**ANT COLONY OPTIMISATION FOR SOLVING CONTINOUS SPACE PROBLEMS**

**BY**

**MC-NIEL NZUBECHI CHINEDU**

**MATRIC NO: 169074008**

**A PROJECT SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENT FOR THE AWARD OF MASTERS OF SCIENCE (M.SC) DEGREE IN COMPUTER SCIENCE, SCHOOL OF POSTGRADUATE STUDIES, UNIVERSITY OF LAGOS, AKOKA, NIGERIA**

**DECEMBER, 2021**

# DECLARATION

I, **MC-NIEL NZUBECHI CHINEDU**, hereby declare that this research work in " **ANT COLONY OPTIMISATION FOR SOLVING CONTINOUS SPACE PROBLEMS** " was solely carried out by me under the supervision of **Dr. B. A. SAWYERR**, as part of the fulfilment for the award of Masters of Science (M.Sc) degree in Computer Science at the University of Lagos, Akoka, Nigeria. I also declare that this paper is all my efforts and not in any way similar to any research work submitted before this in other institutes for degree or diploma award.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**AFOLAYAN Jephthah Oluwaseyi** **Date**

# 

# CERTIFICATION

This is to certify that this thesis work was prepared by MC-NIEL NZUBECHI CHINEDU, with matriculation number 169074008 in partial fulfilment of the requirement for the award of Master of Science (M.Sc) degree in Computer Science, of the University of Lagos.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**DR. SAWYERR, B.A. Date**

**Supervisor**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Head of Department Date**

# 

# DEDICATION

This work is dedicated to Almighty God, the Giver of all good things.

# 

# ACKNOWLEDGEMENT

I would like to thank God, the Giver of all knowledge, for His mercies, favour and goodness upon my life during the period of this research work.

The completion of this thesis was in no little way made possible with the help and guidance that I received from my supervisor Dr B. A. Sawyerr. It is my sincere prayer that God strengthens and blesses you as you impart and inspire the younger generation.

I seize this opportunity to thank every lecturer in the Department of Computer Science for the knowledge imparted on me so far.

To my parents Mr and Mrs Chinedu, for their love and support given to me during the period of working on this thesis. Lastly, to my siblings, I say thank you.

# 

# 

Table of Contents

[Abstract: 3](#_Toc89088749)

[Chapter 1: Introduction 4](#_Toc89088750)

[Chapter 2: Literature Review 5](#_Toc89088751)

[2.1 Understanding Optimization Problems 5](#_Toc89088752)

[2.2 Discrete Problems and Optimization 5](#_Toc89088753)

[2.3 Continuous Problems and Optimization 5](#_Toc89088754)

[2.3.1 Applications of Continuous Optimization 6](#_Toc89088755)

[Chapter 3: Research Methodology 8](#_Toc89088756)

[3.1 Biological Analogy 8](#_Toc89088757)

[3.2 Historical Development 10](#_Toc89088758)

[3.3 Components of ACO 11](#_Toc89088759)

[3.3.1 Definition of Solution Components and Pheromone Trails 14](#_Toc89088760)

[3.3.2 Balancing Exploration and Exploitation 14](#_Toc89088761)

[3.3.3 ACO and Local Search 15](#_Toc89088762)

[3.3.4 Heuristic Information 15](#_Toc89088763)

[3.4 Developments 16](#_Toc89088764)

[3.4.1 Non-standard Applications of ACO 16](#_Toc89088765)

[Chapter 4: Implementation and Conclusion 18](#_Toc89088766)

[4.1 ACO Application to Continuous Optimization 18](#_Toc89088767)

[4.2 The Algorithm 19](#_Toc89088768)

[4.2.1 Archive structure, initialization, and update 20](#_Toc89088769)

[4.2.2 Probabilistic solution construction 21](#_Toc89088770)

[4.2 Using ACO to Solve Continuous Optimization Problems [Experimental Setup and Results] 22](#_Toc89088771)

[4.3 Conclusions 27](#_Toc89088772)

[Chapter 5: Results and Code Review 29](#_Toc89088773)

[Programming Language: 29](#_Toc89088774)

[Version: 29](#_Toc89088775)

[Problem: 29](#_Toc89088776)

[Algorithm: 30](#_Toc89088777)

[Code: 30](#_Toc89088778)

[Results: 37](#_Toc89088779)

[References 41](#_Toc89088780)

# Abstract:

*Swarm intelligence is a relatively new problem-solving method based on the social behaviour of insects and other animals. Ant Colony Optimization (ACO), one of the most popular Swarm Intelligence techniques is based on the foraging behaviour of certain ant species that deposit pheromones in the soil to indicate faster favourable food source routes to other members of the ant colony. ACO algorithms have been applied to many different discrete optimization problems. In this work, ACO was used to solve continuous space optimization problems. The first proposed algorithm for optimising continuous functions is Continuous Ant Colony Optimization (CACO) which uses the structure of the ant colony to perform a local search, while the global search is handled by a genetic algorithm. Here a Probability Density Functions was used to model the ACO's pheromone. The Gaussian function was used to model the PDF and the mean of the Gaussian function was set to 1, so that the best solution had the highest weight. The choice of the Gaussian function was motivated by its flexibility and non-linear characteristic. This new ACO was then benchmarked using standard benchmark functions.*

# Chapter 1: Introduction

Ant Colony Optimization is a metaheuristic for solving difficult combinatorial optimization problems, inspired by the pheromone (a natural chemical found in insects or animals) laying nature of some real ant species. Real ants use pheromones as a means of communication when searching for food sources. Assuming 2 ants find a food source and take two paths with different distances to the nest, the shortest path would have the highest concentration of pheromones and would therefore be more optimal for other ants to follow to the food source. Overtime, this shorter, more-optimal path would have more ants on it while fewer ants will take the longer path with less pheromone. Eventually, the path with less pheromone will have no ants on it while the shorter path to the food source will have all the ants. Similarly, in ACO, artificial ants implement a random construction heuristic that makes probabilistic decisions based on artificial pheromone traces and possibly available heuristic information based on problematic or standardized input data. Therefore, ACO can be interpreted as an extension of traditional construction heuristics that are readily available for many combinatorial optimization problems. An important difference to the construction heuristic, however, is the adaptation of the pheromone traces during the execution of the algorithm to take into account the accumulated search experience.

From the early nineties, when the first ant colony optimization algorithm was proposed, ACO attracted the attention of increasing numbers of researchers and many successful

applications are now available. Moreover, a substantial collection of theoretical results is becoming available that provides useful guidelines to researchers and practitioners in further applications of ACO. There exist now a considerable number of applications of such algorithms where world class performance is obtained. Examples are applications of ACO algorithms to problems such as sequential ordering [76], scheduling [18], assembly line balancing [19], probabilistic TSP [7], 2D-HP protein folding [132], DNA sequencing [25], protein–ligand docking [98], and packet-switched routing in Internet-like networks [47]. The ACO metaheuristic provides a common framework for the existing applications and algorithmic variants [57, 59]. Algorithms which follow the ACO metaheuristic are called ACO algorithms.

Chapter 2: Literature Review  
In this brief chapter we try to give an overview of Continuous problems. Section 2.1 covers a brief understanding of optimization problems. 2.2 gives a brief introduction to discrete problems and 2.3 takes a deeper dive to continuous problems and the advancements that have been made in problems of this type.

2.1 Understanding Optimization Problems  
To understand the need, use cases and end-goal of optimization, we first need to understand what optimization is. To give a preamble we would consider two quotes from Tjalling C. Koopmans, (1975 Nobel Memorial Lecture): – “best use of scarce resources” and “Mathematical Methods of Organizing and Planning of Production”.

Roger Fletcher (text, 1987) also defined optimization thus: “the subject of optimization is a fascinating blend of heuristics and rigor, of theory and experiment.”   
Generally speaking, the most accepted definition of optimization is “finding the best solution from the group of feasible solutions” thus in mathematics, computer science and economics, an optimization problem is the problem of finding the best solution from all feasible solutions.

Optimization problems can be divided into two categories, depending on whether the variables are continuous or discrete.

An optimization problem with discrete variables is known as a discrete optimization, in which an object such as an integer, permutation or graph must be found from a countable set.

Similarly, a problem with continuous variables is known as continuous optimization, in which an optimal value from a continuous function must be found. They can include constrained problems and multimodal problems which we would not delve into in this work. In the writings that follow we would attempt to explain discrete problems on the periphery and take a deeper dive into understanding continuous optimization problems.

## 2.2 Discrete Problems and Optimization

Discrete or Combinatorial Optimization deals mainly with problems where we have to choose an optimal solution from a finite (or sometimes countable) number of possibilities.

In this short introduction we shall visit a sample of Discrete Optimization problems, step through the thinking process of developing a solution and completely solve one problem. Let us start with a short list of problems.

## 2.3 Continuous Problems and Optimization

Continuous optimization is the core mathematical science for real-world problems ranging from design of biomolecules to management of investment portfolios. Continuous optimization means finding the minimum or maximum value of a function of one or many real variables, subject to constraints. The constraints usually take the form of equations or inequalities.

An example of a major focus of the continuous optimization is convex optimization, that is, continuous optimization in the case that the objective function and feasible set are both convex. Convex optimization problems have widespread applications in practice and also have special properties that make them amenable to sophisticated analysis and powerful algorithms.  
Similar to a combinatorial optimization problem, a model for continuous optimization problem (CnOP) may be formally defined:

Definition 2.3

A model Q = (S, Ω, f ) of a CnOP consists of:  
• a search space S defined over a finite set of continuous decision variables and a set Ω of constraints among the variables;  
• an objective function f : S → R to be minimized.

The search space S is defined as follows:

Given is a set of continuous variables Xi, i = 1, ..., n with possible values vi ∈ Di ⊆ R.

A solution s ∈ S ( i.e., a complete assignment, in which each decision variable has a value assigned) that satisfies all the constraints in the set Ω, is a feasible solution of the given CnOP.

If the set Ω is empty, Q is called an unconstrained problem model, otherwise it is called a constrained one.

A solution s∗ ∈ S is called a global optimum if and only if:

f(s∗) ≤ f(s) ∀s∈S.

The set of all globally optimal solutions is denoted by S∗ ⊆ S.

Solving a CnOP requires finding at least one s∗ ∈ S∗.

The main difference between the combinatorial and continuous optimization problems, is the fact that the search space is not finite. Each of the continuous decision variables may assume an infinite number of values. Of course, solving such problems using computers imposes certain limitations, as computers, being digital in nature, can only represent a finite number of values. One could think that this reduces the problem to a combinatorial optimization one. This is however not really the case. In the case of combinatorial optimization, the set of available values is predefined before starting the optimization. The size of this set has a significant influence on the difficulty of finding an optimal or near-optimal solution. In case of continuous optimization problems, the algorithms use an efficient and flexible floating point representation of real-valued variables. This allows for finding solutions with the required accuracy in a more efficient way.

### 2.3.1 Applications of Continuous Optimization

The area of applications for continuous optimization is very wide. Many real-world problems and processes may be presented in the form of a continuous optimization problem. Typical examples include designing optimal shapes (such as wings, turbines, and others) or choosing values of continuous parameters for various industrial processes (e.g. temperature, pressure, etc.). Continuous optimization algorithms may be used to train artificial neural network, which in turn is used for medical diagnosis. Of course, continuous optimization algorithms are also often used to tackle functions that may be defined using relatively simple mathematical formulas. These may vary from simple test functions, to complex mathematical descriptions of various processes. Examples include engine design, power plants design, or computer simulations of many other processes.

It is important to note that certain problems may be transformed from combinatorial to continuous form by relaxing the requirement of having the values from a finite set. In the evaluation of the objective function, the continuous values may be then rounded to the nearest value from the initial set. Similarly, it is also possible to transform the continuous problem into a combinatorial one by dividing the continuous domains into a set of values. This however is only possible if the continuous domain is bounded (i.e., the lower and upper bounds are defined).

Chapter 3: Research Methodology  
This chapter presents an overview of ant colony optimization (ACO) – a metaheuristic in- spired by the behavior of real ants. ACO was proposed by Dorigo and colleagues [Dorigo et al., 1991; Dorigo, 1992; Dorigo et al., 1996] as a method for solving hard combinatorial optimization problems.

ACO algorithms may be considered to be part of swarm intelligence, that is, the research field that studies algorithms inspired by the observation of the behavior of swarms. Swarm intelligence algorithms are made up of simple individuals that cooperate through self-organization without any form of central control over the swarm members. A detailed overview of the self-organization principles exploited by these algorithms, as well as examples from biology, can be found in [Camazine et al., 2003] as we will not delve into those in this work.

This chapter is organized as follows: Section 3.1 outlines the biological analogy that inspired ACO and 3.2 describes the historical developments leading to it. Section 3.3 illustrates how the ACO metaheuristic can be applied to different types of problems and we give an overview of its successful applications. Section 3.4 gives an overview of recent developments in ACO.

