

THESIS DOCUMENT

**USING SIMULATED ANNEALING
AND ANT-COLONY
OPTIMIZATION ALGORITHMS
TO SOLVE THE SCHEDULING
PROBLEM**

July 10, 2012

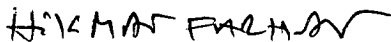
Nader Chmeit
Notre Dame University
Zouk Mosbeh - Lebanon
nbchmait@ndu.edu.lb

Approved by:

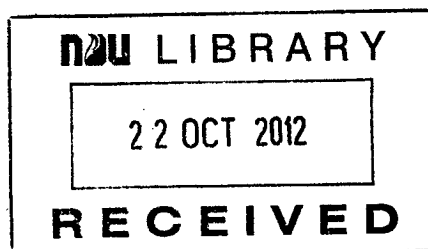
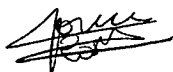
Dr. khalil Challita
Advisor



Dr. Hikmat farhat
Member of Committee



Dr. Rosy Aoun
Member of Committee



Contents

List of Figures	5
Abstract	7
Introduction	8
1 The Scheduling Problem	11
1.1 Problem Formulation	11
2 Heuristics And Optimization Algorithms	13
2.1 Simulated Annealing	13
2.1.1 Initial Temperature	16
2.1.2 Equilibrium State	17
2.1.3 Cooling Schedule	18
2.1.4 Stopping Condition	19
2.1.5 Performance of the SA Algorithm	20
2.2 Ant Colony Optimization Algorithm	21
2.3 Combinatorial Optimization Problems	23
2.4 The Pheromone Model	23
2.5 The Double-Bridge Experiment	25
2.6 Ant System (AS)	26
2.7 Complexity Analysis of Ant Colony Optimization	27
3 Related Work	29
4 SA Applied To The Scheduling Problem	33
4.1 Our Approach Using SA	33
4.2 Empirical Results (SA)	37
5 ACO Applied To The Scheduling Problem	41
5.1 Our Approach Using ACO	41
5.2 Pheromone Update	43

5.3	Empirical Results (ACO)	43
5.4	Performance Analysis: ACO vs. SA	45
6	Conclusion	49
6.1	Summary of the Main Results	49
6.2	Future Work	49

List of Figures

2.1	Double-bridge Experiment	26
4.1	Simple Examination Schedule	34
4.2	Variation in cost values with respect to temperature decrement	38
4.3	Variation in cost values with respect to temperature decrement after constraining unfeasible neighbor configurations	39
5.1	ACO - Cost values at subsequent iterations (Hard Schedule) .	44
5.2	ACO - Cost values at subsequent iterations (Loose Schedule) .	44

Abstract

The scheduling problem is one of the most challenging problems faced in many different areas of everyday life. This problem can be formulated as a combinatorial optimization problem, and it has been solved with various methods using heuristics and intelligent algorithms. We present in this research an efficient solution to the scheduling problem using two different heuristics namely Simulated Annealing and Ant Colony Optimization. A study comparing the performances of both solutions is described and the results are analyzed.

Introduction

A wide variety of scheduling (timetabling) problems have been described in the literature of computer science during the last decade. Some of these problems are: the weekly course scheduling done at schools and universities [7], examination timetables [37], airline crew and flight scheduling problems [31], job and machine scheduling [43], train timetabling problems [49] . . .

Many definitions of the scheduling problem exist. A general definition was given by A. Wren in *1996* as:

“The allocation of, subjects to constraints, of given resources to objects being placed in space and time, in such a way as to satisfy as nearly as possible a set of desirable objectives” [47].

Another way of looking at the scheduling problem is to consider it as a timetable consisting of four finite sets:

- a set of meetings in time,
- a set of available resources such as rooms and people,
- a set of available time-slots,
- a set of constraints.

Some of these resources may be specified by the problem in hand, and some others must be allocated as part of the solution [7].

Cook proved the timetabling problem to be NP-complete [13] in *1971*, then karp showed in *1972* that the general problem of constructing a schedule for a set of partially ordered tasks among k-processors (where k is a variable) is NP-complete [28]. Four years later, a proof showing that even a further restriction on the timetabling problem (e.g the restricted timetable problem) will always lead to an NP-complete problem [22]. This means that the running time for any algorithms currently known to guarantee an optimal solution is an exponential function of the size of the problem. The complexity results of the some of the scheduling problems were classified and simulated in [6] and their reduction graphs were plotted.

The exam scheduling at Notre-Dame University is still done manually using solutions from previous years and altering them, in such a way to meet the present constraints related to the number of students attending the upcoming exams, the number of exams to be scheduled, the rooms and time available for the examination period. We propose an improved methodology where the generation of the examination schedules is feasible, automated, faster and less error prone.

Even though we will present in this research our solution in the context of the *exam scheduling problem*, we can generalize it to solve many different scheduling problems with some minor modifications regarding the variables related the the problem in hand and the resources available.

An informal definition of the exam scheduling problem is the following: a combinatorial optimization problem that consists of scheduling a number of examinations in a given set of exam sessions so as to satisfy a given set of constraints. As Carter states in [37] the basic challenge faced when solving this problem is to “schedule examinations over a limited time period so as to avoid conflicts and to satisfy a number of side-constraints”. These constraints can be split into hard and soft constraints where the hard constraints must be satisfied in order to produce a feasible or acceptable solution, while the violation of soft constraints should be minimized since they provide a measure of how good the solution is with regard to the requirements [8].

The main hard constraints [20] are given below:

1. No student is scheduled to sit for more than one exam simultaneously. So any 2 exams having students in common should not be scheduled in the same period.
2. An exam must be assigned to exactly one period.
3. Room capacities must not be violated. So no exam could take place in a room that has not a sufficient number of seats.

As for the soft constraints, they vary between different academic institutions and depend on their internal rules [11]. The most common soft constraints are:

1. Minimize the total examination period or, more commonly, fit all exams within a shorter time period.
2. Increase students’ comfort by spacing the exams fairly and evenly across the whole group of students. It is preferable for students not to sit for exams occurring in 2 consecutive periods.
3. Schedule the exams in specific order, such as scheduling the maths and sciences related exams in the morning time.
4. Allocate each exam to a suitable room. Lab exams for example, should be held in the corresponding labs.
5. Some exams should be scheduled in the same time.

We will attempt to satisfy all the hard-constraints listed above. Regarding the soft-constraints, we will address the first 2 points, whereby we try to enforce the following:

1. Shorten the examination period as much as possible
2. A student has no more than one exam in two consecutive time-slots of the same day.

In other words, we will try to create an examination schedule spread among the shortest period of time and therefore use the minimum number of days to schedule all the exams, also, we spread the exams shared by students among the exam schedule in such a way that they are not scheduled in consecutive time-slots.

This thesis is organized as follows. Chapter 1 presents the mathematical formulation of the exam scheduling problem; Chapter 2 describes the main structures of the (SA) Simulated Annealing and ACO (Ant Colony Optimization) algorithms. We list and describe in Chapter 3 some of the heuristics and algorithms that are used in solving the scheduling problem. In Chapter 4 and 5 we present and discuss our own approach to solving the exam scheduling problem using the Simulated Annealing and Ant Colony Optimization algorithms respectively. Finally we give a brief conclusion.

Chapter 1

The Scheduling Problem

1.1 Problem Formulation

We will use a variation of D. de Werra's definition of the timetabling problem [14]. Note that a class consists of a set of students who follow exactly the same program. So Let:

- $C = \{c_1, \dots, c_n\}$ be a set of classes
- $E = \{e_1, \dots, e_n\}$ a set of exams
- $S = \{s_1, \dots, s_m\}$ the set of students
- $R = \{r_1, \dots, r_x\}$ the set of rooms available
- $D = \{d_1, \dots, d_z\}$ the set of the examination days
- $P = \{p_1, \dots, p_s\}$ the set of periods (sessions) of the examination days.

Since all the students registered in a class c_i follow the same program, we can therefore associate for each class c_i an exam e_i to be included in the examination schedule. So all the students registered in class c_i will be required to pass the exam e_i .

