CODEA - An Agent Based Multi-Objective Optimization Framework

Juan Castro-Gutierrez
ASAP Research Group
School of Computer Science
University of Nottingham
jpc@cs.nott.ac.uk

Dario Landa-Silva
ASAP Research Group
School of Computer Science
University of Nottingham
dario.landasilva@nottingham.ac.uk

José A. Moreno-Pérez
DEIOC
Facultad de Matemáticas
Universidad de La Laguna
jamoreno@ull.es

Abstract

This work presents CODEA, a COoperative DEcentralized Architecture for Multi-objective Optimization. CODEA is an object-oriented framework that aims at the creation of groups of agents to tackle complex problems by cooperative search. This cooperation is carried out without any individual controlling the cooperation nor the behaviour of the agents. Each agent works on its own to improve itself and collaborates to improve the performance of the group by sharing information.

1 Introduction

Heuristic algorithms have become more elaborate in recent years. This means that the time required for the design, debugging and testing of this kind of methods is considerable. Source code re-utilisation emerges as a means to re-use already coded algorithms to solve different problems without starting a new development from scratch [3].

In this work, we propose a new version of CODEA (COoperative DEcentralized Architecture), first proposed in [6]. This framework is designed to create flexible groups of agents to tackle multi-objective optimisation problems using the paradigm of cooperation. CODEA consists of a number of classes developed in C++ that accelerate the development of cooperative metaheuristics with no centralised control mechanism. This new version incorporates the following main features: support for tackling multi-objective optimisation problems, various ranking schemes to discriminate solutions in multi-objective optimisation (aggregation approach, lexicographic ordering and Pareto dominance), parallelisation supported by OpenMP, etc. We illustrate the use of CODEA by implementing a Discrete Particle Swarm Optimization (DPSO) algorithm to tackle a set of instances of the bi-objective Travelling Salesman Problem (TSP). A number of experiments are conducted to test different communication topologies within the proposed framework.

A number of frameworks and libraries have been proposed to facilitate the development of heuristic algorithms in different programming languages. In particular, for multi-objective optimization some well known frameworks and libraries are: *Paradiseo* [12], *Open Beagle* [5], *PISA* [1], *MOMHLib++* [13] and *TEA* [4]. All these open-source tools are developed in C++ and include a number of features to work with multi-objective problems.

Paradiseo is probably the most elaborated framework. It comes with a number of implemented metaheuristics (evolutionary algorithms, local search, simulated annealing, etc.) that are totally customisable. Moreover, it includes metrics to assess the quality of solutions, methods for ranking solutions, parallelisation mechanisms, etc. Paradiseo also provides with a number of tutorials and examples implementing different methods and problems. The drawback of working with such a framework is its inherent complexity. For rel-

atively simple projects, the use of Paradiseo could severely slow down the implementation process (learning curve) and the performance of the algorithm. This is due to the complex class hierarchy it uses to model the system.

Open Beagle is a high-level evolutionary computation framework. Like Paradiseo, Open Beagle is portable, flexible and robust. It has a good documentation but its last update was in 2007.

PISA (Platform and Programming Language Independent Interface for Search Algorithms) like Paradiseo, includes the implementation of various well-known algorithms (SPEA2, NSGA-II) and problems (ZDT, DTLZ), plus a number of metrics to assess the quality of solutions. As a drawback, PISA seems to be a black box that is not flexible to be extended. As it merely provides a scripting language for search algorithms and ready-to-go modules as binary files.

MOMHLib++ (Multiple Objective Meta-Heuristics Library in C++) is a library that implements a number of multi-objective metaheuristics (PSA, MOGLS). It also provides metrics to measure the quality of solutions and seems to be easy to learn. However, it has not been updated since 2005 and it does not have documentation.

TEA (Toolbox for Evolutionary Algorithms) is a library for the design of evolutionary algorithms (EA). It is a very flexible framework with many features to build up EAs, enhancing the ease of use and avoiding unnecessary abstractions. Some of these features are: use of complex genotypes, on-line exchange of operators, multi-population, etc.