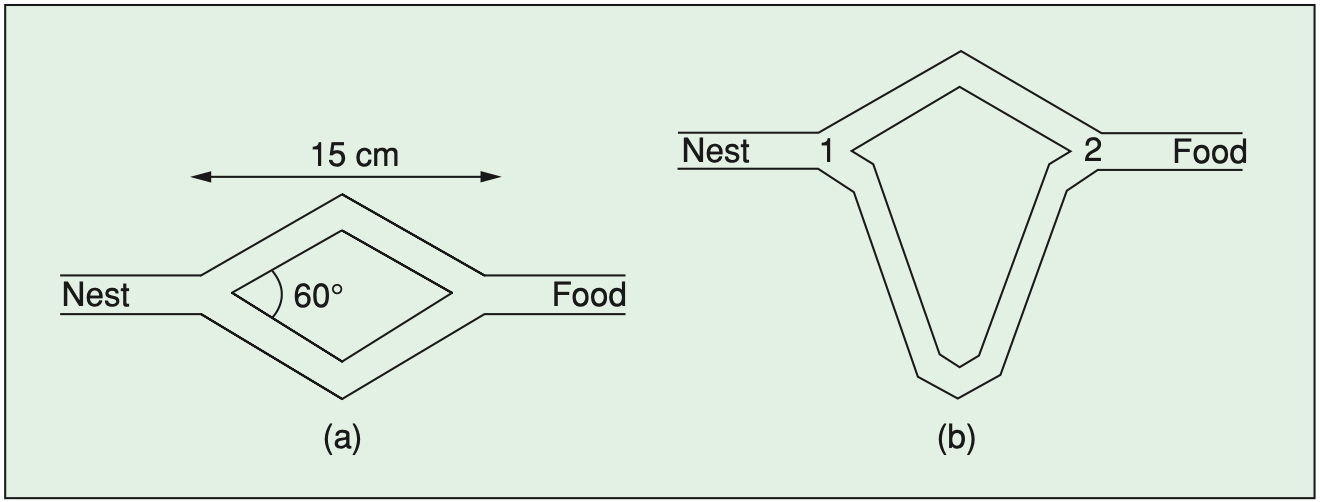
3.1 Biological Analogy  
One of the first researchers to investigate the social behavior of insects was the French entomologist Pierre-Paul Grass ́e. In the forties and fifties of the 20th century, he was observing the behavior of termites – in particular, the Bellicositermes Natalensis and Cubitermes species. Grass ́e discovered that these insects are capable to react to what he called “significant stimuli”, signals that activate a genetically encoded reaction. He observed [Grass ́e, 1959] that the effects of these reactions can act as new significant stimuli for both the insect that produced them and for the other insects in the colony. Grass ́e used the term “*stigmergy*” to describe this particular type of indirect communication in which the “workers are stimulated by the performance they have achieved”.

The two main characteristics of stigmergy that differentiate it from other means of communication are:

* the physical, non-symbolic nature of the information released by the communicating insects, which corresponds to a modification of physical environmental states visited by the insects; and
* the local nature of the released information, which can only be accessed by those insects that visit the place where it was released (or its immediate neighborhood).

Examples of stigmergy can be observed in colonies of ants. In many ant species, individual ants walking to, and from, a food source deposit on the ground a substance called pheromone (a chemical that ants can smell). By depositing pheromone, the ants create a trail that is used, for example, to mark the path from the nest to the food sources and back. Other foraging members of the ant colony can sense the pheromone trails and follow the path to food discovered by the scout ants – i.e. its presence influences the choice of their path as the ants tend to follow strong pheromone concentrations. Several ant species are capable of exploiting pheromone trails to determine the shortest among the available paths leading to the food.

Deneubourg and colleagues [43, 80] used a double bridge connecting a nest of ants and a food source to study pheromone trail laying and following behavior in controlled experimental conditions. They ran a number of experiments called the “binary bridge experiment” in which they varied the ratio between the length of the two branches of the bridge. In one of these experiments, one branch was longer than the other. At the start the ants were left free to move between the nest and the food source and the percentage of ants that chose one or the other of the two branches was observed over time. The outcome was that, although in the initial phase random oscillations could occur, in most experiments all the ants ended up using the shorter branch.



*Figure 1. Experimental setup for double bridge experiment:*

1. *branches have equal lengths (b) branches have different lengths*

This result can be explained as follows. When a trial starts there is no pheromone on the two branches. Hence, the ants do not have a preference and they select with the same probability either of the two branches. It can be expected that, on average, half of the ants choose the short branch and the other half the long branch, although stochastic oscillations may occasionally favor one branch over the other. However, because one branch is shorter than the other, the ants choosing the short branch are the first to reach the food and to start their travel back to the nest. But then, when they must make a decision between the short and the long branch, the higher level of pheromone on the short branch biases their decision in its favor. Therefore, pheromone starts to accumulate faster on the short branch, which will eventually be used by the great majority of the ants.

It should be clear by now how real ants have inspired Ant Systems and later algorithms: the double bridge was substituted by a graph, and pheromone trails by artificial pheromone trails. Also, because the artificial ants were needed to solve problems more complicated than those solved by real ants, extra capacities were given to them, like a memory (used to implement constraints and to allow the ants to retrace their solutions without errors) and the capacity for depositing a quantity of pheromone proportional to the quality of the solution produced (a similar behavior is observed also in some real ant species in which the quantity of pheromone deposited while returning to the nest from a food source is proportional to the quality of the food at the food source ).

In the research field, new algorithms have been proposed that, although retaining some of the original biological inspiration, are less and less biologically inspired and more and more motivated by the need of making ACO algorithms competitive with other state-of-the-art algorithms. Nevertheless, many aspects of the original Ant System remain such as:

* + the need for a colony, the role of autocatalysis,
  + the cooperative behavior mediated by artificial pheromone trails,
  + the probabilistic construction of solutions biased by artificial pheromone trails and local heuristic information,
  + the pheromone updating guided by solution quality,
  + and the evaporation of pheromone trails are present in all ACO algorithms.

## 3.2 Historical Development

Ant System [AS] was the first ACO algorithm to be proposed in literature. In fact, AS was originally a set of three algorithms called *ant-cycle*, *ant-density*, and *ant- quantity*. These three algorithms were proposed in Dorigo’s doctoral dissertation [55] and first appeared in a technical report [63, 64] that was published a few years later in the IEEE Transactions on Systems, Man, and Cybernetics [65].

While in *ant-density* and *ant-quantity* the ants updated the pheromone directly after a move from a city to an adjacent one, in *ant-cycle* the pheromone update was only done after all the ants had constructed the tours and the amount of pheromone deposited by each ant was set to be a function of the tour quality. Because *ant-cycle* performed better than the other two variants, it was later called simply Ant System while the other two algorithms were no longer studied.

The major merit of AS, whose computational results were promising but not competitive with other more established approaches, was to stimulate a number of researchers to develop extensions and improvements of its basic ideas so as to produce better performing, and often state-of-the-art, algorithms.

The basic idea of ACO follows very closely the biological inspiration. Therefore, there are many similarities between real and artificial ants. Both real and artificial ant colonies are composed of a population of individuals that work together to achieve a certain goal. A colony is a population of simple, independent, asynchronous agents that cooperate to find a good solution to the problem at hand. In the case of real ants, the problem is to find the food, while in the case of artificial ants, it is to find a good solution to a given optimization problem. A single ant (either a real or an artificial one) is able to find a solution to its problem, but only cooperation among many individuals through stigmergy enables them to find good solutions.

In the case of real ants, they deposit and react to a chemical substance called pheromone. Real ants simply deposit it on the ground while walking. Artificial ants live in a virtual world, hence they only modify numeric values (called for analogy artificial pheromones) associated with different problem states. A sequence of pheromone values associated with problem states is called artificial pheromone trail. In ACO, the artificial pheromone trails are the sole means of communication among the ants. A mechanism analogous to the evaporation of the physical pheromone in real ant colonies allows the artificial ants to forget the past history and focus on new promising search directions.

Just like real ants, artificial ants create their solutions sequentially by moving from one problem state to another. Real ants simply walk, choosing a direction based on local pheromone concentrations and a stochastic decision policy. Artificial ants also create solutions step-by-step, moving through available problem states and making stochastic decisions at each step.

There are however some important differences between real and artificial ants:

* Artificial ants live in a discrete world – they move sequentially through a finite set of problem states.
* The pheromone update (i.e., pheromone depositing and evaporation) is not ac- complished in exactly the same way by artificial ants as by real ones. Sometimes the pheromone update is done only by some of the artificial ants, and often only after a solution has been constructed.
* Some implementations of artificial ants use additional mechanisms that do not exist in the case of real ants. Examples include look-ahead, local search, backtracking, etc.

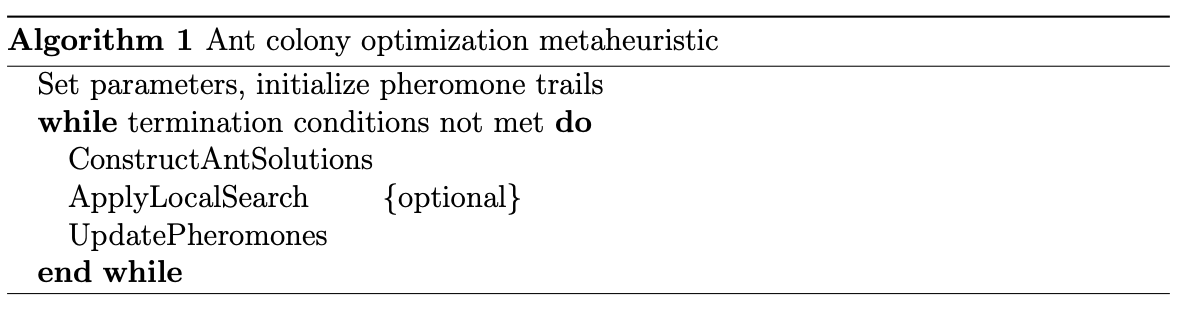
## 3.3 Components of ACO

Given a combinatorial optimization problem (COP), which has been already presented in Section 2.3, the first step for the application of ACO to its solution consists in defining an adequate model. This is then used to define the central component of ACO: the pheromone model.

First, an instantiated decision variable Xi = vij (i.e., a variable Xi with a value vij assigned from its domain Di), is called a solution component and denoted by cij. The set of all possible solution components is denoted by C. A pheromone trail parameter Tij is then associated with each component cij. The set of all pheromone trail parameters is denoted by T. The value of a pheromone trail parameter Tij is denoted by τij (and called pheromone value).2 This pheromone value is then used and updated by the ACO algorithm during the search. It allows modeling the probability distribution of different components of the solution.

In ACO, artificial ants build a solution to a combinatorial optimization problem by traversing the so-called construction graph, GC (V, E). The fully connected construction graph consists of a set of vertexes V and a set of edges E. The set of components C may be associated either with the set of vertexes V of the graph GC, or with the set of its edges E. The ants move from vertex to vertex along the edges of the graph, incrementally building a partial solution. Additionally, the ants deposit a certain amount of pheromone on the components, that is, either on the vertexes or on the edges that they traverse. The amount ∆τ of pheromone deposited may depend on the quality of the solution found. Subsequent ants utilize the pheromone information as a guide towards more promising regions of the search space.

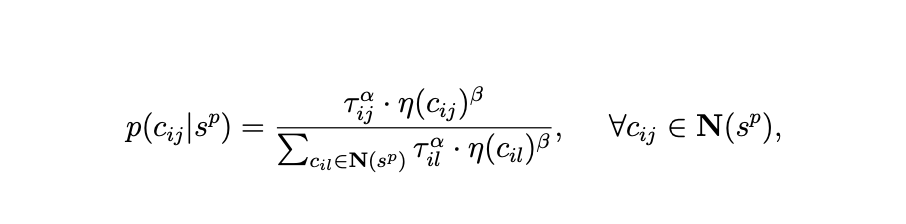
The ACO metaheuristic is shown in Algorithm 1. It consists of an initialization step and a loop over three algorithmic components. A single iteration of the loop consists of constructing solutions by all ants, their (optional) improvement with the use of a local search algorithm, and an update of the pheromones. In the passages that follow, we explain these three algorithmic components in more detail.



*Algorithm 1.*

ConstructAntSolutions: A set of m artificial ants construct solutions from elements of a finite set of available solution components C = {cij}, i = 1,...,n, j = 1,...,|Di|. A solution construction starts with an empty partial solution sp = ∅. Then, at each construction step, the current partial solution sp is extended by adding a feasible solution component from the set of feasible neighbors N(sp) ⊆ C. The process of constructing solutions can be regarded as a path on the construction graph GC = (V, E). The allowed paths in GC are hereby implicitly defined by the solution construction mechanism that defines the set N(sp) with respect to a partial solution sp.

