquired brain injury. Under Review.

[11] F. Meyer and R. Stubs Parpinelli. GUIAnt-Miner, 2002. Available at: http://sourceforge.net/projects/guiantminer/.

[12] E. Michlmayr, A. Pany, and G. Kappel. Using taxonomies for content-based routing with ants. *Computer Networks*, 51:4514–4528, 2007.

[13] C. Solnon. AntClique, 2006. Available at: http://liris.cnrs.fr/csolnon/AntClique.html.

[14] T. Stützle and A. Wilke. ACOTSP, 2004. Available at: http://iridia.ulb.ac.be/ mdorigo/ACO/downloads/ACOTSP-1.03.tgz.

[15] S. Wagner and G. Kronberger. Algorithm and experiment design with heuristic lab: an open source optimization environment for research and education. In *GECCO '12 Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 1287–1316, 2012.