The above systems are a representative sample of existing frameworks that support the development of multi-objective optimisation algorithms. However, many other frameworks exist to deal with single-objective problems. For instance, SATenstein [11] is a highly configurable framework to tackle SAT problems. It consists of a mash-up of high performance algorithms for SAT problems, combined with a number of selection parameter to chose which components are used in the optimization process.

Moreover, an overview on optimization software class libraries for a variety of problem+s can be found in [14].

The reminder of this paper is organised as follows. Section 2 provides some basic concepts on the search techniques and subject problem considered in this work. Section 3 gives a detailed description of CODEA and each component in this framework. A test-case of a DPSO for the bi-objective TSP is detailed in Section 4. The experiments are described in Section 5 while results are presented and discussed in Section 6. Finally, some conclusions and future work are given in Section 7.

2 Background

In order to make this work self-contained, a number of concepts are explained in this section.

2.1 Multi-objective Optimization (MOO)

In MOO, we aim to solve a problem of the type: *minimize* $\vec{f}(\vec{x}) = f_1(\vec{x}),\ f_2(\vec{x}),...,f_k(\vec{x})$, subject to: $g_i(\vec{x}) \leq 0, i = 1,2,...,m$ and $h_j(\vec{x}) = 0,\ \ j = 1,2,...,p$. Where the decision variable vector is $\vec{x} = [x_1, x_2, ..., x_n]^T$, each objective function f_i is defined in $\Re^n \rightarrow \Re, i = 1,2,...,k$ and the constraint functions are g_i and $h_i, i = 1,2,...,m, j = 1,2,...,p$ which are defined in the same domain as f_i.

Without loss of generality, we consider a minimisation problem. A vector $\vec{u}$ is said to *dominate* $\vec{v}$ (denoted by $\vec{u} \prec \vec{v}$), iif $\forall i \in (1,...,k) : u_i \leq v_i \wedge \exists i \in (1,...,k) : u_i < v_i$. Moreover, we say that a vector in the feasible region ($\vec{u} \in \mathcal{F}$) is *Pareto Optimal*, if there is not any other vector in that region ($\vec{u'} \in \mathcal{F}$), such that $\vec{u'}$ *dominates* $\vec{u}$ ($\vec{u'} \prec \vec{u}$). Then, the *Pareto Front* is the set of vectors in $\Re^n$, such that all elements in the set are *Pareto Optimal*.

2.2 Travelling Salesman Problem (TSP)

TSP is probably the most well-known combinatorial optimisation problem. Given a number of cities/costumers and their pairwise distances, the goal is to find the shortest tour that visits all the costumers only once. Formally,

the problem consists of finding the permutation $P = \{c_0, c_1, ..., c_{N-1}\}$ such that $d_P = \sum_{i=0}^{N-1} d[c_i, c_{i+1 mod(N)}]$ is minimum. This is an NP-complete problem particularly important in the fields of planning and logistics. Various formulations and solving methods for the TSP can be found in [7].

2.3 Discrete Particle Swarm Optimization

Particle Swarm Optimization is a nature-inspired algorithm proposed by Kennedy and Eberhart [9]. This algorithm was motivated by the social behaviour of bird flocking and fish schooling. It was presented to tackle continuous optimisation problems by using two equations, one to update the velocity and another to update the position. In 1997, Kennedy and Eberhart [10] presented the first discrete version of this algorithm (DPSO). Since then, a wide variety of versions for the DPSO have been designed to deal with discrete search spaces.

Here we test CODEA by implementing a DPSO based on the work of Consoli et at. [2]. This approach drops the concept of velocity in order to redefine how particles move in a discrete search space. Thus, the movement of particles is modelled using a follower-attractor scheme. Each moving particle performs only one type of move at each generation. The type of move is randomly selected out of four possible types as in the original PSO: 1) move respect to the previous position, 2) move towards its best position achieved, 3) move towards the best positioned neighbour and 4) move towards the best position in the swarm so far. Here, these moves are interpreted as genetic operators applied on particles' positions (solutions). In case of the first type of move, a mutation operator is applied on the current position (solution) of the particle. For the other three types, a crossover operator is triggered so that the moving particle imitates part of the structure in the attractor's solution.