The choice of a solution component from N(sp) is done probabilistically at each construc- tion step. The exact rules for the probabilistic choice of solution components vary across different ACO variants. The best known rule is the one of Ant System (AS) [Dorigo et al., 1996]:



Eq. 3.2

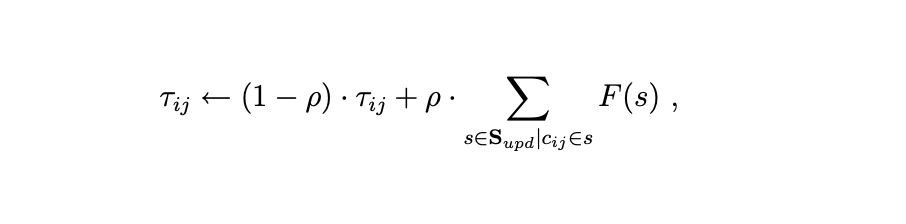
where:   
τij is the pheromone value associated with the component cij,   
and η(·) is a function that assigns at each construction step a heuristic value to each feasible solution component cij ∈ N(sp).

The values that are given by this function are commonly called heuristic information. Furthermore, α and β are positive parameters, whose values determine the relative importance of pheromone versus heuristic information.   
Eq. 3.2 is a generalization of Eq. 3.1 presented in Section 3.1:   
ACO formalization follows closely the biological inspiration.

*ApplyLocalSearch:* Once solutions have been constructed, and before updating pheromones, often some optional actions may be required. These are often called daemon actions, and can be used to implement problem specific and/or centralized actions, which cannot be performed by single ants. The most used daemon action consists in the application of

local search to the constructed solutions: the locally optimized solutions are then used to decide which pheromones to update.

*UpdatePheromones:* The aim of the pheromone update is to increase the pheromone values associated with good or promising solutions, and to decrease those that are asso- ciated with bad ones. Usually, this is achieved   
(i) by decreasing all the pheromone values through pheromone evaporation, and   
(ii) by increasing the pheromone levels associated with a chosen set of good solutions Supd:

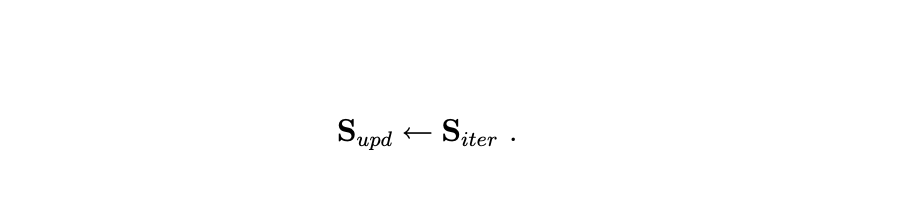


Eq. (3.3)

where Supd is the set of solutions that are used for the update, ρ ∈ (0, 1] is a parameter called evaporation rate, and F : S → R+0 is a function such that f(s) < f(s′) ⇒ F(s) ≥ F(s′),∀s ̸= s′ ∈ S. F(·) is commonly called the fitness function.

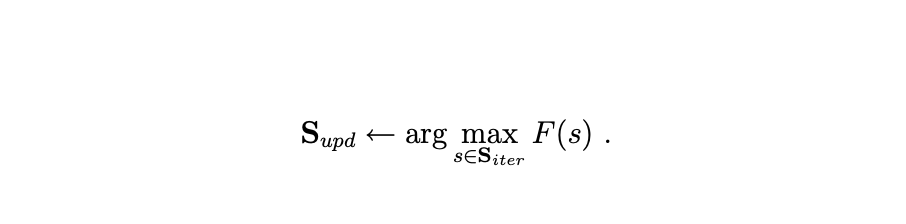
Pheromone evaporation is needed to avoid a too rapid convergence of the algorithm. It implements a useful form of forgetting, favoring the exploration of new areas in the search space. Different ACO algorithms, such as for example Ant Colony System (ACS) [Dorigo and Gambardella, 1997] or MAX-MIN Ant System (MMAS) [Stu ̈tzle and Hoos, 2000] differ in the way they update the pheromone.

Instantiations of the update rule presented in Eq. 3.3 are obtained by different specifi- cations of Supd, which in many cases is a subset of Siter ∪ {sbs}, where Siter is the set of solutions that were constructed in the current iteration, and sbs is the best-so-far solu- tion, that is, the best solution found since the first algorithm iteration. A well-known example is the AS-update rule, that is, the update rule of Ant System [Dorigo et al., 1996], where:



Eq. (3.4)

An example of a pheromone update rule that is more often used in practice is the IB- update rule (where IB stands for iteration-best):



Eq. (3.5)

The IB-update rule introduces a much stronger bias towards the good solutions found than the AS-update rule. Although this increases the speed with which good solutions are found, it also increases the probability of premature convergence. An even stronger bias is introduced by the BS-update rule, where BS refers to the use of the best-so- far solution sbs. In this case, Supd is set to {ssb}. In practice, ACO algorithms that use variations of the IB-update or the BS-update rules and that additionally include mechanisms to avoid premature convergence, achieve better results than those that use the AS-update rule.

### 3.3.1 Definition of Solution Components and Pheromone Trails

Of crucial importance in ACO applications is the definition of the solution components and of the pheromone model. Consider, for example, the differences in the definition of solution components in the TSP and the SMTWTP. Although both problems represent solutions as permutations, the definition of solution components (and, hence, the interpretation of the pheromone trails), is very different. In the TSP case, a solution component refers to the direct successor relationship between elements, while in the SMTWTP it refers to the allocation of a job to a specific position in the permutation. This is intuitively due to the different role that permutations have in the two problems. In the TSP, only the relative order of the solution components is important and a permutation π = (1,2,...,*n*) has the same tour length as the permutation π ′ = (*n*, 1, 2, . . . , *n* − 1)—it represents the same tour. On the contrary, in the SMTWTP (as well as in many other scheduling problems), π and π′ would represent two different solutions with most probably very different costs; in this case the position information is very important.

In some applications, the role of the pheromone trail definition has been investigated in more depth. Blum and Sampels compare different ways of defining the pheromone model for shop scheduling problems [23]. In [22], Blum and Dorigo show that the choice of an inappropriate pheromone model can result in an undesirable performance degradation over time. Fortunately, in many applications the solution components used in high-performing constructive algorithms, together with the correct choice of the pheromone model, typically result in high-performing algorithms. However, finding the best pheromone model is not always a straightforward task. Examples of some more complex or unusual choices are the ACO application to the shortest common super sequence problem [114] or the application of ACO to protein–ligand docking [98].

### 3.3.2 Balancing Exploration and Exploitation

Any effective metaheuristic algorithm has to achieve an appropriate balance between the exploitation of the search experience gathered so far and the exploration of unvisited or relatively unexplored search space regions. In ACO, several ways exist for achieving such a balance, typically through the management of the pheromone trails. In fact, the pheromone trails induce a probability distribution over the search space and determine which parts of the search space are effectively sampled, that is, in which part of the search space the constructed solutions are located with higher frequency.

The best performing ACO algorithms typically use an *elitist strategy* in which the best solutions found during the search contribute strongly to pheromone trail updating. A stronger exploitation of the “learned” pheromone trails can be achieved during solution construction by applying the pseudo-random proportional rule of ACS, as explained in Section 2.2. These exploitation features are then typically combined with some means to ensure enough search space exploration trying to avoid convergence of the ants to a single path, corresponding to a situation of search stagnation. There are several ways to try to avoid such stagnation situations. For example, in ACS the ants use a local pheromone update rule during solution construction to make the path they have taken less desirable for subsequent ants and, thus, to diversify the search. *MM*AS introduces an explicit lower limit on the pheromone trail value so that a minimal level of exploration is always guaranteed. *MM*AS also uses a reinitialization of the pheromone trails, which is a way of enforcing search space exploration. Finally, an important role in the balance of exploration and exploitation is played by the parameters α and β in Equation (8.1). Consider, for example, the influence of parameter α. (Parameter β has an analogous influence on the exploitation of the heuristic information.) For α > 0, the larger the value of α the stronger the exploitation of the search experience; for α = 0 the pheromone trails are not taken into account at all; and for α < 0 the most probable choices taken by the ants are those that are less desirable from the point of view of pheromone trails. Hence, varying α could be used to shift from exploration to exploitation and conversely.

### 3.3.3 ACO and Local Search

In many applications to NP-hard combinatorial optimization problems, ACO algorithms perform best when coupled with local search algorithms. Local search algorithms locally optimize the ants’ solutions and these locally optimized solutions are used in the pheromone update.

The use of local search in ACO algorithms can be very interesting as the two approaches are complementary. In fact, ACO algorithms perform a rather coarse- grained search, and the solutions they produce can then be locally fine-tuned by an adequate local search algorithm. On the other side, generating appropriate

248 Marco Dorigo and Thomas Stu ̈tzle

initial solutions for local search algorithms is not an easy task. In practice, ants probabilistically combine solution components which are part of the best locally optimal solutions found so far and generate new, promising initial solutions for the local search. Experimentally, it has been found that such a combination of a probabilistic, adaptive construction heuristic with local search can yield excellent results [26, 61, 147].

Despite the fact that the use of local search algorithms has been shown to be crucial for achieving state-of-the-art performance in many ACO applications, it should be noted that ACO algorithms also show very good performance when local search algorithms cannot be applied easily [47, 114].

### 3.3.4 Heuristic Information

The possibility of using heuristic information to direct the ants’ probabilistic solution construction is important because it gives the possibility of exploiting problem- specific knowledge. This knowledge can be available a priori (this is the most frequent situation in NP-hard problems) or at runtime (this is the typical situation in dynamic problems).

For most NP-hard problems, the heuristic information η can be computed at initialization time and then it remains the same throughout the whole algorithm’s run. An example is the use, in the TSP applications, of the length *dij* of the edge connecting cities *i* and *j* to define the heuristic information η*ij* = 1/*dij* . However, the heuristic information may also depend on the partial solution constructed so far and therefore be computed at each step of an ant’s solution construction. This determines a higher computational cost that may be compensated by the higher accuracy of the computed heuristic values. For example, in the ACO applications to the SMTWTP and the SCP the use of such “adaptive” heuristic information was found to be crucial for reaching very high performance.

Finally, it should be noted that while the use of heuristic information is rather important for a generic ACO algorithm, its importance is strongly reduced if local search is used to improve solutions. This is due to the fact that local search takes into account information about the cost to improve solutions in a more direct way.

## 3.4 Developments

In this section, we review recent research trends in ACO. These include (i) the application of ACO algorithms to non-standard problems; (ii) the development of ACO algorithms that are hybridized with other metaheuristics or techniques from mathematical programming; (iii) the parallel implementation of ACO algorithms; and (iv) theoretical results on ACO algorithms.

### 3.4.1 Non-standard Applications of ACO

We review here applications of ACO to problems that involve complicating factors such as multiple objective functions, time-varying data, and stochastic information about objective values or constraints. In addition, we review some recent applications of ACO to continuous optimization problems.

#### 3.4.1.1 Multiobjective Optimization

Frequently, in real-world applications, various solutions are evaluated as a function of multiple, often conflicting objectives. In simple cases, objectives can be ordered with respect to their importance or they can be combined into a single-objective by using a weighted sum approach. An example of the former approach is the application of a two-colony ACS algorithm for the vehicle routing problem with time windows [77]; an example of the latter is given by Doerner et al. [51] for a bi-objective transportation problem.

If a priori preferences or weights are not available, the usual option is to approximate the set of Pareto-optimal solutions—a solution *s* is Pareto optimal if no other solution has a better value than *s* for at least one objective and is not worse than *s* for the remaining objectives. The first general ACO approach targeted to such problems is due to Iredi et al. [93], who discussed various alternatives to apply ACO to multi-objective problems and presented results with a few variants for a bi-objective scheduling problem. Since then, several algorithmic studies have tested various alternative approaches. These possible approaches differ in whether they use one or several pheromone matrices (one for each objective), one or several heuristic information, how solutions are chosen for pheromone deposit, and whether one or several colonies of ants are used. Several combinations of these possibilities have been studied, for example, in [3, 92]. For a detailed overview of the available multi-objective ACO algorithms we refer to the review articles by Garc ́ıa-Mart ́ınez [78], that also contains an experimental evaluation of some proposed ACO approaches, and by Angus and Woodward [5].

#### 3.4.1.2 Dynamic Versions of NP-Hard Problems

As said earlier, ACO algorithms have been applied with significant success to dynamic problems in the area of network routing [47, 49]. ACO algorithms have also been applied to dynamic versions of classical NP-hard problems. Examples are the applications to dynamic versions of the TSP, where the distances between cities may change or where cities may appear or disappear [70, 81, 82]. More recently, applications of ACS to dynamic vehicle routing problems are reported in [54, 117], showing good results on academic instances of the problem as well as on real-world instances.