We will use the notation below:

•

$$esp_{ijk} = \begin{cases} 1, & \text{if exam } e_i \text{ and student } s_j \text{ meet at period } p_k \\ 0, & \text{otherwise} \end{cases}$$

•

$$sc_{ji} = \begin{cases} 1, & \text{if student } s_j \text{ is taking class } c_i \\ 0, & \text{otherwise} \end{cases}$$

•

$$se_{ji} = \begin{cases} 1, & \text{if student } s_j \text{ has exam } e_i \\ 0, & \text{otherwise} \end{cases}$$

•

$$ep_{ui} = \begin{cases} 1, & \text{if exam } e_i \text{ is held in period } p_u \\ 0, & \text{otherwise} \end{cases}$$

•

$$capacity_{zi} = \begin{cases} \text{the number of seats in room } r_z, & \text{if } r_z \text{ holds exam } e_i \\ \text{null}, & \text{otherwise} \end{cases}$$

We shall assume that all exam sessions have the same duration (say one period of 2 hours). We recap that the problem is, given a set of periods, we need to assign each exam to some period in such a way that no student has more than one exam at the same time, and the room capacity is not breached. We therefore have to make sure that the equations (constraints) below are always satisfied [14]:

1. $\forall s_j \in S : \sum_{j=1}^m se_{ji} \leq capacity_{zi}$
2. $\forall s_j \in S, \forall e_i \in E, \text{ and one } p_k \in P : \sum_{j=1}^m esp_{ijk} \leq 1$
3. $\forall e_i \in E, \forall p_k \in P : \sum_{u=1}^s ep_{ui} = 1$

These equations reveal only the hard constraints which are critical for reaching a correct schedule. They must always evaluate to true otherwise we will end up by an erroneous schedule. So our aim is to find a schedule that meets all the hard constraints and try to adhere as much as possible to the soft constraints.

Chapter 2

Heuristics And Optimization Algorithms

2.1 Simulated Annealing

The idea behind simulate annealing (SA) comes from a physical process known as annealing [35]. Annealing happens when you heat a solid past its melting point and then cool it. If we cool the liquid slowly enough, large crystals will be formed, on the other hand, if the liquid is cooled quickly the crystals will contain imperfections. The algorithm simulates the cooling process by gradually lowering the temperature of the system until it converges to a steady, frozen state [29]. This allows the system to settle into a low energy state without getting trapped in a local minimum [14].

SA exploits this analogy with physical systems in order to solve combinatorial optimization problems [14]. A *combinatorial optimization problem* [27] is regarded as an optimization/minimization problem where we try to find optimal solutions over a well defined discrete space. Formally, a combinatorial problem is briefly described as follows:

Let $E = \{e_1, e_2, \dots, e_n\}$ be a finite set of all solutions, S a set of feasible solutions defined over E , and $f : S \rightarrow \mathbb{R}$ an objective function. A combinatorial optimization problem is to find a solution $i \in S$ which minimizes f over S .

There are many articles and papers that define and explain the SA algorithm. Most of these articles give similar definitions and refer to a paper published in 1953 by Metropolis [35]. We will present the Simulated Annealing algorithm and the physical analogy on which it is based, as it was described by R. W. Eglese [21].

SA is a type of local search algorithm with some variations. If we look at

a simple form of local search, like the *Hill Climbing* algorithm, we notice that it starts with an initial solution usually chosen at random. Then a neighbor of this solution is generated by some suitable mechanism depending on the problem we are solving, and the change in the cost of the new solution is calculated [21]. If a reduction in cost is found, the current solution is replaced by the generated neighbor, otherwise the current solution is retained. The process is repeated until no further improvement can be found in the neighborhood of the current solution. We give below the *Hill Climbing* local search algorithm:

Algo-I : **Hill Climbing** [21]

Select an initial state $i \in S$

Repeat

Generate state j , a neighbor of i

Calculate $\varphi = f(j) - f(i)$

If $\varphi \leq 0$

Then $i := j$

Until $f(j) \geq f(i), \forall j$ in the neighborhood of i

As shown in the algorithm above, the local search algorithm works by:

1. First generating an initial solution i and calculating its cost $f(i)$.
2. The second step consists of generating a neighbor j of this solution and calculating its cost $f(j)$.
3. Then we calculate the change in cost φ which is the difference between the costs of the new solution $f(j)$ and the previous one $f(i)$.
4. If a reduction in cost is found, we replace the current solution by the generated neighbor.
5. We repeat until no further improvements can be found.

Since we are calculating the change in cost φ and choosing the solution with the lowest cost, the above algorithm solves a minimization problem. It can easily be used for solving a maximization problem by checking for solutions with higher costs instead.

G. Kendall [29] states that “hill climbing” suffers from the problem of getting stuck at local minima (or maxima depending on whether it is a minimization or an optimization problem). There are many techniques described in the literature to try to overcome these problems as:

- try a hill climbing algorithm using different starting points, or
- increase the size of the neighborhood so that we consider more of the search space at each move.

But neither of these techniques has proved satisfactory in practice [29]. Simulated annealing solves this problem by allowing worse moves (moves that increase the cost of the solution) to be taken in certain cases. These moves which increase the value of f are called uphill steps. This strategy helps us escape from getting stuck in local minima/maxima (depending on the problem). In other words, SA escapes from local optima due to the probabilistic acceptance of some non-improving neighbors [21]. But to be able to climb out from the local optima, we have to control when SA accepts uphill moves (worse solutions). Uphill moves are accepted or rejected depending on a sequence of random numbers, but with a controlled probability. The probability of accepting a move which causes an increase in f is called the acceptance function and is normally set to:

$$\exp(-\varphi/T) \tag{2.1}$$

where T is a control parameter which corresponds to temperature in the analogy with physical annealing [21]. We can see from the formula above that, as the temperature of the system decreases, the probability of accepting a worse move is decreased and when the temperature reaches zero then only better moves will be accepted which effectively makes simulated annealing act like a hill climbing algorithm [29].

Hence, to avoid being prematurely trapped in a local optimum, SA is started with a relatively high value of T . The algorithm proceeds by attempting a certain number of neighborhood moves at each temperature, while the temperature parameter is gradually dropped. Algo-II below shows the SA algorithm as listed in [21]:

Algo-II Simulated Annealing

Select an initial solution $i \in S$

Select an initial temperature $T_0 > 0$

Select a temperature reduction function α

Repeat*Set repetition counter $n = 0$* **Repeat***Generate state j , a neighbor of i* *calculate $\varphi = f(j) - f(i)$* **If** $\varphi < 0$ **Then** $i := j$ **Else***generate random number $x \in]0,1[$* **If** $x < \exp(-\varphi/t)$ **Then** $i := j$ $n := n + 1$ **Until** $n = \text{maximum neighborhood moves allowed at each temperature}$ **endif****endif***update temperature decrease function α* $T = \alpha \cdot (T)$ **Until** *stopping condition = true.*

There are many heuristics used for choosing the *initial temperature* at which SA starts [38]. We will next introduce many important concepts in SA which are crucial for building efficient solutions. These concepts are listed below:

- Initial Temperature
- Equilibrium State
- Cooling Schedule
- Stopping Condition

2.1.1 Initial Temperature

For the algorithm to work properly, we must choose a starting temperature that is hot enough to allow a move to almost any neighborhood state [29], otherwise we will end up by a solution very similar to the starting one. On the other hand, if we choose a starting temperature that is too high we might be implementing a random search algorithm since the search can freely move to any neighbor even when the neighbor has a non-desirable cost [29]. At very high temperatures we accept all neighbors during the initial phase of the algorithm [38] and the main drawback of this strategy is its high

computational cost.

We have 2 main methods for finding a suitable starting temperature:

1. The Acceptance deviation method and
2. The tuning for initial temperature method

Acceptance deviation method

Before explaining how this method works, we recap that the neighbor solution is accepted with a probability p depending on the energy difference between the new and the old solution. This selection scheme is called the *Metropolis criterion* [38], and therefore SA consists of a series of Metropolis chains at different decreasing temperatures.

The *acceptance deviation method* computes the starting temperature using preliminary experimentations by:

$$k \cdot \sigma \tag{2.2}$$

where σ represents the standard deviation of difference between values of objective functions representing the energy of the system (cost of solution), and

$$k = \frac{-3}{\ln(p)} \tag{2.3}$$

where p denotes the acceptance probability of the next solution.

Tuning for initial temperature method

This method chooses a high temperature as a start, and then reduces the temperature quickly until about 60% of worse moves are accepted and uses this temperature as the starting temperature T_0 [38].