This methodology aims to help the swarm to evolve by copying pieces of structures from the best positioned particles (solutions) at each generation into the rest of the swarm.

3 CODEA

CODEA (COoperative DEcentralised Architecture) is a library of classes to build up systems of agents who cooperate to tackle complex problems. Since the system is decentralised, there is no entity controlling what other agents do. That is, agents are free to carry out different phases of the search with no limitations. Each individual can perform its own operations, to send whatever information it wants, delivering this information to any agent it considers appropriate.

CODEA was mainly developed at the University of La Laguna (Spain) by the Group of Intelligent Computing (GCI). The core idea of the project was to create a simple, flexible and fast framework to enhance an agent-based system to emerge an intelligent behaviour. To this aim, we designed CODEA using a number of modules with abstract functionalities. These functionalities can be easily extended by the user without a deep knowledge on how the core of CODEA works.

Figure 1 shows the basic scheme of CODEA. Taking a closer look into this figure, we see an element called *System* at the top of the structure. This element contains a number of agents and some properties. This class stores the data structure that hosts the agents and orchestrates the operations the agents perform. It is worth noting that even if *System* holds the group of agents, it does not know what the agents do (operations), nor the relation (neighbourhood) among them. *System* is merely intended to store general properties like up-time (*elapsedTime*) and the iteration (*iteration*) at which the best solution was found, the best solution found (*bestSoluion*) by the agents, and the criterion used to stop the execution of the main-loop in which agents run operations (*phases*) in class agent.

The next element (*agent*) in the figure represents the agent as a general individual who is able to communicate and operate on its own. This cell shows the sub-elements: *neighborhood*, *core* and *phases*. These components