#### 3.4.1.3 Stochastic Optimization Problems

In many optimization problems data are not known exactly before generating a solution. Rather, what is available is stochastic information on the objective function value(s), on the decision variable values, or on the constraint boundaries due to uncertainty, noise, approximation or other factors. ACO algorithms have been applied to a few stochastic optimization problems. The first stochastic problem to which ACO was applied is the probabilistic TSP (PTSP), where for each city the probability that it requires a visit is known and the goal is to find an apriori tour of minimal expected length over all the cities. The first to apply ACO to the PTSP were Bianchi et al. [14], who used an adaptation of ACS. This algorithm was improved by Branke and Guntsch and, very recently, by Balaprakash et al. [7], resulting in a state-of- the-art algorithm for the PTSP. Other applications of ACO include vehicle routing problems with uncertain demands [13] and the selection of optimal screening policies for diabetic retinopathy [28]; the latter approach builds on the S-ACO algorithm proposed earlier by Gutjahr [85].

# Chapter 4: Implementation and Conclusion

4.1 ACO Application to Continuous Optimization

Although ACO was proposed for combinatorial problems, researchers started to adapt it to continuous optimization problems. The simplest approach for applying ACO to continuous problems would be to discretize the real-valued domain of the variables. Recently, ACO algorithms that handle continuous parameters natively have been proposed. An example is the work of Socha and Dorigo [137], where the probability density functions that are implicitly built by the pheromone model are explicitly represented by Gaussian kernel functions. Their approach has also been extended to mixed-variable (continuous and discrete) problems [136]. Other references on this subject are [134, 151, 153].   
Combinatorial optimization deals with finding optimal combinations or permutations of available problem components. Hence, it is required that the problem is partitioned into a finite set of components, and the combinatorial optimization algorithm attempts to find their optimal combination or permutation. Many real-world optimization problems may be represented as COPs in a straightforward way. There exists however an important class of problems for which this is not the case: the class of optimization problems that require choosing values for continuous variables. Such problems may be tackled with a combinatorial optimization algorithm only once the continuous ranges of allowed values are converted into finite sets. This is not always convenient, especially if the initial possible range is wide, and the resolution required is very high. In these cases, algorithms that can natively handle continuous variables usually perform better.

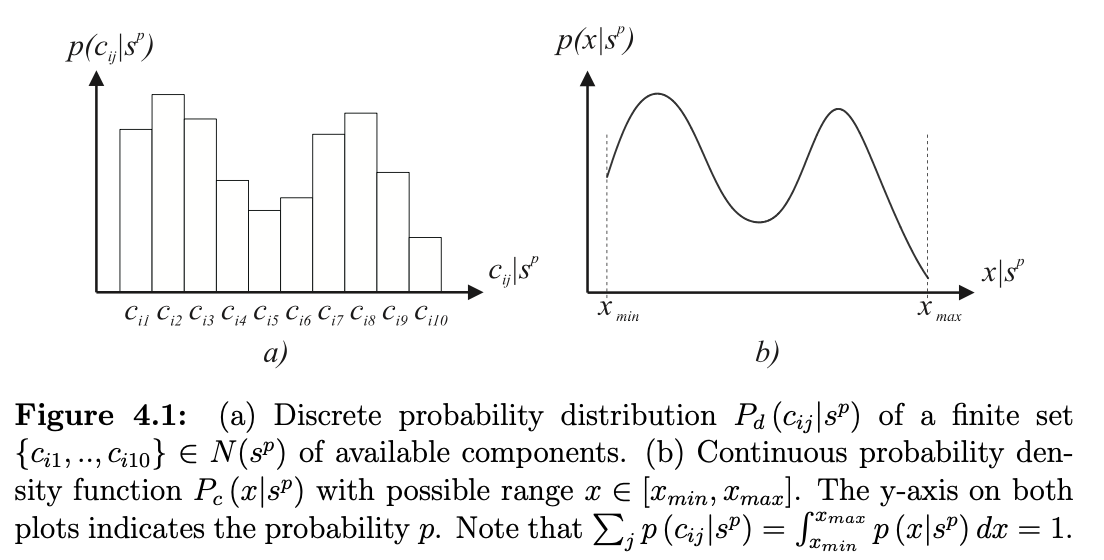
Chapter 3 presented the basics of ant colony optimization. This chapter presents a way to effectively apply ACO to continuous optimization problems in the form of the AROr algorithm presented by Socha et al [].

In Socha et al [], the extended ACO to continuous domains was denoted by ACOr. The ACOr performed competitively against other hybrid implementations of ACO for continuous domains such as CACO, API and CIAC. However, in his work, we noticed that .Our aim is to present the core idea of applying rACOm to continuous domains and compare its unbaised performance on select standard benchmark test problems. In order to have a proper perspective on the unbiased performance of ACOm, we compare it other metaheuristics used for continuous optimization. This unbiased, modified version of the ACOr algorithm will be demoted by mACOr.

The reminder of the chapter is organized as follows. Section 4.2 presents the ACOr algorithm. Section 4.3 provides a discussion of the approach used with regard to other methods for continuous optimization. It also presents the experimental setup and results obtained, for ACOr and the unbiased version mACOr and compares them to the results found in the literature. Finally, Section 4.4 presents the conclusions and future work plans.

## 4.2 The ACOr Algorithm

The central idea in the way ACO works is the incremental construction of solutions based on the biased probabilistic choice of solution components. In ACO applied to combinatorial optimization problems, the set of available solution components is defined by the problem formulation. At each construction step, ants make a probabilistic choice of the solution component ci from the set N(sp) of available components according to Equation 3.2.



The probabilities associated with the elements of the set N(sp) make up a discrete probability distribution (Figure 4.1a) that an ant samples in order to choose a component to be added to the current partial solution sp.

The fundamental idea underlying ACOm is the shift from using a discrete probability distribution to using a continuous one, that is, a probability density function (PDF) (Figure 4.1b). In ACOr, instead of choosing a component cij ∈ N(sp) according to Equation 3.2, an ant samples a PDF.

The ACO metaheuristic finds approximate solutions to an optimization problem by iterating the following two steps:

Fig(4.1)

1. Candidate solutions are constructed in a probabilistic way using a probability

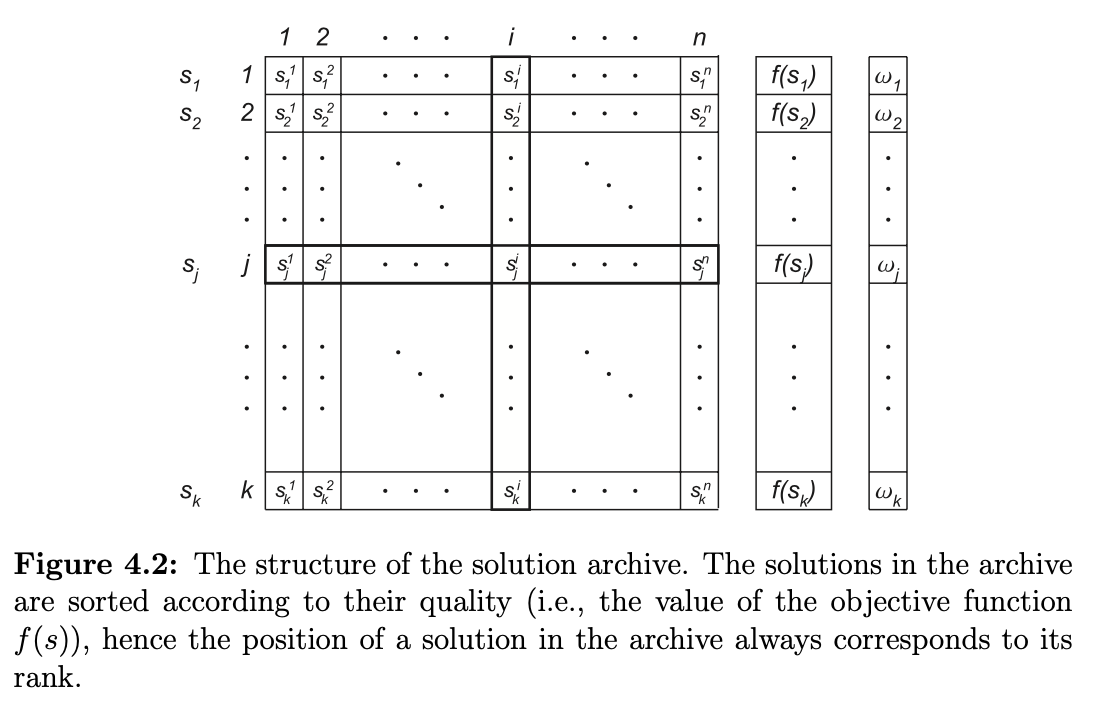
distribution over the search space;

2. The candidate solutions are used to modify the probability distribution in a way that is deemed to bias future sampling toward high quality solutions.

ACO algorithms for combinatorial optimization problems make use of a pheromone model in order to probabilistically construct solutions. A pheromone model is a set of so-called pheromone trail parameters. The numerical values of these pheromone trail parameters (that is, the pheromone values) reflect the search experience of the algorithm. They are used to bias the solution construction over time towards the regions of the search space containing high quality solutions.

In ACO for combinatorial problems, the pheromone values are associated with a finite set of discrete values related to the decisions that the ants make. This allows to represent the pheromone values in the form of a pheromone table. This is not possible in the continuous case, as the number of possible values is not finite. Hence, ACOm uses rather a solution archive as way of describing the pheromone distribution over the search space. The solution archive contains a number of complete solutions to the problem. While a pheromone model in combinatorial optimization can be seen as an implicit memory of the search history, a solution archive is an explicit memory.1

The basic flow of the ACOm algorithm is as follows. As a first step, the solution archive is initialized. Then, at each iteration, a number of solutions is probabilistically constructed



by the ants. These solutions may be improved by any improvement mechanism (for example, local search or gradient techniques). Finally, the solution archive is updated with the generated solutions. In the following we outline the components of ACOm in more details.

### 4.2.1 Archive structure, initialization, and update

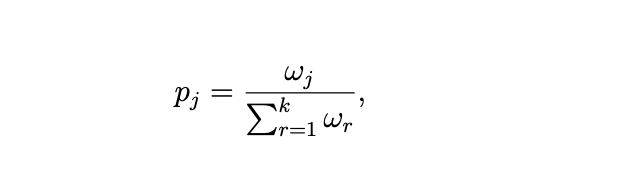
ACOm keeps a history of its search process by storing solutions in a solution archive T of dimension |T| = k. Given an n-dimensional continuous optimization problem and k solutions, ACOm stores in T the values of the solutions’ n variables and the value of their objective functions. The value of the i-th variable of the j-th solution is in the following denoted by sij. Figure 4.2 shows the structure of the solution archive.

Before the start of the algorithm, the archive is initialized with k random solutions. At each algorithm iteration, first, a set of m solutions is generated by the ants and added to those in T . From this set of k + m solutions, the m worst ones are removed. The remaining k solutions are sorted according to their quality (i.e., the value of the objective function) and stored in the new T. In this way, the search process is biased towards the best solutions found during the search. The solutions in the archive are

always kept sorted based on their quality (i.e., the objective function values), so that the best solution is on top.

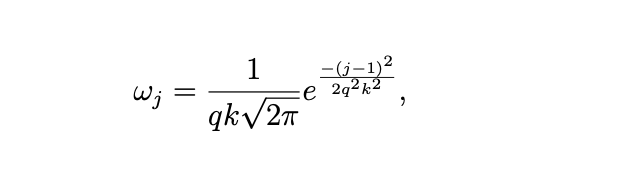
### 4.2.2 Probabilistic solution construction

Construction of new solutions by the ants is accomplished in an incremental manner, variable by variable. First, an ant chooses probabilistically one of the solutions in the archive. The probability of choosing solution j is given by:



( Eq 4.1)

where ωj is a weight associated with solution j. The weight may be calculated using various formulas depending on the problem tackled. In the remainder of this paper we use a Gaussian function g(μ,σ) = g(1,qk), which was also used in our previous work [Socha and Dorigo, 2008]:



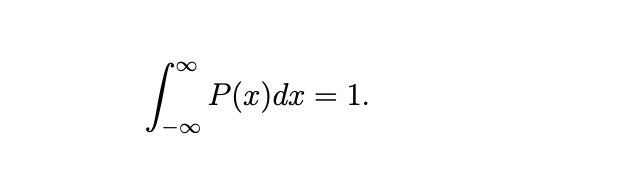
(Eq 4.2)

where, q is a parameter of the algorithm and k is the size of the archive. The mean of the Gaussian function is set to 1, so that the best solution has the highest weight.

The choice of the Gaussian function was motivated by its flexibility and non-linear characteristic. Thanks to its non-linearity, it allows for a flexible control over the weights. It is possible to give higher probability to a few leading solutions, while significantly reducing the probability of the remaining ones.

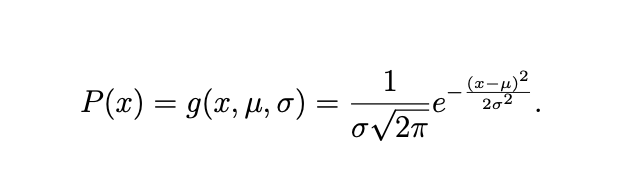
Once one of the solutions in the archive is chosen, an ant may start constructing a new solution. However, before the new solution is created, the algorithm attempts to de-correlate the variables of the solutions in the archive. It does so through a linear transformation of the soltuion archive while generating new solutions and reverting to original state afterwards. As this does not have an impact on the way the solutions are generated, we do not describe this process here.