2.1.2 Equilibrium State

The process of selecting and proposing a move is repeated until the system is considered in thermal equilibrium, at this step the system is considered frozen. To reach an equilibrium state at each temperature we have to process a number of sufficient moves [44]. But the number of iterations to achieve at each temperature might be exponential with respect to the size of the problem which means that we need to compromise between the quality of the obtained solutions and the complexity of the cooling schedule [29]. So, either we run a large number of iterations at a few temperatures or, we run a small number of iterations at many temperatures.

Different strategies are used to determine the number of moves to be made in the neighbor solution. They all depend on:

- The size of the problem instance, and
- The size of the neighborhood denoted by $|N(s)|$

Static strategy In this strategy, the number of moves within each iteration is determined before starting the search. We define a proportion y of the neighborhood size $|N(s)|$ to be explored. The number of generated neighbors from a solution s is as large as: $y \times |N(s)|$

The more significant the ratio y the higher the computational cost and the better the results.

Adaptive strategy A simple approach used in this method consists of achieving a predetermined number of iterations without improvement of the best found solution at the same temperature.

2.1.3 Cooling Schedule

Another critical point for the success of the algorithm is the way in which the temperature is decremented. In SA the temperature is decreased gradually such that: $T_i > 0, \forall i$.

If we decrease the temperature slowly, better solutions are obtained but with a more significant computational time. There are many ways in which the temperature can be updated as we will show next.

Static Strategy: Linear

We can use a simple linear cooling schedule where the temperature T is updated as follows:

$$T_i = T_0 - i \times \beta \quad (2.4)$$

where T_i represents the temperature at iteration i , β is a specified constant value and T_0 is the initial temperature.

Dynamic Strategy: geometric and logarithmic approach*Geometric approach*

The geometric approach to this method is to decrement the temperature T corresponding to the equation below:

$$T_{i+1} = T_i \cdot \alpha \quad \text{where } \alpha < 1. \quad (2.5)$$

It is clear from the equation above that the higher the value of α , the longer it will take to decrement the temperature to the stopping criterion. The best assumption is to take α between 0.8 and 0.99 [29].

Logarithmic approach

The following formula is used:

$$T_i = \frac{T_0}{\log(i)} \quad (2.6)$$

where T_i represents the temperature at iteration i and T_0 is the initial temperature.

Even though this cooling schedule is too slow to be applied in practice, it has the property of the convergence proof to a global optimum [44].

Adaptive Strategy

The adaptive cooling schedule depends on the characteristics of the search landscape. The temperature decrease rate is dynamic and it depends on some information obtained during the search. So the adaptive strategy approach carries out a small number of iterations at high temperatures and a large number of iterations at low temperatures [44].

2.1.4 Stopping Condition

The decision of when to stop is a very important measure. Though theory suggests a final temperature equal to 0, this might make the algorithm run for a lot longer especially when we are using a geometric cooling schedule [29]. When the temperature approaches zero the chances of accepting a worse move are almost the same as the temperature being equal to zero, so one can stop the search when the probability of accepting a move is negligible without waiting for T to decrease to zero [29].

According to [44], the stopping criteria is not only governed by the temperature T . Sometimes we reach the stopping condition when:

- no improvements in the solutions' cost are found after some pre-determined number of successive iterations done at several temperature values,
- after a fixed amount of CPU time,
- when we reach the objective function we are searching for.

Finally, we should note that the SA algorithm might not necessarily find the optimal solution to the problem in hand but it is designed to give an acceptable solution within a reasonable computing time without getting trapped in a local maximum or minimum [21].

2.1.5 Performance of the SA Algorithm

In general, the performance analysis of an approximation algorithm [2] concentrates on the following quantities:

- The quality (cost) of the final solution.
- The running time of the algorithm.

Since the Simulated Annealing algorithm is a stochastic algorithm requiring a neighborhood structure to be specified as well as a number of parameters to be provided as part of the cooling schedule, then its performance analysis depends on the problem instance as well as the cooling schedules. Moreover, the performance analysis of SA consists not only on worst-case and average-case analysis, but also on its probabilistic aspects. For any problem, we have to consider its probability distribution for a set of instances and over the set of possible solution. we have two approaches to investigate the performance of SA:

- Theoretical analysis: given a particular problem instance and a cooling schedule, we provide analytical expressions for the quality of the solution and its running time.
- Empirical analysis: the conclusion (with regard to the quality of the solution and its running time) drawn from the results of solving many instances of different combinatorial optimization problems with different cooling schedules.

As part of the theoretical study of the worst-case analysis it was shown [2] that:

- Some cooling schedules can be executed in a number of elementary operations bounded by a polynomial in the problem size.
- Upper bounds can be given for the proximity of the probability distribution of the configurations after generating a finite number of transitions. An upper bound on the quality of the final solution is only known for the *maximum matching problem* which is known to be in P .

As part of the empirical results, compared to repeating descent algorithms (like hill climbing) with different random starting positions, studies of a number of different problems have shown that SA can give significantly better results in the same amount of computing time [21].

2.2 Ant Colony Optimization Algorithm

The second algorithm (ACO) is a meta-heuristic method proposed by *Marco Dorigo* in 1992 in his PhD thesis [15] about optimization, learning and natural algorithms. It is used for solving computational problems where these problems can be reduced to finding good paths through graphs. The ant colony algorithms are inspired by the behavior of natural ant colonies. In real life, ants wander randomly, and upon finding food, they return to their colony while laying down pheromone trail [48]; if other ants find such a path, they are likely not to keep traveling at random, but to instead follow the trail, returning and reinforcing it if they eventually find food. In other words, ants solve their problems by multi-agent cooperation using indirect communication through modifications in the environment. They communicate indirectly via distribution and dynamic change of information (pheromone trails). The weight of these trails reflects the collective search experience exploited by the ants in their attempts to solve a given problem instance [4]. Many problems were successfully solved by ACO, such as the problem of satisfiability, the scheduling problem [34], the traveling sales man problem [18], the frequency assignment problem (FAP) [1] ...

M. Dorigo and T. Stutzle describe the ACO as a set of computational concurrent and asynchronous agents' moves through states of the problem corresponding to partial solutions of the problem to solve [19]. The ants or agents apply a stochastic local decision policy when they move. This policy is based on two parameters, called trails and attractiveness [19]. Therefore, each ant incrementally constructs a solution to the problem each time it moves and

when it completes a solution, the ant evaluates it and modifies the trail value on the components used in its solution which helps in directing the search of the future ants [19]. There is also a mechanism called *trail evaporation* used in ACO. This mechanism decreases all trail levels after each iteration of the AC algorithm. Trail evaporation ensures that unlimited accumulation of trails over some component are avoided and therefore the chances to get stuck in local optimums are decreased [33]. Another optional mechanism that exists in ACO algorithms is *daemon actions*. “Daemon actions can be used to implement centralized actions which cannot be performed by single ants, such as the invocation of a local optimization procedure, or the update of global information to be used to decide whether to bias the search process from a non-local perspective” [33].

The pseudo-code for the ACO as described in [10] is shown below:

Algo-III Ant Colony Optimization algorithm

```

Set parameters, initialize pheromone trails
while termination condition not met do
    ConstructAntSolutions
    ApplyLocalSearch (optional)
    UpdatePheromones
end while

```

According to Maniezzo [33], “ACO algorithms rely mainly on the combination of a priori information about the structure of a promising solution with a posteriori information about the structure of previously obtained good solutions”. Just like the Simulated Annealing algorithm, ACO is also considered to be a meta-heuristic, that is an algorithm that drive basic heuristic trying to escape from local optima [33]. These heuristics are either:

- a constructive heuristic starting from a null solution and adding elements to build a good complete one, or
- a local search heuristic starting from a complete solution and iteratively modifying some of its elements in order to achieve a better one.

This allows us to reach solutions with better costs than those we would have achieved without the use of metaheuristics, even if we make the algorithms run through many iterations [33]. We can control the running

mechanism by constraining or randomizing the set of neighbor solutions to consider in local search (e.g tabu search), or by combining elements taken by different solutions as done in genetic algorithms [33].

2.3 Combinatorial Optimization Problems

Before going into the details of ACO, we will define a model $P = (S, \Omega, f)$ [10] of a combinatorial optimization problem (COP). The model defined in this section is different and more complete than the one described in Section 2.1. The model consists of:

- a search space S defined over a finite set of discrete decision variables $X_i: i = 1, \dots, n$;
- a set Ω of constraints among the variables;
- an objective function $f : S \rightarrow \mathbb{R}$ to be minimized.