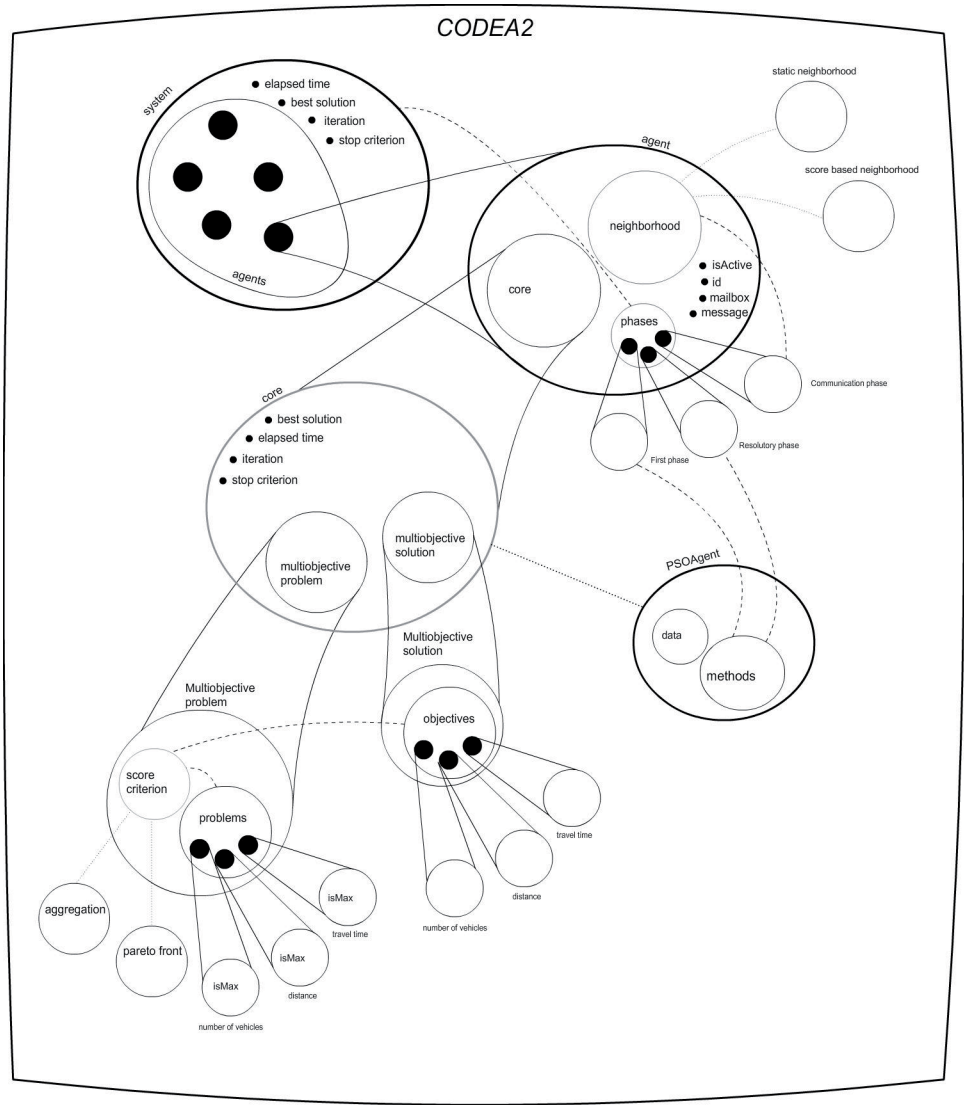

Figure 1: Diagram of CODEA implementing a MOPSO

give agents special abilities to share information (neighborhood), to hold the information to solve problems (core), and the way to carry out their operations (*phase*). An agent also has a number of parameters to control its state (isActive, id, etc.).

The *neighborhood* component manages the list of population members that this agent communicates with. In this way, *neighborhood* is implemented as an interface so the user can develop its own rules to establish new neighborhoods. By default, CODEA allows to use three types of neighborhoods: star topology (all to all communication), ring topology (each individual receives and sends from/to two other individuals) and k-random (each individual shares its information with k random individuals) topology. Although these topologies are static, CODEA does not limit the creation of dynamic systems of cooperation. It is fairly easy to implement dynamic cooperation schemes based on scores or rules.

The *core* element is in charge of manipulating the information of the problem, its solution and solution process. In order to re-utilise the code as much as possible, the *core* has three main components: *Multiobjective problem*, *Multiobjective solution* and solver, which in figure 1 is represented by *PSOAgent*. By using this methodology, changing the solution or the problem does not affect the rest of the structure. For example, we might want to change the manner we evaluate solutions, but without modifying the solutions themselves. Or just the opposite, keep the evaluators unchanged, while the solutions have a different encoding. The same is designed to happen with the solver. The way the solver is implemented should not depend on the solution encoding nor on the data structure of the problem. For this purpose, both *multiObjectiveProblem* and *multiObjectiveSolution* have an independent design. The former component has two parts: an score criterion and a number of function evaluators. These function evaluators are related to the problem, in the figure there are three evaluators: *number of vehicles*, *distance* and *travel time*. The *score criterion* is another interface that enables the user to create criteria to rank solutions in a multi-objective scenario. So far, we have implemented: Pareto dominance, aggregation and lexicographic ordering. The *multiObjectiveProblem* also holds the data of the problem and the set of evaluation functions.