The ant treats each problem variable i = 1, ..., n separately. It takes the value sij of the variable i of the chosen j-th solution and samples its neighborhood. This is done using a probability density function (PDF). Again, as in the case of choosing the weights, many different functions may be used. A PDF P(x) must however satisfy the condition:



(Eq 4.3)

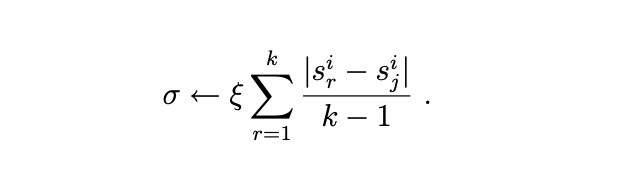
In this work, similarly to earlier publications presenting ACOr [Socha and Dorigo, 2008], we use as PDF the Gaussian function:



(Eq 4.4)

The function has two parameters that must be defined: μ, and σ. When considering

variable i of solution j, we assign μ ← sij. Further, we assign σ:



(Eq 4.5)

which is the average distance between the i-th variable of the solution sj and the i-th

variables of the other solutions in the archive, multiplied by a parameter ξ.  
This whole process is repeated for each dimension i = 1, ..., n in turn by each of the m

ants.

## 4.3 Using ACO to Solve Continuous Optimization Problems [Experimental Setup and Results for ACOr and mACOr]

In this section of the chapter, we present the experimental setup for evaluating the performance of ACOr and the results obtained. In order to have an overview of the performance of ACOr in comparison to other methods for continuous optimization, we used the typical benchmark test functions that have been used in the literature for presenting the performance of different methods and algorithms. Seeing that it is impractical to compare ACOr to every method that has been used for continuous optimization in the past, we decided to compare the performance of ACOr using direct search methods for the following standard problems.

* Ackley’s Function—methods which explicitly model and sample probability distributions.
* Rosenbrock’s Function—methods that claim to draw inspiration from the behavior of ants.
* Shekel Function — originally developed for combinatorial optimization and later adapted to continuous domains.

*For ACOr, a slightly different experimental methodology was used for each comparison in order to make the results obtained by ACOr as comparable as possible to those obtained with the other methods. However, with mACOr, we used the same methodology to eliminate any favorable biases and compare the results to other search methods. We did this to ascertain its unbiased competitiveness.*

## 4.3.1 ACOr and mACOr compared with Direct Search Methods

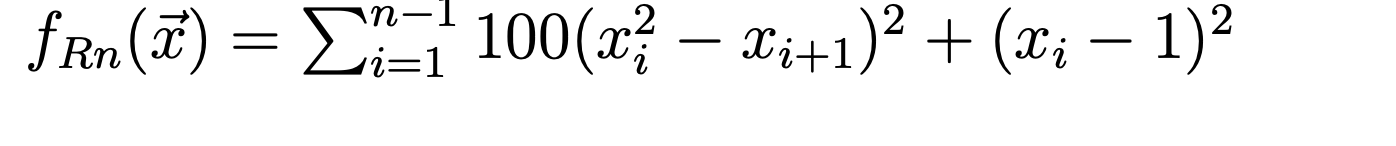
As mentioned in Chapter 2, there exist also a set of direct search methods for continuous optimization. They include among others the Nelder-Mead (or Simplex) method the Powell method. These methods, similarly to metaheuristics, do not need any additional information about the problem being solved. Hence, they may be potentially applied to the same problems as our ACOm proposed algorithm. The direct search methods are known to perform well on many real-world problems. Their main disadvantage is that they search for local minima rather than for a global one. Hence, they are often used either as local search methods for metaheuristics, or as random-restart local search algorithms, where multiple runs of a direct search method are used in order to increase the chances of finding the global optimum.

This section focuses on how our ACOm algorithm compares to these direct search meth- ods on different types of problems. In order to have a proper overview, we have chosen three continuous optimization test problems of different characteristics. We chose the Rosenbrock function (n = 10) as an example of a difficult unimodal test problem, the Shekel (4,7) function as an example of low dimensional (n = 4) multimodal test problem with few local optima, and finally an Ackley function (n = 10), as an example of a multimodal test function with many local optima. For all the test functions we used the earlier mentioned skewed initialization in order to avoid bias of population based methods towards the center of the search space.

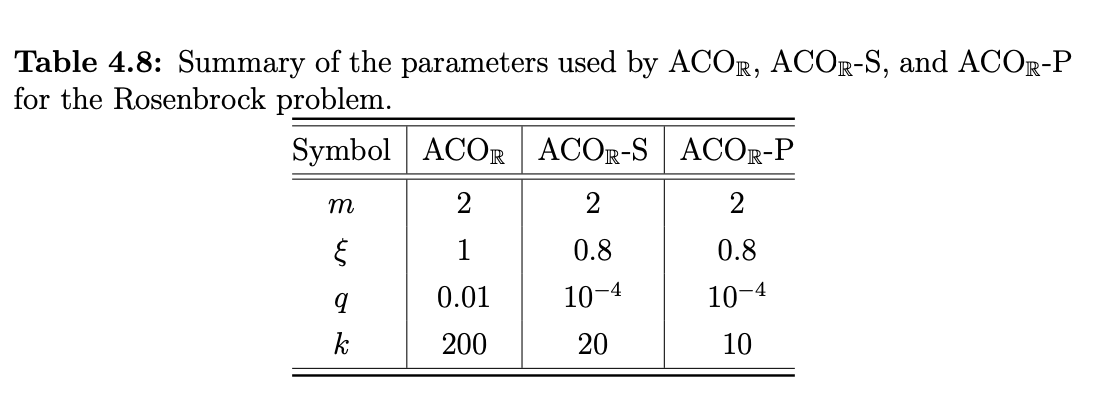
As examples of the direct search methods, we used the Simplex (S) method and the Powell (P) method. In order to have a proper perspective, we have tested several different configurations, including: ACOm algorithm alone, random-restart versions of Simplex (RRS) and Powell (RRP), as well as hybrids of ACOr with respectively Simplex (ACOr- S) or Powell (ACOr-P) algorithm as a local search and mACOr algorithm with the same state parameters as the ACOr-S and ACOr-P algorithms. We have implemented the mACOr and ACOr methods in python and all needed packages and extensions are found in the readme section of the project code base.

For each of the test problems, we have done a separate set of experiments without tuning the algorithms’ parameters in mACOr and tuning the algorithm’s parameters in ACOr. In the following subsections, we describe the results of each of these experiments.

Rosenbrock Function

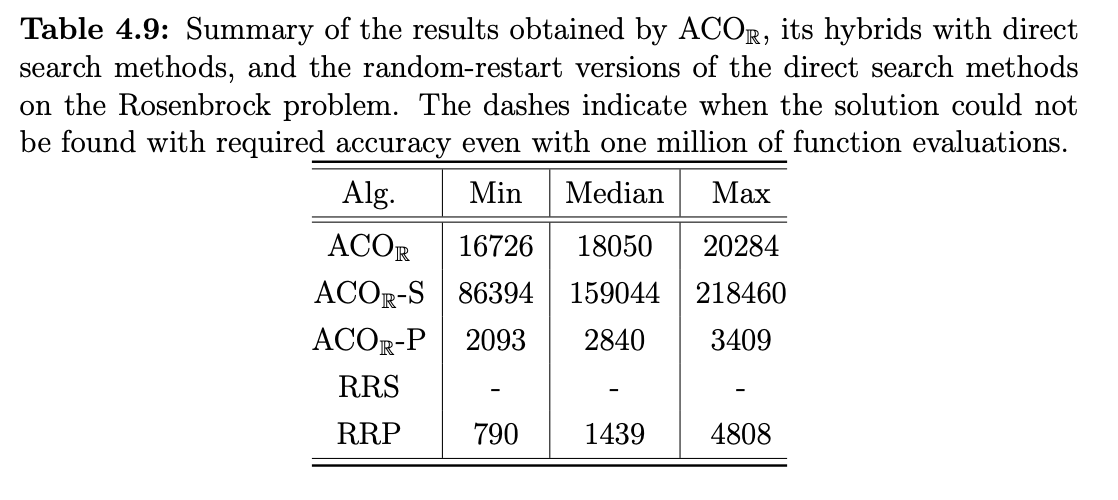


The first problem that we tackled was the Rosenbrock test function with n = 10 di- mensions. This function has already been used in Section 4.3.1. ACOm, ACOm-S, and ACOm-P required choosing appropriate parameters. We have accomplished this using the F-Race method mentioned earlier. Table 4.8 presents the set of parameters chosen for these algorithms.



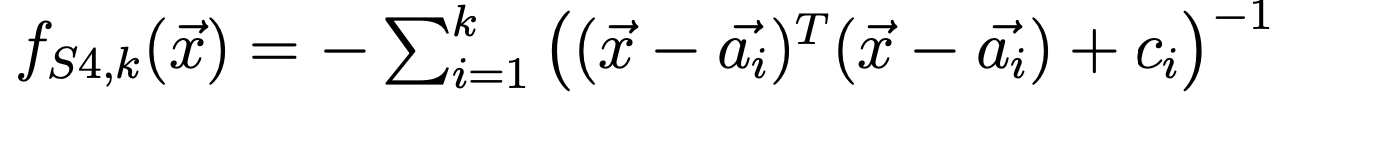
For the mACOr we used the following Parameters: *m* = 2 : *ξ* = 0.8 : *q* = 0.001 : *k* = 10

The experiment aimed at establishing how many function evaluations would each of the algorithms need to optimize the 10-dimensional Rosenbrock function with accuracy of at least e = 10−4. We have done 100 independent runs of each algorithm. Table 4.9 presents the values of the minimum, median, and maximum number of objective function evaluations needed to achieve the required accuracy by each of the algorithms.

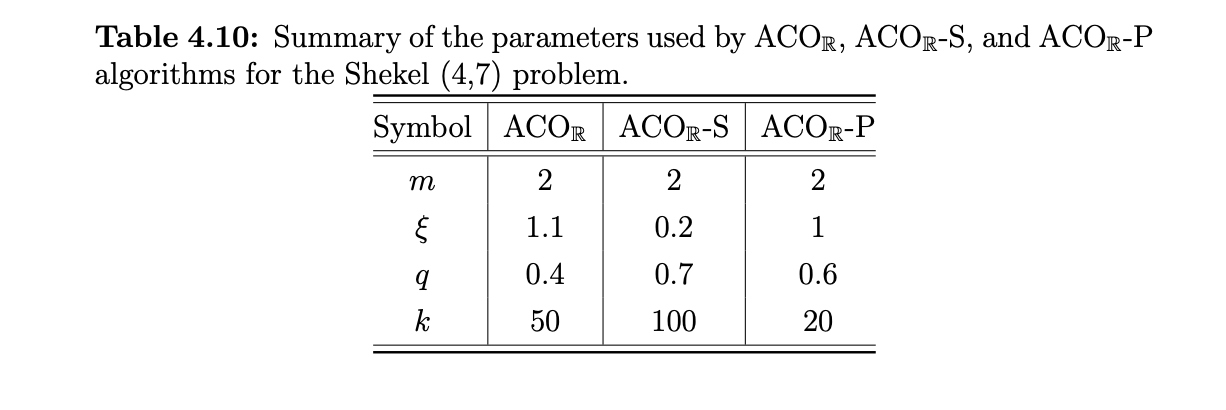


After analyzing the results obtained, it is clear that the Powell method is the most ef- ficient for solving this problem. Hybridizing the Powell method with ACOm does not improve the results. Most likely, ACOm adds an unnecessary overhead, which increases the number of evaluations needed to solve the problem. Contrary to the Powell method, the Simplex method does not seem to be able to deal very well with the Rosenbrock problem. The random-restart version of the Simplex algorithm could not find the op- timum solution even with one million of function evaluations. A hybrid of ACOm with the Simplex method as a local search did reach the optimum, but it took on average 100 times more function evaluations than the Powell method, and about 10 times more than ACOm alone. Note that ACOm was run here with different parameters than in Sec- tion 4.3.1. While there ACOm was converging faster, there were some unsuccessful runs. Here, more robust parameter settings were used, which allowed to obtain 100% success rate, that is, all ACOm runs resulted in finding the optimum solution with required accuracy.