The generic variable X_i takes values $\in D_i = \{v_i^1, \dots, v_i^{|D_i|}\}$. “A feasible solution $s \in S$ is a complete assignment of values to variables that satisfies all constraints in Ω ” [10].

2.4 The Pheromone Model

We will use the model of COP described in Section 2.3 above to derive the pheromone model used in ACO [16]. In order to do this we have to go through the following steps:

1. We first instantiate a decision variable $X_i = v_i^j$ (e.g a variable X_i with a value v_i^j assigned from its domain D_i)
2. We denote this variable by c_{ij} and we call it a solution component
3. We denote by C The set of all possible solution components.
4. Then we associate with each component c_{ij} a pheromone trail parameter T_{ij}
5. We denote by τ_{ij} the value of a pheromone trail parameter T_{ij} and we call it the pheromone value

6. This pheromone value is updated later by the ACO algorithm, during and after the search iterations.

These values of the pheromone trails will allow us to model the probability distribution of the components of the solution.

The pheromone model of an ACO algorithm is closely related to the model of a combinatorial optimization problem. Each possible *solution component*, or in other words each possible assignment of a value to a variable define a pheromone value [10, 5]. As described above, the pheromone τ_{ij} is associated with the solution component c_{ij} , which consists of the assignment $X_i = v_i^j$, and the set of all possible solution components is denoted by C .

So how do ACO algorithms build a solution to such combinatorial problems?

ACO uses artificial ants to build a solution to a COP by traversing a fully connected graph $G_C(V, E)$ called the *construction graph*. This graph is obtained from the set of solution components either by representing these components by the set of *vertices* \mathbf{V} of G_c or by the set of its *edges* \mathbf{E} [10].

The artificial ants move from vertex to vertex along the edges of the graph $G_C(V, E)$, incrementally building a partial solution while they deposit a certain amount of pheromone on the components, that is, either on the vertices or on the edges that they traverse.

The amount $\Delta\tau$ of pheromone that ants deposit on the components depends on the quality of the solution found. In subsequent iterations, the ants follow the path with high amounts of pheromone as an indicator to promising regions of the search space [10].

It is also common (optional) to improve the solutions obtained by the ants through a local search before updating the pheromone [16]. We then update the pheromone positively in order to increase the pheromone values associated with good or promising solutions, and sometimes negatively in order to decrease those that are associated with bad solutions [16]. The pheromone update usually consists of:

1. Decreasing all the pheromone values through *pheromone evaporation*, and
2. Increasing the pheromone levels associated with a chosen set of good solutions.

To model the ants' behaviour formally, we consider a finite set of available solution components $C = \{c_{ij}, i = \{1, \dots, n\}, j = \{1, \dots, |D_i|\}$ and a set

of m artificial ants which construct solutions from the elements of the set C [16]. We start from an empty partial solution $s^p = \emptyset$ and extend it by adding a feasible solution s^p using the components from the set $N(s^p) \subseteq C$ (where $N(s^p)$ denotes the set of components that can be added to the current partial solution s^p without violating any of the constraints in Ω). It is clear that the process of constructing solutions can be regarded as a walk through the construction graph $G_C = (V, E)$.

The choice of a solution component from $N(s^p)$ is done probabilistically at each construction step. Before giving the rules controlling the probabilistic choice of the solution components, we will next describe an experiment that was run on real ants called the double-bridge experiment which derived these probabilistic choices.

2.5 The Double-Bridge Experiment

In the double-bridge experiment, we have two bridges connecting the food source to the nest, one of which is significantly longer than the other. Of course, the ants choosing by chance the shorter bridge are the first to reach the nest [16]. Therefore, the short bridge receives pheromone earlier than the long one. This increases the probability that further ants select it rather than the long one due to the higher pheromone concentrations over it. This is shown in the figure 2.1 below.

Based on this observation, a model was developed to depict the probabilities of choosing one bridge over the other. So assuming that at a given moment in time m_1 ants have used the first bridge and m_2 ants have used the second bridge, the probability p_1 for an ant to choose the first bridge is shown in the equation [16] below:

$$p_1 = \frac{(m_1 + k)^h}{(m_1 + k)^h + (m_2 + k)^h} \quad (2.7)$$

where k and h are variables depending on the experimental data.

Obviously the probability p_2 of choosing the second bridge is: $p_2 = 1 - p_1$.

As stated above, the choice of a solution component is done probabilistically at each construction step. Although the exact rules for the probabilistic choice of solution components vary across the different ACO variants, the best known rule is the one of ant systems (AS).

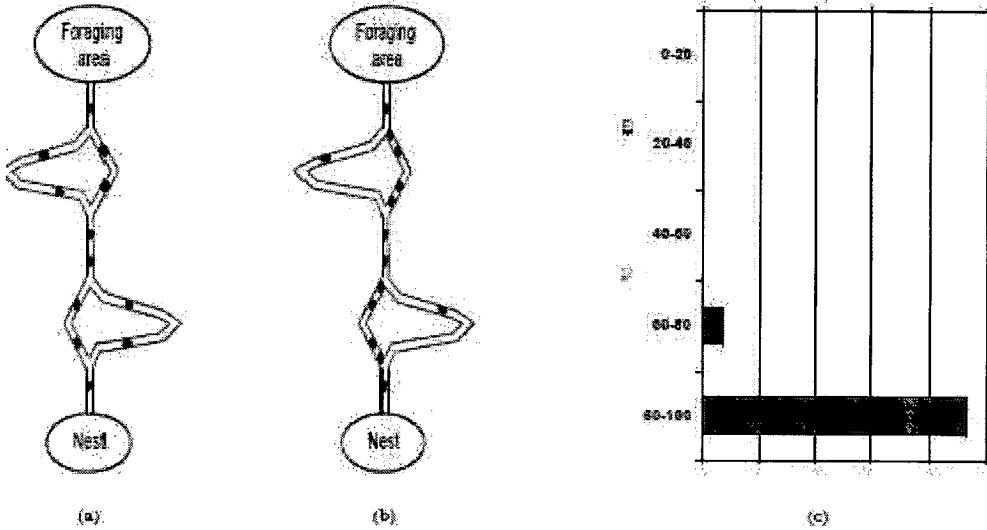


Figure 2.1: Double-bridge Experiment

2.6 Ant System (AS)

The Ant System is the first ACO algorithm where the pheromone values are updated at each iteration by all the $|m|$ ants that have built a solution in the iteration itself [17, 33]. Using the *traveling salesman problem* (TSP) as an example model, the pheromone τ_{ij} is associated with the edge joining cities i and j , and it is updated as follows:

$$\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij} + \sum_{k=1}^m \Delta\tau_{ij}^k \quad (2.8)$$

where ρ is the evaporation rate, m is the number of ants, and $\Delta\tau_{ij}^k$ is the quantity of pheromone laid on edge (i, j) by ant k and

$$\Delta\tau_{ij}^k = \begin{cases} Q/L_k, & \text{if ant } k \text{ used edge } (i, j) \text{ in its tour,} \\ 0, & \text{otherwise} \end{cases}$$

where Q is a constant used as a system parameter for defining a high quality solutions with low cost, and L_k is the length of the tour constructed by ant k [17]. In the construction of a solution, each ant selects the next city to be visited through a stochastic mechanism. When ant k is in city i and has so far constructed the partial solution s^p , the probability of going to city j is given by:

$$p_{ij}^k = \frac{\tau_{ij}^\alpha \cdot \eta_{ij}^\beta}{\sum_{c_{il} \in S^p} \tau_{il}^\alpha \cdot \eta_{il}^\beta} \forall j \in S \quad (2.9)$$

or zero otherwise. The parameters α and β control the relative importance of the pheromone versus the *heuristic information* η_{ij} , which is given by:

$$\eta_{ij} = \frac{1}{d_{ij}} \quad (2.10)$$

where d_{ij} is the distance between cities i and j .

Ant Colony System (ACS)

In ACS (e.g ACO) we introduce the *local pheromone update* mechanism in addition to the *offline pheromone update* performed at the end of the construction process [17]. The *local pheromone update* is performed by all the ants after each construction step. Each ant applies it only to the last edge traversed using the following function:

$$\tau_{ij} = (1 - \varphi) \cdot \tau_{ij} + \varphi \cdot \tau_0 \quad (2.11)$$

where $\varphi \in (0, 1]$ is the pheromone decay coefficient, and τ_0 is the initial value of the pheromone. Local pheromone update decreases the pheromone concentration on the traversed edges in order to encourage subsequent ants to choose other edges and, hence, to produce different solutions [17].