On the other hand, for each objective that the agents are optimising, there is an associated function to calculate its value. This function is represented in the structure as a simple file that specifies how to assess a certain objective. In the example shown in the figure, there are three files: *number of vehicles*, *distance* and *travel time*. In this way, adding a new objective is as easy as creating a new file describing how to compute that objective. Moreover, this philosophy provides the means for enabling and disabling objectives on line. Regarding the score criterion, one may consider that, since the system compares two solutions, this sub-component should be contained within *multiObjectiveSolution* rather than within *multiObjectiveProblem*. However, it must be noted that many ranking schemes require to know whether we are maximising or minimising objectives. This characteristic is inherent to the evaluation of objective function and therefore, must be placed within *multiObjectiveProblem*. Regarding *multiObjectiveSolution*, this element has two basic units: a vector of objective values and the subjacent solution. The latter unit simply stores the solution using the encoding provided according the problem and the user preference. In the case of the TSP, the subjacent solution might be a class containing a vector with the tour of cities. The evaluation of this solution in the set of objectives described in *multiObjectiveProblem* is always saved to the vector of objectives. Thus, the system avoids to re-evaluate all the objective functions every time we want to compare two solutions. Both *multiObjectiveSolution* and *multiObjectiveProblem* are handled by the component *solver*. This part of the *core* contains atomic operations creating an abstraction of the behaviour of the agent. For example, in order to develop the Multi-objective Particle Swarm Optimisation (MOPSO), we designed a class that coordi-

nates the operations without direct relation to the problem or the solution being used. Beneath in the structure, is where we make the connection to our problem and solution design. In this way, the core of the solver does not depend on the problem tackled by the system.

The atomic operations contained in the *core* are orchestrated by the *phase* component within the agent class. For the TSP, these operations are for example: mutation and crossover operators. This part of the agent acts as an interface for the user to provide a standard to implement his/her own phases. The flow of CODEA starts with the system invoking the agent's phases. Then, the agents take the control coordinating their operations using the phases. There is not a limited number of phases and they do not have to be synchronous (all agents doing the same task simultaneously). In addition, an agent is able to delete, add or modify phases in real time.

4 Implementing PSO and MOTSP in CODEA

For this test case, we implemented in CODEA a DPSO based on the one outlined in Section 2.3 to tackle the bi-objective travelling salesman problem (TSP). In simple terms, agents within CODEA have two phases: a *Resolutory Phase* and a *Communication Phase*. The first phase is in charge of specifying how the DPSO agents should work, and the second phase will merely run the delivery of messages (*solutions*) according to the neighbourhood structure specified by the user. However, this second phase is implemented in CODEA by default, so we do not have to implement anything here.

We should mention that for this test case, we use *simulated parallelism*. That is, if n agents perform m phases, the system first triggers the loop for the phases ($i = 0, ...n-1$) and then, the one for the agents ($j = 0, ..., m-1$).

Since CODEA makes a distinction between the problem and the solving method, we split the explanation for each in two subsections: *Solving Method* and *Problem*.

4.1 Solving Method

In other to implement the DPSO, we create the following files:

- DPSOResolutorPhase (inherits from *phases*). This class models the behavior of the DPSO agent. It contains the sequence of operations defined in a problem-independent manner. Basically, this class contains a method (*core*) that randomly chooses an attractor. A virtual method (defined in *DPSOAgent*) is then invoked with the moving particle's and attractor's positions (solutions).

- DPSOAgent (inherits from *agent*). This class contains four data members that hold the probability for each type of move (see section 2.3) and four pure virtual methods. These virtual methods must be inherited for non-abstract classes in order to connect the agents' abstract behaviour with the problem-dependent operators.

4.2 Problem

Regarding the implementation of the problem, we develop the following files:

- TSPDataProblem. This class acts as the container of data for each instance. In our case, it contains a vector of matrices, one per objective. That is, if we work with two objectives, this class will hold two matrices of costs.

- TSPOperatorsLib. In this library, we develop a number of functions (operators) to work with the travel-plans. For this test case, we implement a mutation operator, a crossover operator and a local search. The former will swap pairs of randomly selected costumers *sqrt* times the size of the travel-plan. The crossover operator will create an offspring (travel-plan) out of the interchange of a random random section of the parents. An finally, the local search will explore the neighborhood of each travel-plan by swapping pairs of costumers iteratively.