Shekel Function

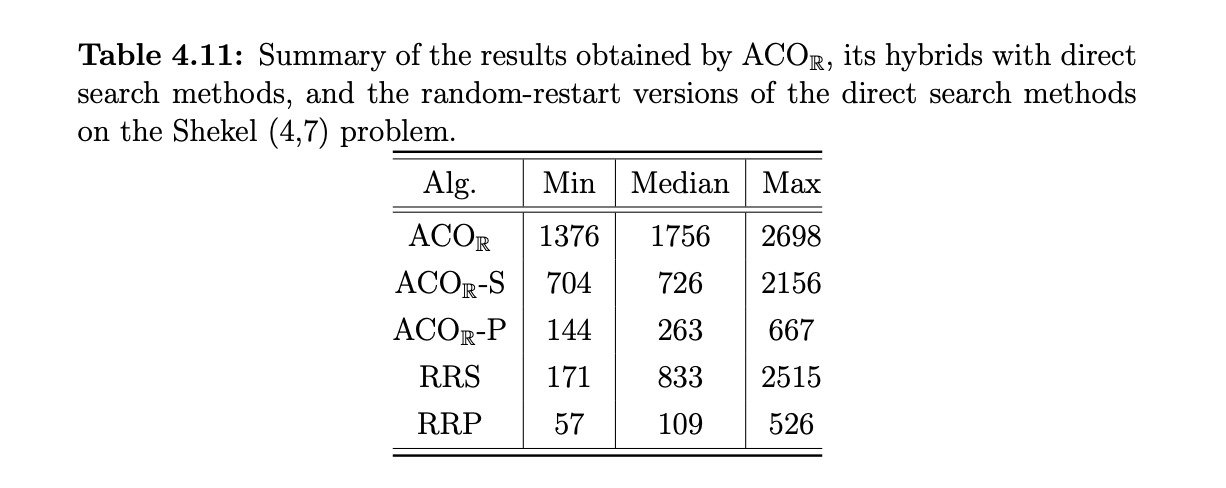


The second function that was used to compare the performance of ACOm with direct search methods, was Shekel (4,7). It was used before in Section 4.3.2. It is a multimodal function with n = 4 dimensions, and few local optima. We have chosen the parameters for ACOm, ACOm-S, and ACOm-P using the F-Race method mentioned earlier. Table 4.10 presents these parameters.



For the mACOr we used the following Parameters: *m* = 2 : *ξ* = 1 : *q* = 0.6 : *k* = 20  
and : *m* = 2 : *ξ* = 0.2 : *q* = 0.7 : *k* = 100

Similarly to the Rosenbrock function, we measured the number of function evaluations required to reach the accuracy of e = 10−4. We have done 100 independent runs of each algorithm. Table 4.11 presents the values of the minimum, median, and maximum number of the objective function evaluations needed to achieve the required accuracy by each of the algorithms.

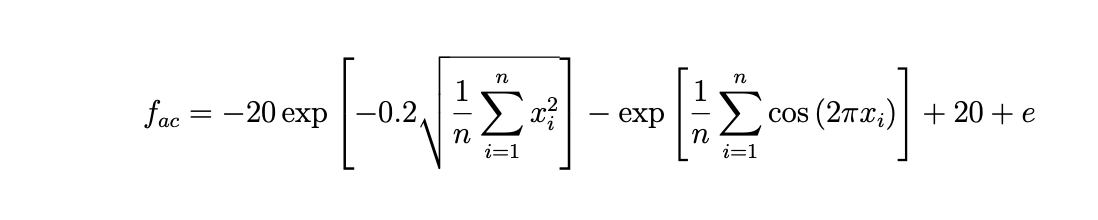


Clearly, for the Shekel (4,7) problem, both direct search methods are much more efficient than ACOm. Powell is also much better than the Simplex method. Hybridizing ACOm with the direct search methods improves the ACOm performance, but it is still worse than the random-restart version of the respective direct search method alone.

The results show that a random-restart direct search method is quite efficient for a problem of few dimensions and not too many local optima. In the worst case only few iterations allow to find the global optimum. While for the direct search methods it is sufficient to start with an initial point that is the right valley to be certain to find the optimum, this is more complicated for ACOm. In the Shekel (4,7) problems, the valleys of all optima are rather large and the quality of local optima does not differ significantly. Hence, ACOm tends to sample many of them at the same time, and it takes some tome before it converges to the correct one. Therefore, this experiment shows that random- restart direct search heuristics significantly outperform ACOm for this type of problems.

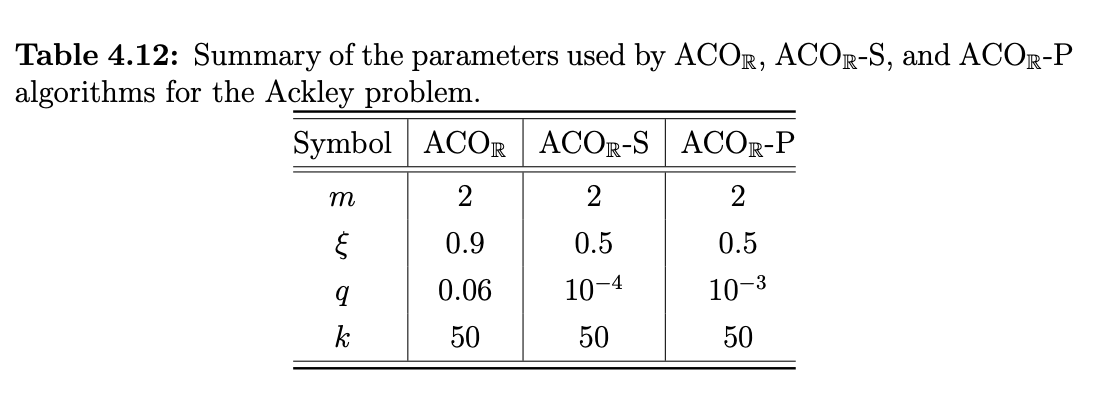
Ackley Function

The third and final test function that we chose for comparing ACOm to direct search methods, is the Ackley function [Ackley, 1987]. The Ackley function is a multimodal function with a large number of local optima:



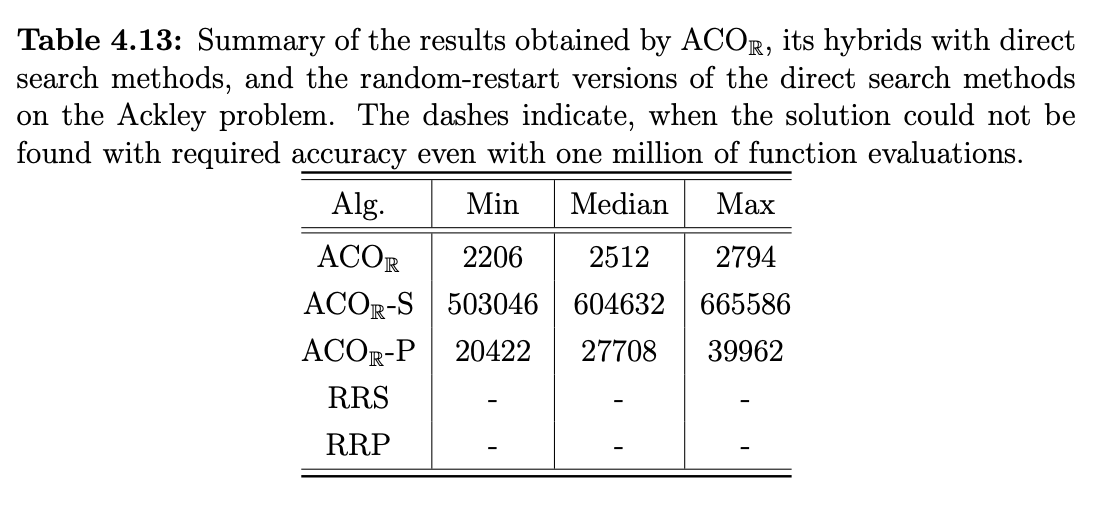
For our experimentation, we used the Ackley function with n = 10 dimensions. Usually in the literature, the domain of the Ackley function is defined as X ∈ (−32, 32)n. This however leads to a bias performance of population-based methods, as the optimum is found exactly in the center of the search space, X ∗ = {0, 0, ..., 0}. In order to avoid this, we used the skewed initialization, that is, we used the domain of the same size, but shifted off the center, X ∈ (−16, 48)n.

As before, we have chosen the parameters for ACOm, ACOr-S, and ACOr-P using the F-Race method. Table 4.12 presents these parameters.



For the mACOr we used the following Parameters: *m* = 2 : *ξ* = 0.5 : *q* = 0.001 : *k* = 50  
and : *m* = 2 : *ξ* = 0.5 : *q* = 0.01 : *k* = 0

The performance of the algorithms was compared using the number of function evalua- tions needed to reach the accuracy of e = 10−4. We have done 100 independent runs of each algorithm. Table 4.13 presents the values of the minimum, median, and maximum number of the objective function evaluations needed to achieve the required accuracy by each of the algorithms.



The results are quite different than on the other two test problems. Both random-restart

versions of the direct search heuristics failed to find the optimum of the Ackley function even when allowed one million of function evaluations. This is most likely due to the fact that the Ackley function has so many local optima that finding a starting point that would allow the direct search method to reach the global optimum is very difficult when doing it at random. A higher level strategy is probably needed to guide the search process towards the promising regions of the search space. This is supported by the fact that the hybrids of ACOm with either of the direct search methods managed to find the optimum.

However, the most interesting result is the one obtained by the ACOm algorithm alone. It performs on average 10 times better than ACOr and Powell hybrid and 200 times better than ACOr and Simplex hybrid. The poor performance of the hybridized versions of ACOm may be explained. While ACOm is able to guide the search process towards the global optimum, the direct search methods used as local search use many function evaluations in order to find each local optimum on the way. However, this is not really necessary to converge to the global optimum. The ACOm algorithm when used alone can therefore quickly converge towards the global optimum, and only then intensify the search to reach the required accuracy. Hence, for problems with many local optima and a general structure that may be exploited by a higher level strategy, ACOm appears to be much more efficient than any of the two direct search methods investigated.

## 4.3 Conclusions

Since the proposal of the first ACO algorithms in 1991, the field of ACO has attracted a large number of researchers and nowadays a large number of research results of both experimental and theoretical nature exist. By now ACO is a well- established metaheuristic. The importance of ACO is exemplified by (i) the biannual conference ANTS (International conference on Ant Colony Optimization and Swarm Intelligence (http://iridia.ulb.ac.be/∼ants/), where researchers meet to discuss the properties of ACO and other ant algorithms, both theoretically and experimentally; (ii) the IEEE Swarm Intelligence Symposium series; and (iii) various conferences on metaheuristics and evolutionary algorithms, where ACO is a central topic.  
In this research work we have

Chapter 5: Results and Code Review  
This section reviews the code that was written to solve the sample Continuous Problems.

Comparison Methodology: *It is important to emphasize that, the comparison of algorithms for continuous optimization is usually not done based on CPU time unlike combinatorial optimization. A vast majority of the papers on continuous optimization algorithms use the number of function evaluations needed to achieve a certain solution quality [Kern et al., 2004; Bilchev and Parmee, 1995; Monmarch ́e et al., 2000; Dr ́eo and Siarry, 2004] as criterion of comparison. This approach offers several key advantages, namely:   
1. It solves the problem of the algorithms being implemented using different programming languages;   
2. It is insensitive to the code-optimization skills of the programmer (or to the compiler used);   
3. It allows comparing easily the results obtained on different machines.*

*The drawback of this approach is that it does not take into consideration the time-complexity of the algorithms being compared.*

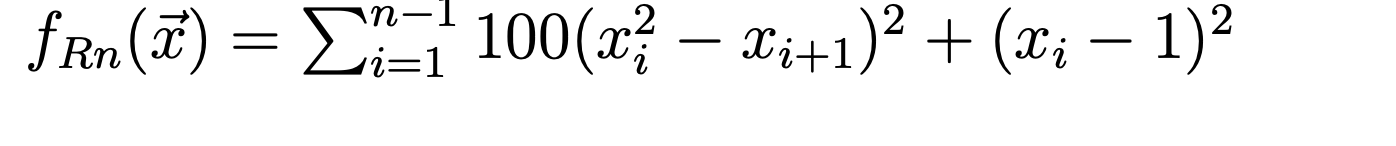
*In view of the other numerous disadvantages of using the CPU time as a criterion, it is an acceptable methodology, hence its adoption in this work. Also, in case of continuous optimization problems, usually the majority of the time of the execution of the algorithm is spent on evaluating the objective function, so in general the measuring the number of function evaluations is a good way to also approximate the time needed for the algorithm to run.*

*The use of the number of function evaluations as a criterion allows us to run the experiments only with ACOm and compare the results obtained to those found in the literature. Additionally, in order to ensure a fair comparison, we replicate carefully the experimental setup (in particular: initialization interval, parameter tuning methodology, and termination condition) used by the competing algorithms.*