The *offline pheromone update* is applied at the end of each iteration by only one ant, which can be either the iteration-best or the best-so-far as shown below:

$$\tau_{ij} \leftarrow \begin{cases} (1 - \rho) \cdot \tau_{ij} + \rho \cdot \Delta\tau_{ij}, & \text{if } (i, j) \text{ belongs to best tour,} \\ \tau_{ij}, & \text{otherwise.} \end{cases}$$

The next section describes in more detail the *Ant System Algorithm* and briefly presents its complexity bounds.

2.7 Complexity Analysis of Ant Colony Optimization

Now that we have described the characteristics of Ant Systems (AS) we give their pseudo-code in more detail as shown below. We use the following

notation: for some state i , j is any state that can be reached from i ; η_{ij} is a *heuristic information* between i and j calculated depending on the problem in hand, and τ_{ij} is the pheromone trail value between i and j .

Algo-IV The Ant System Algorithm

1. Initialization

$\forall (i,j)$ initialize τ_{ij} and η_{ij}

2. Construction

For each ant k (currently in state i) **do**

repeat

choose the next state to move to by means of Equation 2.9

append the chosen move to the k^{th} ant's set tabu_k

until ant k has completed its solution

end for

3. Trail update

For each ant k **do**

find all possible transitions from state i to j

compute $\Delta\tau_{ij}$

update the trail values

end for

4. Terminating condition

If not(end test) go to step 2

Walter J. Gutjahr [26] analyzed the runtime complexity of two ACO algorithms: the *Graph-Based Ant System (GBAS)* and the *Ant System*. In both analysis, the results showed computation times of order $\mathcal{O}(m \log m)$ for reaching the optimum solution, where m is the size of the problem instance. The results found were based on basic test functions.

After we have describe the *SA* and *ACS* in the previous sections, the two algorithms will be applied on the exam scheduling problem. We might not find the optimal solution (NP problem) where the hard and soft constraints are all satisfied completely, but we will attempt to reach a near-optimal solution after several iterations.

Chapter 3

Related Work

During the past years, many algorithms and heuristics were used in solving the timetabling problem. The algorithms vary from simple local search algorithms to variations of genetic algorithms and graph representation problems. We list some of the recognized techniques which proved to find acceptable solutions concerning the scheduling problem:

- I. Simulated Annealing [24]
- II. Ant-Colony Optimization [41]
- III. Tabu Search [12]
- IV. Graph Coloring [36]
- V. Hybrid Heuristic Approach [30, 12]

Simulated Annealing (SA) (I) and *Ant Colony Optimization (ACO)* (II) algorithms were described in the previous chapter.

Tabu Search (TS) (III) is a heuristic method originally proposed by Glover [25] in 1986 that is used to solve various combinatorial problems. *TS* pursues a local search whenever it encounters a local optimum by allowing non-improving moves. The basic idea is to prevent cycling back to previously visited solutions by the use of memories, called tabu lists, that record the recent history of the search. This is achieved by declaring tabu (disallowing) moves that reverse the effect of recent moves.

As for the *Graph Coloring* (IV) problem, it can be described as follows. Suppose we have as an input a graph G with vertex set V and edge set E , where the ordered pair $(R,S) \in E$ if and only if an edge exists between

the vertices R and S . A k -coloring of graph G is an assignment of integers $\{1, 2, \dots, k\}$ (the colors) to the vertices of G in such a way that neighbors receive different integers. The chromatic number of G is the smallest k such that G has a k -coloring. That is, each vertex of G is assigned a color (an integer) such that adjacent vertices have different colors, and the total number of colors used (k) is minimum.

The problem of Optimizing Timetabling Solutions using *Graph Coloring* is to partition the vertices into a minimum number of sets in such a way that no two adjacent vertices are placed in the same set. Then, a different color is assigned to each set of vertices.

In the *Hybrid Approach(V)*, the idea is to combine more than one algorithm or heuristic and apply them on the same optimization problem in order to reach a better and more feasible solution. Sometimes the heuristics are combined into a new heuristic and then the problem is solved using this new heuristic. In other cases the different heuristics are used in phases, and every phase consists of applying one of these heuristics to solve a part of the optimization problem.

Duong T.A and Lam K.H presented in [20] a solution method for *examination timetabling*, consisting of two phases: a constraint programming phase to provide an initial solution, and a simulated annealing phase with *Kempe chain neighborhood*. They also refined mechanisms that helped to determine some crucial cooling schedule parameters. In [36] a method using *Graph Coloring* was developed for optimizing solutions to the *timetabling problem*. The eleven course timetabling test data-sets were introduced by Socha K. and Sampels M. [41] who applied a *Max-Min Ant System (MMAS)* with a construction graph for the problem representation. Socha K. also compared *Ant Colony System (ACS)* against a *Random Restart Local Search (RRLS)* algorithm and *Simulated Annealing (SA)* [40]. A comparison of five metaheuristics for the same eleven data-sets was presented by Rossi-Doria O. [39]; the approaches included in this study were: the *Ant Colony System (ACS)*, *Simulated Annealing (SA)*, *Random Restart Local Search (RRLS)*, *Genetic Algorithm (GA)* and *Tabu Search (TS)*. The conclusions drawn from the comparisons done in [39] are the following:

- The problem difficulty varies between problem instances, across categories, and to a lesser extent, within a category, in terms of the observed aggregated performance of the metaheuristics. In the context of the scheduling problem, an example is when a particular choice of subjects by a particular student make the timetabling much more difficult

in a particular year.

- The absolute performance of a single metaheuristic, and the relative performance of any two metaheuristics vary between instances, within and, to a lesser extent, across categories.
- The performance of a metaheuristic with respect to satisfying hard constraints and satisfying soft constraints may be very different.
- It is very difficult to design a metaheuristic that can tackle general instances.
- Promising solutions are generated when using algorithms consisting of at least two phases, one taking care of feasibility, the other taking care of minimizing the number of soft-constraint violations.

Abdullah S. and Burke E. K. [3] developed a *Variable Neighborhood Search* based on a random descent local search with Monte-Carlo acceptance criterion. Burke et al. [9] employed *Tabu Search* within a *Graph-based Hyper-Heuristic* and applied it to both course and examination timetabling benchmark data-sets in order to raise the level of generality by operating on different problem domains.

Since we will be using the SA and ACO algorithms in this research, we will highlight the most common scheduling problems that were solved using these algorithms.

Apart from the exam scheduling problem and the generation of weekly timetables, SA has been applied to many optimization problems occurring in areas such as *Very Large Scale Integration (VLSI)* design, image processing, molecular physics and chemistry, and job shop scheduling.

Laarhoven, Aarts and Lenstra [45] described an approximation algorithm for the problem of finding the minimum make-span in a job shop using SA. This problem is a form of machine scheduling where we are given a set of jobs and a set of machines and each job consists of a chain of operations that must be processed during an uninterrupted time period of a given length on a given machine. Each machine can process at most one operation at a time and a schedule is an allocation of the operations to time intervals on the machines. The problem is to find a schedule of minimum length.

Another scheduling problem solved by SA is *Grid computing* [23] which is a form of distributed computing that involves coordinating and sharing computing, application, data storage or network resources across dynamic and geographically dispersed organizations. The goal of grid tasks scheduling is

to achieve high system throughput and to match the application need with the available computing resources.

D. F. Wong presented a method for optimizing *VLSI design* [46] using SA. As described in his book, the *VLSI design problem* is divided into 2 sub-problems. the first one is the general placement problem of placing a set of circuit modules on a chip such that a certain objective function is minimized. The second one is to minimize the total chip area occupied by circuit modules.

J-M. Su and J-Y. Huang used Ant Colony Optimization to solve another scheduling problem which is the *Train Timetabling Problem* [49]. The aim is to determine a periodic timetable for a set of trains that does not violate track capacities and some other operation constraints.

ACO was also used to solve *Fuzzy Job Shop Scheduling Problems* [43] used in complex equipment manufacturing systems to validate the performance of heuristic algorithms. The idea was to move ants from one machine (nest) to another machine (food source) depending upon the job flow, thereby optimizing the sequence of jobs.