- TSPSolution (inherits from *solution*). This class holds the structure of the travel-plan. In this case, a travel-plan is encoded as a simple vector, in which each position corresponds to a costumer id. It also contains general purpose methods to print, set and get the route-plan.
- matrixSumObjective. Since the *TSPDataProblem* has a number of matrices with the same structure, we implement just one objective function that is re-used for all objectives.

In order to connect the behavior of each agent (methods in *DPSOAgent*) and the operators (functions in *TSPOperatorsLib*), we create a class (*TSPDPSOAgent*). This class (inheriting from *DPSOAgent*) is in charge of casting solutions from mother classes. In this wya, it receives solutions from upper classes, casts them to get the travel-plan and sends them to *TSPOperatorsLib* with the pertinent parameters. Using this approach, only one class acts as an interface connecting the solving method and the problem. This makes it possible to change the method without changing the solving method and viceversa.

5 Experiments

In order to test our implementation, we evaluate the performance of three communication topologies on a subset of Hui Li's MOTSP instance-set [8]. We used 20 bi-objective instances, half of them have 50 customers and the other half have 100.

The three communication topologies set are: star (all to all communication), ring (each agent sends and receives messages only from two neighbours) and k-random (each agent sends to k random neighbours). In order to discriminate solutions when updating and selecting the leader, CODEA was set to use Pareto dominance. That is, a solution x is preferred over a solution y, if x Pareto dominates solution y. For these experiments, we used a swarm of 50 particles (agents) and let them evolve for 2000 generations (iterations). For each problem instance, we ran the algorithm using three different seeds for each topology in order to build the approximation set for each topology. We assess the performance of each of the three topologies proposed according to the S-metric [15], a widely used performance measure that computes the hyper-volume covered by the approximation set of non-dominated solutions and a reference point.

6 Results

Table 1 depicts the results of our experiments. This table is divided in two sections, the top half shows results for instances with 50 customers, and the second half for instances with 100 cities. For each row, the first column gives the instance name. The other three columns show the normalised value of the S-metric calculated from the approximation sets obtained for each topology.

It can be observed that the best performing communication structure is, on average, the K-random topology. This communication topology seems to work better with larger instances, as it gets the best S-metric value in almost all the 100 costumers instances. It should also be noted that the worst performance is that of the Ring topology, but with a very small distance: 0.09 on average. Bearing in mind that this is the least expensive strategy, as only two messages must be sent by each agent, its performance is quite acceptable. Although the Star topology gets the second best result, it does not seem to provide a good trade-off between computational cost and performance respect to the S-metric.

Wilcoxon signed-rank test is often used to test the difference between scores of data collected out of experiments. We use this test to compare the significance in the performance improvement observed in each method. We obtain that K-random is better than Ring at 95.49% of confidence level (W sample value = 138), K-random is better that Star at 96.41% of confidence level (W sample value = 97) and that Ring is better than Star at 96.41% of confidence level (W sample value = 97).

Table 1: Performance, measured with the S-metric, of three communication topologies used in a DPSO approach implemented in CODEA when applied to some instances of the bi-objective TSP.

Instance	Star	Ring	K-Rand
krobc50	**1.00**	0.82	0.97
kroac50	**1.00**	0.95	0.98
krode50	0.93	0.92	**1.00**
krocd50	0.98	0.94	**1.00**
kroce50	0.92	**1.00**	0.84
kroab50	0.90	**1.00**	0.94
kroad50	**1.00**	0.94	0.98
kroae50	0.99	**1.00**	0.97
krobd50	**1.00**	0.87	0.97
krobe50	0.91	0.98	**1.00**
kroad100	0.80	0.65	**1.00**
kroae100	0.99	0.76	**1.00**
krobd100	**1.00**	0.79	0.73
krobe100	0.90	0.81	**1.00**
kroab100	0.76	0.88	**1.00**
krode100	0.94	0.91	**1.00**
kroac100	0.93	0.68	**1.00**
krobc100	0.92	**1.00**	0.96
krocd100	0.89	0.87	**1.00**
kroce100	0.82	0.74	**1.00**
Average	0.93	0.88	**0.97**

7 Conclusions

A new version of CODEA, a COoperative DEcentralised Architecture for implementing multi-agent systems, incorporating a number of new futures to support tackling multi-objective optimisation problems has been presented in this work.