Programming Language: Python

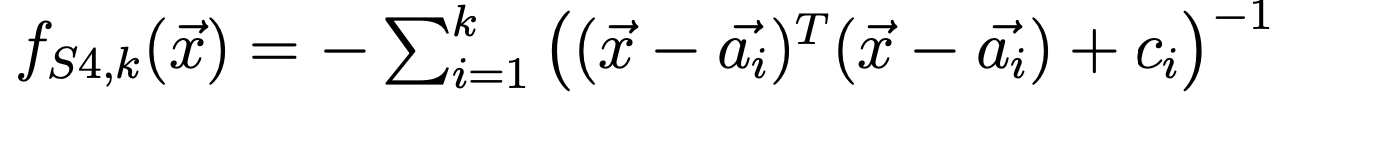
Version: Py3  
  
IDE: PyCharm

Problem: Implementing ACOR for Direct Search Methods   
  
GIT Repository: https://github.com/robo-mac/Unilag-MSc-modACOr

**Rosenbrock Function:**  


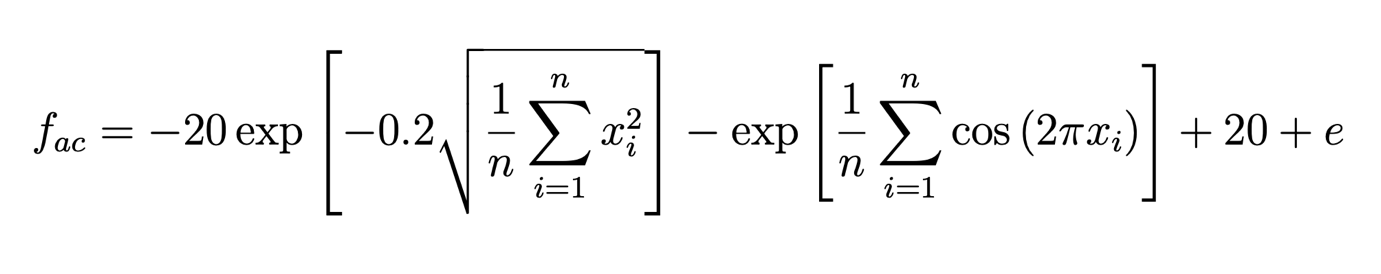
|  |  |
| --- | --- |
| Interpretation | *f(x,y)=(x-1)\*\*2 +b(x\*\*2-y)\*\*2* |

**Shekel Function:**



|  |  |
| --- | --- |
| Interpretation | *f(x,y)=(((x-b)\*\*C) \*(x-b)+a)\*\*-1* |

**Ackley Function:**



|  |  |
| --- | --- |
| Interpretation | *f(x,y) = -20* |

Algorithm:

|  |
| --- |
| ACO Metaheuristic |
| Set parameters, initialize artificial pheromone trails **while** Termination Criteria Not Met **do**  Construct Ant Solutions Apply Local Search Update artificial pheromone  **end while** |

## Code:

*'''  
 ==============================================================  
 Ant Colony Optimization algorithm for continuous domains ACO\_R  
 ==============================================================  
  
 author: GreenBrush  
 Github:  
'''*import math  
from scipy.stats import norm  
import matplotlib.pyplot as plt  
from collections import defaultdict  
from operator import itemgetter  
import csv  
from time import time  
import multiprocessing  
import datetime  
import os  
import sys  
import shutil  
import numpy as np  
import random  
  
  
  
print("Kindly select one of the 3 Standard Problems you would like to solve on this run:")  
print("For Ackleys Function: Key in A")  
print("For Rosenbrock Function: Key in R")  
print("For Shekel Function: Key in S")  
std\_problem = input ("Please key in a value: ")  
  
  
if std\_problem in 'A' or 'a':  
 std\_problem = "Ackley's Function"  
elif std\_problem in 'R' or 'r':  
 std\_problem = "Rosenbrock Function"  
elif std\_problem in 'S' or 's':  
 std\_problem = "Shekel Function"  
else:  
 sys.exit("Kindly insert a valid key")  
  
  
  
  
def evaluator(x):  
 *'''Evaluator function, returns fitness and responses values'''* # give the normalized candidates values inside the real design space  
 x = [10\*i-5 for i in x]  
 # calculate fitness  
 f = (sum([math.pow(i,4)-16\*math.pow(i,2)+5\*i for i in x])/2)  
 # calculate values for other responses  
 res = {'r1':f-5,'r2':2\*f}  
 fitness = dict(Obj=f,\*\*res)  
 return fitness  
  
  
def mp\_evaluator(x):  
 *'''Multiprocessing evaluation'''* # ste number of cpus  
 nprocs = 2  
 # create pool  
 pool = multiprocessing.Pool(processes=nprocs)  
 results = [pool.apply\_async(evaluator,[c]) for c in x]  
 pool.close()  
 pool.join()  
 f = [r.get()['Obj'] for r in results]  
 for r in results:  
 del r.get()['Obj']  
 # maximization or minimization problem  
 maximize = False  
 return (f, [r.get() for r in results],maximize)  
  
  
def ackley\_function(x1, x2):  
 # returns the point value of the given coordinate  
 part\_1 = -0.2 \* math.sqrt(0.5 \* (x1 \* x1 + x2 \* x2))  
 part\_2 = 0.5 \* (math.cos(2 \* math.pi \* x1) + math.cos(2 \* math.pi \* x2))  
 value = math.exp(1) + 20 - 20 \* math.exp(part\_1) - math.exp(part\_2)  
 # returning the value  
 return value  
  
  
def ackley\_function\_range(x\_range\_array):  
 # returns an array of values for the given x range of values  
 value = np.empty([len(x\_range\_array[0])])  
 for i in range(len(x\_range\_array[0])):  
 # returns the point value of the given coordinate  
 part\_1 = -0.2 \* math.sqrt(  
 0.5 \* (x\_range\_array[0][i] \* x\_range\_array[0][i] + x\_range\_array[1][i] \* x\_range\_array[1][i]))  
 part\_2 = 0.5 \* (math.cos(2 \* math.pi \* x\_range\_array[0][i]) + math.cos(2 \* math.pi \* x\_range\_array[1][i]))  
  
 value\_point = math.exp(1) + 20 - 20 \* math.exp(part\_1) - math.exp(part\_2)  
 value[i] = value\_point  
 # returning the value array  
 return value  
  
def initialize(ants,var):  
 *'''Create initial solution matrix'''* X = np.random.uniform(low=0,high=1,size=(ants,var))  
 return X  
  
  
def init\_observer(filename,matrix,parameters,responses):  
 *'''Initial population observer'''* p = []  
 r = []  
 f = []  
 res = ['{0:>10}'.format(i)[:10] for i in responses]  
 par = ['{0:>10}'.format(i)[:10] for i in parameters]  
 for i in range(len(matrix)):  
 p.append(matrix[i][0:len(parameters)])  
 r.append(matrix[i][len(parameters):-1])  
 f.append(matrix[i][-1])  
 r = np.array(r)  
 p = np.array(p)  
  
 for i in range(len(r)):  
 r[i] = ['{0:>10}'.format(r[i][j])[:10] for j in range(len(responses))]  
  
 for i in range(len(p)):  
 p[i] = ['{0:>10}'.format(p[i][j])[:10] for j in range(len(parameters))]  
  
 f = ['{0:>10}'.format(i)[:10] for i in f]  
  
 iteration = 0  
  
 filename.write('{0:>10}, {1}, {2:>10}, {3}\n'.format('Iteration',', '.join(map(str, par)),'Fitness',', '.join(map(str, res))))  
  
 for i in range(len(matrix)):  
 filename.write('{0:>10}, {1}, {2:>10}, {3}\n'.format(iteration,', '.join(map(str, p[i])),f[i],', '.join(map(str, r[i]))))  
  
  
  
def iter\_observer(filename,matrix,parameters,responses,iteration):  
 *'''Iterations observer'''* p = []  
 r = []  
 f = []  
 for i in range(len(matrix)):  
 p.append(matrix[i][0:len(parameters)])  
 r.append(matrix[i][len(parameters):-1])  
 f.append(matrix[i][-1])  
 r = np.array(r)  
 p = np.array(p)  
  
 for i in range(len(r)):  
 r[i] = ['{0:>10}'.format(r[i][j])[:10] for j in range(len(responses))]  
  
 for i in range(len(p)):  
 p[i] = ['{0:>10}'.format(p[i][j])[:10] for j in range(len(parameters))]  
  
 f = ['{0:>10}'.format(i)[:10] for i in f]  
  
 for i in range(len(matrix)):  
 filename.write('{0:>10}, {1}, {2:>10}, {3}\n'.format(iteration,', '.join(map(str, p[i])),f[i],', '.join(map(str, r[i]))))  
  
  
def correct\_par(filename,par):  
 *"""Replace normalized values with real"""* columns = defaultdict(list)  
 with open(filename) as f:  
 reader = csv.DictReader(f,skipinitialspace=True)  
 for row in reader:  
 for (k,v) in row.items():  
 columns[k].append(v)  
 keys = columns.keys()  
 for p in par:  
 if p in keys:  
 col = []  
 for i,k in enumerate(columns[p]):  
 k = float(k)  
 if p in par:  
 n = 10\*k-5  
 col.append(n)  
 columns[p] = col  
  
 outputfile = filename  
  
 file = open(outputfile,'w+')  
 head = []  
 head.append('Iteration')  
 for i in par:  
 head.append(i)  
 head.append('Fitness')  
 for i in keys:  
 if i not in head:  
 head.append(i)  
 par = ['{0:>10}'.format(i)[:10] for i in par]  
 line = ['{0:>10}'.format(l)[:10] for l in head]  
 file.write('{0}\n'.format(', '.join(map(str, line))))  
 for i in range(len(columns.get('Iteration'))):  
 line = []  
 for j in head:  
 line.append(columns.get(j)[i])  
 line = ['{0:>10}'.format(l)[:10] for l in line]  
 file.write('{0}\n'.format(', '.join(map(str, line))))  
 file.close()  
  
  
def formatTD(td):  
 *""" Format time output for report"""* days = td.days  
 hours = td.seconds // 3600  
 minutes = (td.seconds % 3600) // 60  
 seconds = td.seconds % 60  
 return '%s days %s h %s m %s s' % (days, hours, minutes, seconds)  
  
def evolve(display):  
 *'''Executes the optimization'''* start\_time = time()  
  
 # number of variables  
 parameters\_v = ['x1', 'x2']  
 response\_v = ['r1','r2']  
  
 # create output file  
 projdir = os.getcwd()  
 if std\_problem in "Rosenbrock Function":  
 ind\_file\_name = '{0}/RosenbrockResults.csv'.format(projdir)  
 elif std\_problem in "Ackley's Function":  
 ind\_file\_name = '{0}/AckleyResults.csv'.format(projdir)  
 elif std\_problem in "Shekel Function":  
 ind\_file\_name = '{0}/Shekel\_Results.csv'.format(projdir)  
 else:  
 ind\_file\_name = '{0}/results.csv'.format(projdir)  
 ind\_file = open(ind\_file\_name, 'w')  
  
 # number of variables  
 nVar = len(parameters\_v)  
 # size of solution archive  
 nSize = 1000  
 # number of ants  
 nAnts = 10  
  
 # parameter q  
 q = 0.3  
  
 # standard deviation  
 qk = q\*nSize  
  
 # parameter xi (like pheromone evaporation)  
 xi = 0.85  
  
 # maximum iterations  
 maximumiterations = 500  
 # tolerance  
 errormin = 0.01  
  
 # bounds of variables  
 Up = [1]\*nVar  
 Lo = [0]\*nVar  
  
 # initilize matrices  
 S = np.zeros((nSize,nVar))  
 S\_f = np.zeros((nSize,1))  
  
 plt.figure()  
  
  
 # initialize the solution table with uniform random distribution and sort it  
 print ('-----------------------------------------')  
 print ('Starting initilization of solution matrix for ', std\_problem)  
 print ('-----------------------------------------')  
  
 Srand = initialize(nSize,nVar)  
 f,S\_r,maximize = mp\_evaluator(Srand)  
  
 S\_responses = []  
  
 for i in range(len(S\_r)):  
 S\_f[i] = f[i]  
 k = S\_r[i]  
 row = []  
 for r in response\_v:  
 row.append(k[r])  
 S\_responses.append(row)  
  
 # add responses and "fitness" column to solution  
 S = np.hstack((Srand,S\_responses,S\_f))  
 # sort according to fitness (last column)  
 S = sorted(S, key=lambda row: row[-1],reverse = maximize)  
 S = np.array(S)  
  
 init\_observer(ind\_file,S,parameters\_v,response\_v)  
  
 # initilize weight array with pdf function  
  
 '''if'''  
 w = np.zeros((nSize))  
 for i in range(nSize):  
 if std\_problem in "Rosenbrock Function":  
 int\_eq = 1/(qk\*2\*math.pi)\*math.exp(-math.pow(i,2)/(2\*math.pow(q,2)\*math.pow(nSize,2)))  
 elif std\_problem in "Ackley's Function":  
 int\_eq = 1/(qk\*2\*math.pi)\*math.exp(-math.pow(i,2)/(2\*math.pow(q,2)\*math.pow(nSize,2)))  
 elif std\_problem in "Shekel Function":  
 int\_eq = 1/(qk\*2\*math.pi)\*math.exp(-math.pow(i,2)/(2\*math.pow(q,2)\*math.pow(nSize,2)))  
 else:  
 int\_eq = 1/(qk\*2\*math.pi)\*math.exp(-math.pow(i,2)/(2\*math.pow(q,2)\*math.pow(nSize,2)))  
 w[i] = int\_eq  
  
  
 if display:  
 x = []  
 y = []  
 for i in S:  
 x.append(i[0])  
 y.append(i[1])  
  
 plt.scatter(x,y)  
 plt.xlim(0,1)  
 plt.ylim(0,1)  
 plt.pause(2)  
 plt.cla()  
  