Another important problem solved using ACO is the *quadratic assignment problem (QAP)* [42]. QAP can best be described as the problem of assigning a set of facilities to a set of locations with given distances between the locations and given flows between the facilities. The goal then is to place the facilities on locations in such a way that the sum of the product between flows and distances is minimal. The QAP is considered to be one of the hardest combinatorial problems, and can be solved to optimality only for small instances. Several ACO applications dealt with the QAP, starting using the simple AS and then by means of many more advanced versions like MMAS (Min-Max Ant System).

In the following chapter we will present our solution in the context of the exam scheduling problem using the Simulated Annealing meta-heuristic.

Chapter 4

SA Applied To The Scheduling Problem

4.1 Our Approach Using SA

In this research we provide a general solution that allows to produce examination schedules that meet the various academic rules of the universities. We will apply our method to a simple instance of the scheduling problem. Therefore we consider the example schedule with the following attributes:

1. There are exactly 4 examination periods (time-slots) in each examination day.
2. We have a fixed number of rooms (equal to 3) which can be used to hold the exams.
3. There are 24 exams to be scheduled in a total examination duration of 2 days.
4. Room allocation is maximized. Any room available during the examination period should be allocated to hold an exam.

We start by building an initial solution for the schedule. All the following work was implemented using *Matlab*.

The schedule is represented by an $m \times n$ matrix denoted by *Sched*. *Sched*[i,j] holds a set S_{ij} of exams scheduled at day d_i and period p_j , where $i=1, 2, \dots, m$ and $j= 1, 2, \dots, n$. Hence the matrix *Sched* will have the following properties:

- A number of columns $n = 4$ since we have exactly 4 examination periods (time-slots) per day.
- A number of rows $m \geq 1$ depending on the number of exams to be scheduled.
- $|S_{ij}| = 3$ since we have 3 examination rooms that we wish to use at each examination period.

In our example, the 24 exams are scheduled in 2 examination days over 4 examination periods each day. This is depicted in Figure 4.1.

		<i>Periods</i>			
		3:00-10:00	11:00-1:00	2:00-4:00	5:00-8:00
Days	Day1	(E1, E2, E3)	(E7, E8, E9)	(E13, E14, E15)	(E19, E20, E21)
	Day2	(E4, E5, E6)	(E10, E11, E12)	(E16, E17, E18)	(E22, E23, E24)

Example schedule with 24 exams

Figure 4.1: Simple Examination Schedule

The exams are denoted by the Letter E concatenated to the course code. For example if we have a course code $CS111$ the exam code for this course will be $ECS111$. Exam $E9$ in the picture above is scheduled from 11:00 am to 1:00 pm of day one. As stated above, each day we have 3 exams at each period, this is because we have 3 examination rooms available and we wish to maximize their utilization by always allocating non-scheduled exams to the empty rooms.

Depending on the number of exams to be scheduled, a case frequently appears where we sometimes end up by having rooms which are not allocated to any exam at the final examination day. This is logical since the number of exams might not be a multiple of the size of the matrix $Sched$. We accommodate for this by adding virtual exams in the remaining empty cells. These exams have no conflicts whatsoever with any of the other exams. This is done to allow for the algorithm to run on *Matlab* since it is necessary to fill-in all matrix cells.

In some other cases, the examination rooms are vast halls, and might hold more than one exam at one period. To account for this change, we consider

the hall to be multiple examination rooms (the exact multiple depends on the number of seats in the Hall).

We have provided a function $ReturnConflicts(E_{i1}, E_{i2})$ that takes two exams as parameter and returns an integer equal to the number of students taking these exams (E_{i1} and E_{i2} in the case above) in common. Using the notations of Section 1.1 the function can be described as follows:

```

Function: ReturnConflicts( $e_a, e_b$ )
set counter = 0
for all  $s_j \in S$  do
    if  $se_{ja} = 1 \ \&\& \ se_{jb} = 1$  then
        counter = counter + 1
    end if
end for
return counter

```

We start by inserting the exams to be scheduled into the matrix $Sched$. The exams are first inserted randomly, and the cost of the random solution is calculated (the details of the calculations will be showed hereafter).

The cost a solution schedule is the sum of:

1. the cost of its hard constraints returned by checking for any students having exam clashing (more than one exam in the same time-slot), and
2. a *fraction* of the cost of its soft constraints. This is done by multiplying the cost of these soft constraints by a decimal $\varepsilon \in]0,1[$

This is illustrated below:

$$Total\ Cost = cost\ of\ hard\ constraints + \varepsilon \cdot cost\ of\ soft\ constraints \quad (4.1)$$

Still remains to explain how to calculate the cost of the hard and soft constraints of a solution schedule. The cost of the hard constraints is calculated over 2 steps:

1. Run the function $ReturnConflicts()$ for any two exams occurring in the *same time-slot of the same day*

2. Take the *Sum* of the returned values.

As for the soft constraints, we need to make sure that we space the exams fairly and evenly across the whole group of students thus, we need to check that a student has no more than one exam in two consecutive time-slots of the same day. The same procedure will be used to calculate the cost of the soft constraints with only one modification. This time the function *Return-Conflicts()* is run on all exams pairs occurring in *consecutive time-slots of the same day*, and their costs are added together.

To control the relative importance of the hard constraints over the soft constraints, we added the cost of the hard constraints to a *fraction* of the total cost of the solution's soft constraints by multiplying it by a decimal $\varepsilon \in]0,1[$.

Once we have defined how to calculate the cost of the solution in hand, we can now use the SA algorithm to iterate over neighbor solutions in the aim of reaching better cost solutions. This is describe in the following sections.

Choosing a starting temperature

We used the *Tuning for initial temperature method* as described in Subsection 2.1.1, whereby we start at a very high temperature and then cool it rapidly until about 60% of worst solutions are being accepted. we then use this temperature as T_0 .

Temperature Decrement

In this research we will use an alternative method from those described in Subsection 4.1 to decrement the temperature. This method was first suggested by Lundy [32] in 1986 and it consists of doing only one iteration at each temperature, but to decrease the temperature very slowly. The formula that illustrates this method is the following:

$$T_{i+1} = \frac{T_i}{1 + \beta \cdot T_i} \quad (4.2)$$

where β is a suitably small value and T_i is the temperature at iteration i .

In our test case we will take β to be equal to 0.001. Another solution to the temperature decrement is to dynamically change the number of iterations as the algorithm progresses [29].

Final Temperature

As a final temperature, we chose a suitably low temperature where the system gets frozen at. We used similar results as those found in [20] where many experiments using SA were run to solve the university timetabling problem. Each experiment was done using different final temperatures, namely [0.5, 0.05, 0.005, 0.0005, 0.00005] and a fixed value of $T_f = 0.005$ was chosen. This final temperature T_f returned the best solution cost. In our solution, we used the same T_f even though any temperature $\in [0.005, \dots, 1]$ would have given very close results.

But our stopping criterion does not only depend on T_f . A check was made on consecutive solutions where no moves appeared to be improving the cost afterwards and, whenever we receive the same cost over more than $T_0/4$ iterations we stop since we probably have reached a best-case solution. Of course we would also stop whenever we reach a schedule with $cost = 0$ since it would be an optimal solution.

Neighborhood structure

In SA, a neighbor solution s' of s is usually any acceptable solution that can be reached from s . In the context of the scheduling problem, a neighbor s' of the current schedule s is another schedule where the exams have been distributed differently starting from s . This works in practice, but we have improved it by constraining some schedule configurations which return very high and impractical costs. This was done by adding these configurations to a black-list in such a way that whenever such configurations appear during the running time of the algorithm, they are skipped and a search for new neighbors with different configuration is launched. Although this is naturally controlled in the SA algorithm by the acceptance probability of neighbor solutions, constraining such high cost solutions can save many unsatisfactory iterations and therefore a big amount of computer processing time.

4.2 Empirical Results (SA)

We have run the SA algorithm on the exam scheduling problem using the structures and values defined in the sections above, and plotted the results on *Matlab*. Part of the plot, where the temperature goes down from 100 to 0 (without using the black-list constraint on neighbor configurations) is shown in the Figure 4.2 together with the respective solutions costs at each temperature.