As an illustrative test case, we explained how to implement a DPSO (Discrete Particle Optimisation) approach to tackle a set of instances of the bi-objective TSP. The implementation was tested comparing three communication topologies within the DPSO. The best strategy to send and receive information, measured with respect to the S-metric, turned out to be the K-random topology. However, the winner with respect to a good compromise between cost of sharing information and quality of solutions was the Ring topology, which gets the second best results according to this metric using the least expensive type of message passing.

Currently, our main focus in the development of CODEA is on the implementation of multi-objective problems and algorithms to solve them. In the short term, we also plan to release a free beta version of CODEA for others to test our algorithms/problems and contribute to the implementation of new ones.

References

[1] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler, *Pisa - a platform and programming language independent interface for search algorithms*, Proceedings of the 2nd international conference on evolutionary multi-criterion optimization (EMO2003), LNCS, vol. 2632, Springer, 2003, pp. 494–508.

[2] S. Consoli, J.A. Moreno-Pérez, K. Darby-Dowman, and N. Mladenović, *Discrete particle swarm optimization for the minimum labelling steiner tree problem*, Natural Computing **9** (2010), no. 1, 29–46.

[3] I. García del Amo, F. García López, M. García Torres, B. Melián Batista, J. Moreno Pérez, and J. Moreno Vega, *From theory to implementation: Applying metaheuristics.*, ch. 11, pp. 311–351, Springer, 2006.

[4] M. Emmerich and R. Hosenberg, *TEA - a toolbox for the design of parallel evolutionary algorithms in c++*, Tech. Report CI-106/01, SFB 531, University of Dortmund, Germany, 2001.

[5] C. Gagné and M. Parizeau, *Genericity in evolutionary computation software tools: Principles and case study*, International Journal on Artificial Intelligence Tools (IJAIT) **15** (2006), no. 2, 173–194.

[6] Juan Gutiérrez, Belén Batista, José Pérez, J. Vega, and Jonatan Bonilla,

CODEA: an architecture for designing nature-inspired cooperative decentralized heuristics, Nature Inspired Cooperative Strategies for Optimization (NICSO 2007), Studies in Computational Intelligence, vol. 129, Springer, 2008, pp. 189–198.

[7] G. Gutin and A. Punnen, *The traveling salesman problem and its variations*, Springer, 2004.

[8] Hui Li, *Hui's TSP instance-set*, http://www.cs.nott.ac.uk/, 2010.

[9] J. Kennedy and R. Eberhart, *Particle swarm optimization*, IEEE International Conference on Neural Networks - Conference Proceedings, vol. 4, 1995, pp. 1942–1948.

[10] ———, *Discrete binary version of the particle swarm algorithm*, Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, vol. 5, 1997, pp. 4104–4108.

[11] R. KhudaBukhsh, X. Lin, H. Holger, and K. Leyton-Brown, *Satenstein: automatically building local search sat solvers from components*, Twenty-first International Joint Conference on Artificial Intelligence (IJCAI-09), 2009, pp. 517–524.

[12] A. Liefooghe, L. Jourdan, T. Legrand, J. Humeau, and E.-G. Talbi, *ParadisEO-MOEO: a software framework for evolutionary multi-objective optimization*, Studies in Computational Intelligence, vol. 272/2010, pp. 87–117, Springer, 2010.

[13] MOMH Team, *MOMHLib++*, http://home.gna.org/momh/index.html, 2010.

[14] S. Voss and D. Woodruff (Eds.), *Optimization software class libraries*, Kluwer academic publishers, 2002.

[15] E. Zitzler, D. Brockhoff, and L. Thiele, *The hypervolume indicator revisited: On the design of pareto-compliant indicators via weighted integration*, LNCS, vol. 4403, pp. 862–876, Springer, 2007.