 # initialize variables  
 iterations = 1  
 best\_par = []  
 best\_obj = []  
 best\_sol = []  
 best\_res = []  
 worst\_obj = []  
 best\_par.append(S[0][:nVar])  
 best\_obj.append(S[0][-1])  
 best\_sol.append(S[0][:])  
 best\_res.append(S[0][nVar:-1])  
 worst\_obj.append(S[-1][-1])  
  
 stop\_condition = 0  
  
 # iterations  
 while True:  
 print ('-----------------------------------------')  
 print ('Iteration', iterations)  
 print ('-----------------------------------------')  
 print(best\_sol)  
 # choose Gaussian function to compose Gaussian kernel  
 p = w/sum(w)  
  
 # find best and index of best  
 max\_prospect = np.amax(p)  
 ix\_prospect = np.argmax(p)  
 selection = ix\_prospect  
  
 # calculation of G\_i  
 # find standard deviation sigma  
 sigma\_s = np.zeros((nVar,1))  
 sigma = np.zeros((nVar,1))  
 for i in range(nVar):  
 for j in range(nSize):  
 sigma\_s[i] = sigma\_s[i] + abs(S[j][i] - S[selection][i])  
 sigma[i] = xi / (nSize -1) \* sigma\_s[i]  
  
  
 Stemp = np.zeros((nAnts,nVar))  
 ffeval = np.zeros((nAnts,1))  
 res = np.zeros((nAnts,len(response\_v)))  
 for k in range(nAnts):  
 for i in range(nVar):  
 Stemp[k][i] = sigma[i] \* np.random.random\_sample() + S[selection][i]  
 if Stemp[k][i] > Up[i]:  
 Stemp[k][i] = Up[i]  
 elif Stemp[k][i] < Lo[i]:  
 Stemp[k][i] = Lo[i]  
 f,S\_r,maximize = mp\_evaluator(Stemp)  
  
 S\_f = np.zeros((nAnts,1))  
 S\_responses = []  
  
 for i in range(len(S\_r)):  
 S\_f[i] = f[i]  
 k = S\_r[i]  
 row = []  
 for r in response\_v:  
 row.append(k[r])  
 S\_responses.append(row)  
  
 # add responses and "fitness" column to solution  
 Ssample = np.hstack((Stemp,S\_responses,S\_f))  
  
 # add new solutions in the solutions table  
 Solution\_temp = np.vstack((S,Ssample))  
  
 # sort according to "fitness"  
 Solution\_temp = sorted(Solution\_temp, key=lambda row: row[-1],reverse = maximize)  
 Solution\_temp = np.array(Solution\_temp)  
  
 # keep best solutions  
 S = Solution\_temp[:nSize][:]  
  
 # keep best after each iteration  
 best\_par.append(S[0][:nVar])  
 best\_obj.append(S[0][-1])  
 best\_res.append(S[0][nVar:-1])  
 best\_sol.append(S[0][:])  
 worst\_obj.append(S[-1][-1])  
  
 iter\_observer(ind\_file,S,parameters\_v,response\_v,iterations)  
  
 if display:  
 # plot new table  
 x = []  
 y = []  
 for i in S:  
 x.append(i[0])  
 y.append(i[1])  
  
 plt.scatter(x,y)  
 plt.xlim(0,1)  
 plt.ylim(0,1)  
 plt.pause(2)  
  
 if iterations > 1:  
 diff = abs(best\_obj[iterations]-best\_obj[iterations-1])  
 if diff <= errormin:  
 stop\_condition += 1  
  
 iterations += 1  
 if iterations > maximumiterations or stop\_condition > 5:  
 break  
 else:  
 if display:  
 plt.cla()  
  
 ind\_file.close()  
  
 total\_time\_s = time() - start\_time  
 total\_time = datetime.timedelta(seconds=total\_time\_s)  
 total\_time = formatTD(total\_time)  
  
 # fix varibales values in output file  
 correct\_par(ind\_file\_name,parameters\_v)  
  
 best\_sol = sorted(best\_sol, key=lambda row: row[-1],reverse = maximize)  
  
 print ("Best individual:", parameters\_v)  
 print (best\_sol[0][0:len(parameters\_v)])  
 print ("Fitness:")  
 print (best\_sol[0][-1])  
 print ("Responses:", response\_v)  
 print (best\_sol[0][len(parameters\_v):-1])  
  
  
# Executes optimization run.  
# If display = True plots ants in 2D design space  
evolve(display = True)

## Results:

Disclaimer: Due to the length of the results gotten from each run, we only capture the best solutions for each iteration.  
The rest of the results are saved in .csv format with the respective filenames depending on which of the standard problem functions is run.  
The CSV files with the complete run result can be found in the Project Folder from where the code is stored and run. Refer to the GIT listed above to download full folder paths.  
  
/Users/mc/PycharmProjects/AntColony\_MSc/venv/bin/python /Users/mc/PycharmProjects/AntColony\_MSc/ACOR\_Interactive.py

Kindly select one of the 3 Standard Problems you would like to solve on this run:

For Ackleys Function: Key in A

For Rosenbrock Function: Key in R

For Shekel Function: Key in S

Please key in a value: A

-----------------------------------------

Starting initilization of solution matrix for Ackley's Function

-----------------------------------------

-----------------------------------------

Iteration 1

-----------------------------------------

[array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097])]

-----------------------------------------

Iteration 2

-----------------------------------------

[array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097])]

-----------------------------------------

Iteration 3

-----------------------------------------

[array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097])]

-----------------------------------------

Iteration 4

-----------------------------------------

[array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097])]

-----------------------------------------

Iteration 5

-----------------------------------------

[array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.2263842 , 0.19438903, -82.45131875, -154.90263749,

-77.45131875])]

-----------------------------------------

Iteration 6

-----------------------------------------

[array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.2263842 , 0.19438903, -82.45131875, -154.90263749,

-77.45131875]), array([ 0.2263842 , 0.19438903, -82.45131875, -154.90263749,

-77.45131875])]

-----------------------------------------

Iteration 7

-----------------------------------------

[array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.2263842 , 0.19438903, -82.45131875, -154.90263749,

-77.45131875]), array([ 0.2263842 , 0.19438903, -82.45131875, -154.90263749,

-77.45131875]), array([ 0.2263842 , 0.19438903, -82.45131875, -154.90263749,

-77.45131875])]

-----------------------------------------

Iteration 8

-----------------------------------------

[array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.2263842 , 0.19438903, -82.45131875, -154.90263749,

-77.45131875]), array([ 0.2263842 , 0.19438903, -82.45131875, -154.90263749,

-77.45131875]), array([ 0.2263842 , 0.19438903, -82.45131875, -154.90263749,

-77.45131875]), array([ 0.22778813, 0.2148487 , -82.75139101, -155.50278202,

-77.75139101])]

-----------------------------------------

Iteration 9

-----------------------------------------

[array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.21451612, 0.18127563, -81.76433097, -153.52866195,

-76.76433097]), array([ 0.2263842 , 0.19438903, -82.45131875, -154.90263749,

-77.45131875]), array([ 0.2263842 , 0.19438903, -82.45131875, -154.90263749,

-77.45131875]), array([ 0.2263842 , 0.19438903, -82.45131875, -154.90263749,

-77.45131875]), array([ 0.22778813, 0.2148487 , -82.75139101, -155.50278202,

-77.75139101]), array([ 0.22778813, 0.2148487 , -82.75139101, -155.50278202,

-77.75139101])]

Best individual: ['x1', 'x2']

[0.22778813 0.2148487 ]

Fitness:

-77.75139100816185

Responses: ['r1', 'r2']

[ -82.75139101 -155.50278202]

Process finished with exit code 0

# References

Afshar, A., Massoumi, F., Afshar, A., & Mariño, M. A. (2015). State of the Art Review of Ant Colony Optimization Applications in Water Resource Management. *Springer Science+Business Media Dordrecht 2015*, 3891-3904.

Anand, N., & Rajeshwar, S. (2017). Ant Colony Optimization (ACO) based Routing Protocols for Wireless Sensor Networks (WSN): A Survey. *(IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 8, No. 2*, 148-155.

Dorigo, M., Birattari, M., & tzle, T. S. (2006, November). Ant Colony Optimization (Artificial Ants as a Computational Intelligence Technique). *IEEE COMPUTATIONAL INTELLIGENCE MAGAZINE*, pp. 28-39.

Gendreau, M., & Potvin, J.-Y. (2010). International Series in Operations Research & Management Science. In M. G.-Y. Potvin, *Handbook of Metaheuristics* (pp. 227-264). London: Springer.

Jr., I. F., Yang, X.-S., Fister, I., Brest, J., & Fister, D. (2013). A Brief Review of Nature-Inspired Algorithms for Optimization. *ELEKTROTEHNISˇKI VESTNIK*, 1-7.

Jun-man, K., & Yi, Z. (2012). Application of an Improved Ant Colony on Generalised Travelling Salesman Problem. *Energy Procedia 17*, 319-325.

Lee, C.-Y., Lee, Z.-J., Lin, S.-W., & Ying, K.-C. (2008). An enhanced ant colony optimization (EACO) applied to capacitated vehicle routing problem. *Springer Science+Business Media, LLC 2008*, 88-95.

Li, H., & Zhang, H. (2013). Ant colony optimization-based multi-mode scheduling under renewable and nonrenewable resource constraints. *Automation in Construction*, 1-2.

LIU, X. (2017). Routing Protocols Based on Ant Colony Optimization in Wireless Sensor Networks: A Survey. *IEEE Access [SPECIAL SECTION ON INTELLIGENT SYSTEMS FOR THE INTERNET OF THINGS]*, 26303-26317.

Mohan, B. C., & Baskaran, R. (2012). A survey: Ant Colony Optimization based recent research and implementation on several engineering domain. *Expert Systems with Applications 39*, 4618–4627.

Neto, R. T., & Filho, M. G. (2013). Literature review regarding Ant Colony Optimization applied to scheduling problems: Guidelines for implementation and directions for future research. *Engineering Applications of Artificial Intelligence 26*, 150-161.

Nour, C., Khelifa, E. H., & Belmadani, A. (2020). New Approach for Continuous and Discrete Optimization: Optimization by Morphological Filters. *Heuristics for Optimization and Learning*, 425-440.

Ostfeld, A. (2011). Ant Colony Optimization for Water Resources Systems Analysis – Review and Challenges. *Intechopen - Ant Colony Optimization - Methods and Applications*, 147-158.

Otero, F. E., Freitas, A. A., & Johnson, C. G. (2012). Inducing Decision Trees with an Ant Colony Optimization Algorithm. *Applied Soft Computing, Volume 12, Issue 11*, 3615–3626.

Pedemonte, M., Nesmachnow, S., & Cancela, H. (2011). A survey on parallel ant colony optimization. *Applied Soft Computing 11*, 5181–5197.

Pourtakdoust, S. H., & Nobahari, H. (2004). An Extension of Ant Colony System to Continuous Optimization Problems. *International Workshop on Ant Colony Optimization and Swarm Intelligence*, 294-301.

Stutzle, T., ́pez-Ibanez, M. L., Pellegrini, P., Maur, M., Oca, M. M., Birattari, M., & Dorigo, M. (2010). Parameter Adaptation in Ant Colony Optimization. *IRIDIA – Technical Report Series Technical Report No. TR/IRIDIA/2010-002*, 1-24.

Tabakhi, S., Moradi, P., & Akhlaghian, F. (2014). An unsupervised feature selection algorithm based on ant colony optimization. *Engineering Applications of Artificial Intelligence 32*, 112–123.

Tasgetirenc, F., Liangd, J., P.N., Q.-K., & M., S. (2010). A self-adaptive global best harmony search algorithm for continuous optimization problem. *Applied Mathematics and Computation*, 830-848.

Yang, Q., Chen, W.-N., Yu, Z., Gu, T., Li, Y., Zhang, H., & Zhang, J. (2017). Adaptive Multimodal Continuous Ant Colony Optimization. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, VOL. 21, NO. 2*, 191-205.

ZHANG, H., WANG, X., MEMARMOSHREFI, P., & HOGREFE, D. (2017). A Survey of Ant Colony Optimization Based Routing Protocols for Mobile Ad Hoc Networks. *IEEE Access*, 24139-24161.

̃ez, M. L.-I., & ̈tzle, T. S. (2011). The Automatic Design of Multi-Objective Ant Colony Optimization Algorithms. *IRIDIA – Technical Report Series ISSN 1781-3794*, 1-38.