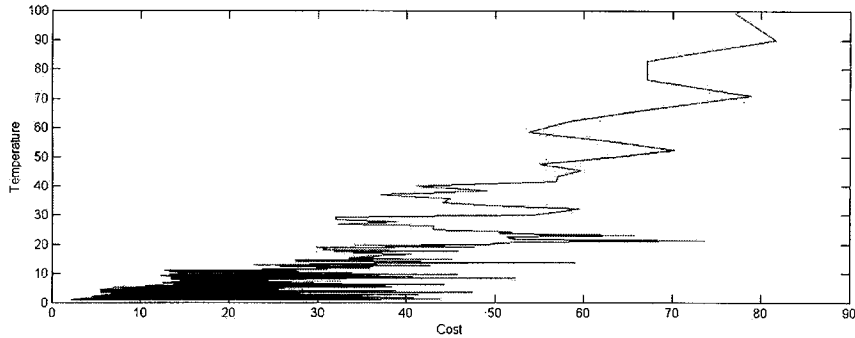


Figure 4.2: Variation in cost values with respect to temperature decrement

We can see that the cost starts with high value (equal to 78) when the temperature is near 100, and it drops gradually with the temperature until it reaches a value equal to 3. At some points of the plot, the cost increases even though the temperatures are decreasing. These are exactly the uphill steps that appear in SA where worse moves are allowed to be taken to escape from local minimum. One more thing to notice in the plot is that the probability of accepting a worse move is decreased when the temperature decreases just as expected from Equation 2.1.

Another plot is shown in Figure 4.3, where the SA algorithm is started from the same initial configuration and run within the same range of temperatures, but now using the improved version that consists of constraining unfeasible neighbor configurations.

We can see that the difference between the costs in adjacent temperatures is narrower than those in Figure 4.2. Even though the final cost reached is the same, the cost function with respect to temperature drops more strictly now. This resulted from the fact that many non-useful iterations were avoided due to the restrictions of unfeasible schedule configurations.

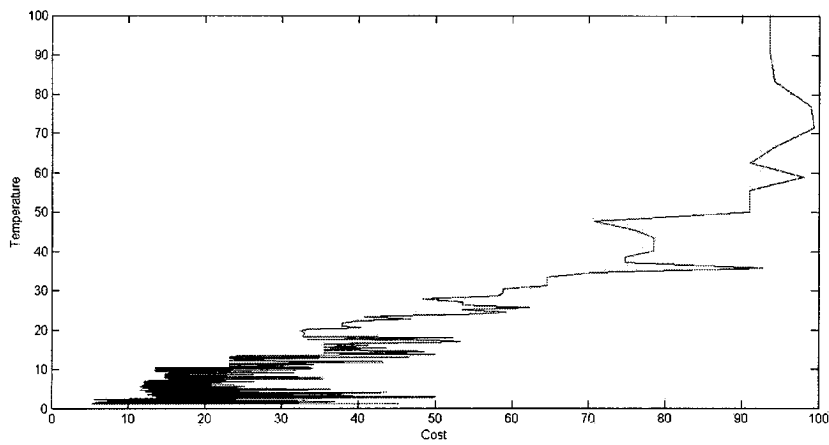


Figure 4.3: Variation in cost values with respect to temperature decrement after constraining unfeasible neighbor configurations

Chapter 5

ACO Applied To The Scheduling Problem

5.1 Our Approach Using ACO

We will use the same definition of the scheduling problem as described in the Introduction. Furthermore, we consider the same example problem as in Section 4.1 whereby *Sched* denotes the schedule we are building and the function *ReturnConflicts*(E_{i1}, E_{i2}) takes two exams as parameter and returns an integer equal to the number of students taking these exams E_{i1} and E_{i2} in common (conflicts between the 2 exams). We also recap that we are trying to schedule 24 exams in a period of 2 examination days.

To be able to use the *ACO* algorithm on this problem, we create a 24×24 matrix to hold the pheromone values between the exams and call it *PhMatrix*.

The pheromone values in the *timetabling* problem will be used differently than what has been done in *TSP* since the relative position of an exam does not only depend on its direct predecessor or its direct successor in the schedule, but also on all the exams that are to be scheduled within the same time-slot of the same day (that is in the same set S_{ij}) and therefore the costs are calculated differently in these 2 problems. The difference is highlighted in the example below:

To calculate the cost of scheduling 3 exams in the same set S_{ij} we must check for the all the conflicts between every other exam in this set. On the other hand, to calculate the cost of going from a city i to another city j in a *TSP*, we only need to check for the distance between the two cities without considering the distances to the cities we have previously visited.

In the context of the scheduling problem, the ants will therefore decide

which exams are feasible to be placed in the same set S_{ij} of the schedule. We know that we can have as many exams in each S_{ij} as there are available rooms.

PhMatrix is first initialized so that the values $\tau_{ij} : i, j \in \{1, \dots, n\}$ are all equal to 1. The attractiveness η_{ij} is defined as follows:

$$\eta_{ij} = \frac{1}{\text{ReturnConflicts}(E_i, E_j)} \quad (5.1)$$

where E_i, E_j refer to exams i and j respectively.

At each iteration the ants will start from a new source (exam) and build their solution (schedule). The ants choose an exam as a source; they move to the next exam which has a highest probability according to equation 2.9. It is clear that during the first iteration the ants will choose the next exam having the minimum number of conflicts (highest η_{ij}) with the previous one, since the pheromone values are all equal.

Once the next exam is chosen by ant k , the previous one in the schedule is put in the *tabu list* of ant k , that is a list containing all moves which are infeasible for ant k . This is done to ensure that the same exam is not scheduled twice in the timetable. Ant k continues its colony, and chooses the next exam in the same way.

But, as we showed in the example above, even if this works for adjacent exams in the schedule, this might lead to conflicts with other exams scheduled in the same time-slot. So the point to make here is that, even when η_{ij} is optimal between 2 consecutive exams, it leads in some cases to high costs returned from conflicts with other exams scheduled within the same set S_{ij} . This might be accounted for by elevating the defined parameter α in equation 2.9, but this does not solve the problem completely.

Therefore, a global pheromone evaluation rule is proposed where an ant k at exam i that has to decide about the next exam j of the permutation, makes the selection probability:

- Considering every exam j not in the tabu list of k
- Depending on the sum of all pheromone values between, the exams already scheduled in the same set as i (denoted by l), and the candidate exam j which is: $\sum_{l=1}^i \tau_{lj}$.

So we end up by the following equation:

$$p_{ij} = \frac{(\sum_{l=1}^i \tau_{lj})^\alpha \cdot \eta_{ij}^\beta}{\sum_{z \in S} (\sum_{l=1}^i \tau_{lz})^\alpha \cdot \eta_{iz}^\beta} \quad \forall j \in S \quad (5.2)$$

This makes sure that we have considered the probabilities of moving to the next exam, for all the exams already scheduled in the same time-slot of the current day, before choosing it.

The steps above are repeated until every ant completes its solution asynchronously from other ants. During this construction phase, the ants evaluate their solution and modify the pheromone trail values on the components of this solution. This pheromone information will direct the search of the future ants. We next show how the pheromone update is done in our solution.

5.2 Pheromone Update

After scheduling a set of exams (one cell) in the timetable *Sched*, we do a *local pheromone update* whereby we update the pheromone values between all pairs of scheduled exams according to Equation 2.11.

On the other hand, to avoid unfeasible distributions (with conflicting exams that lead to dead-end configuration) from being placed in the same set S_{ij} , we have decided to induce negative pheromone values between the exams leading to such distributions in such a way that, if exam i and j lead to future conflicting configurations in the timetable, they will be assigned a negative pheromone value τ_{ij} even if i and j have no conflicts with each other.

The negative pheromone update equation is the inverse of Equation 2.11 where the addition is replaced by a subtraction. The equation is the following:

$$\tau_{ij} = (1 - \varphi) \cdot \tau_{ij} - \varphi \cdot \tau_{old} \quad (5.3)$$

where $\varphi \in (0, 1]$ is the pheromone decay coefficient, and τ_{old} is the old value of the pheromone. This negative pheromone update can be induced directly after the pheromone initialization between the exams resulting in conflicting configurations, therefore before the ants start building their solution. This will ensure that for ant k the probability of choosing these exams in the same set is very low.

5.3 Empirical Results (ACO)

The results of running the *ACO* algorithm on the example described previously were plotted on *Matlab*. Figure 5.1 below shows the variation in

the cost of the schedule at several iterations. At each iteration we start from a different nest (source) and build a complete solution.

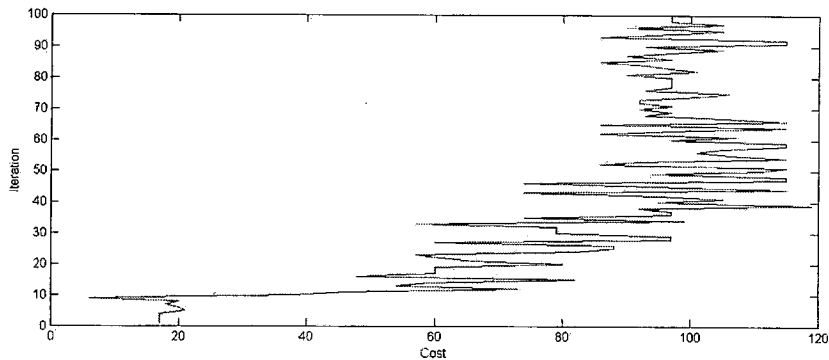


Figure 5.1: ACO - Cost values at subsequent iterations (Hard Schedule)

We can see that the cost drops significantly from a value around *120* to almost *5* after *100* iterations. Plot 5.1 corresponds to solution for a scheduling problem instance with a huge number of conflicts between students. We have also run the ACO algorithm on a different instance having a lower number (still a considerable number) of conflicts between exams and the results are plotted in Figure 5.2. The initial solution has a cost equal to *20* and it drops to *0* after only *45* iterations, therefore an optimal solution was found in this case.

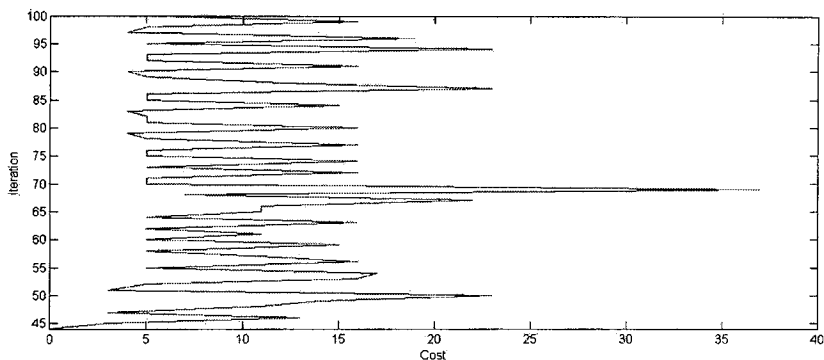


Figure 5.2: ACO - Cost values at subsequent iterations (Loose Schedule)

5.4 Performance Analysis: ACO vs. SA

Each of the *Simulated Annealing* and *Ant Colony Optimization* algorithms was tested by performing 15 trials in the aim of building complete examination schedules, starting from different initial configurations and using different numbers of exams.

The first 5 trials consisted of generating schedules for 24 exams (timetable size = 24) in the minimum possible timescale. In the next 5 trials we generated timetables of size equal to 32, while in the last 5 trials we generated timetables of size equal to 38.

Note that the schedules chosen in these trials have a percentage of conflicting exams that varies between 55% and 65% (so more than half of the exams in these schedules have conflicts with each other). Also note that *ACO* was run using 50 ants, and each ant chose a random exam as its starting point (source).

The results with the lowest cost were recorded at each trial, together with their corresponding CPU running time. The standard deviation (σ) of every 5 trials was also calculated. All the trials were run on a Core2 Duo PC with 2.0 GHz CPU and 2 GB of RAM. The results are shown in the table 5.1 below.

We have made the following observations:

1. The running times of ACO are better than those of SA in all 15 trials. SA annealing takes more time to discover and evaluate the neighbor solutions at each iteration (temperature). ACO uses information from prior iterations to guide subsequent colonies to new states (neighbors) which reduces the processing time needed to calculate the cost of such moves.
2. ACO found the least cost solution in all 3 timetable sizes, even though it sometimes lead to high cost solutions compared to those found in SA. The standard deviations of the timetables' costs produced using SA are lower than those found in the case of ACO which means that SA provide tight results where the costs of the solutions are close to each other, while ACO gives broad results where the difference between the costs can be high.
3. If we choose to use more ants in ACO, the running time will increase and we will not get any better results, so we fixed the number of ants

Table 5.1: Performance of ACO and SA algorithms: Empirical Results

SA -	Running Time (sec)	Cost	ACO-	Running Time (sec)	Cost
Trial#1	1.5444	3.55	-	0.2028	0.0100
Trial#2	1.5000	6.80	-	0.2184	1.0500
Trial#3	1.2948	3.70	-	0.1716	10.900
Trial#4	1.5132	3.80	-	0.2184	2.9500
Trial#5	1.5288	3.85	-	0.0936	5.9500
-	-	$\sigma = 1.37$	-	-	$\sigma = 4.38$
SA -	Running Time (sec)	Cost	ACO-	Running Time (sec)	Cost
Trial#6	2.1300	3.97	-	0.3400	11.8000
Trial#7	1.9909	5.81	-	0.9909	4.3500
Trial#8	1.9001	7.20	-	0.4411	7.5000
Trial#9	2.0152	3.10	-	0.2214	1.3300
Trial#10	2.1001	4.20	-	0.1999	3.4999
-	-	$\sigma = 1.63$	-	-	$\sigma = 4.06$
SA -	Running Time (sec)	Cost	ACO-	Running Time (sec)	Cost
Trial#11	2.5120	4.36	-	1.0933	13.950
Trial#12	4.6000	3.37	-	0.9922	4.0011
Trial#13	3.0011	5.11	-	1.0056	2.3500
Trial#14	3.3111	3.91	-	0.8851	5.9500
Trial#15	2.7222	3.56	-	0.9111	2.7600
-	-	$\sigma = 0.69$	-	-	$\sigma = 4.76$

to 50. On the other hand, if we reduce the number of ants we will not reach such low cost solutions even though we will achieve better running times. The same was noticed in SA: we can use a lower initial temperature or even decrease the number of neighbors visited at each temperature, and thus do less iterations (better running times), but we would have not been able to achieve the above costs.

4. When the number of *conflicting exams* chosen is too high in such a way that more than 70% of the exams have conflicts with each other (not shown in table 5.1), the ACO algorithm outperforms the SA algorithm. We had to highly increase the number of iterations done at each temperature (using the static strategy of temperature decrement) to allow for the SA algorithm to explore enough neighbors so that it is able to find a better move (neighbor).

5. When the number of *conflicting exams* chosen is very loose in such a way that less than 20% of the exams have conflicts with each other (also not shown in table 5.1), both approaches lead to near-optimal solutions. Although ACO converged to a near-optimal solution using 50 ants, its running time was higher than that of SA, since SA reached the stopping criteria in only very few iterations.

Chapter 6

Conclusion

6.1 Summary of the Main Results

We have used two algorithms namely SA (simulated annealing) and ACO (ant-colony algorithm) to solve the scheduling problem. We first introduced the problem and provided its mathematical formulation in Chapter 1 and then we described the *SA* and *ACO* algorithms and illustrated the way they are used to solve combinatorial optimization problems. We presented our approach to solving the scheduling problem using these algorithms in Chapter 4 and 5. All the results were implemented using *Matlab*, and a comparison between the performance and running times of *SA* and *ACO* in producing different examination schedules over several trials was depicted in Table 5.1. The solution we provided was based on an exam scheduling problem model that could be implemented in Notre-Dame University. It is not difficult to generalize our solution to solve many different scheduling problems with some minor modifications regarding the variables related to the problem in hand and the resources available.

6.2 Future Work

Our future work consists of parallelizing these algorithms in order to improve their running times and also on using a hybrid *Ant Colony - Simulated Annealing* approach to improve the cost of our solution. We intend to parallelize the two algorithms at the level of data whereby we work on solving sub-problems and then combine them into a bigger low cost problem. We also intend to achieve parallelism at the level of ants when using the ACO algorithm in such a way that ants can work in parallel to find their solution.

Bibliography

- [1] Karen I. Aardal, Stan P. M. Van Hoesel, Arie M. C. A. Koster, Carlo Mannino, and Antonio Sassano. Models and solution techniques for frequency assignment problems. pages 261–317, 2001.
- [2] Emile H. L. Aarts and Peter J. M. Van Laarhoven. *Simulated Annealing: Theory and Applications*. Kluwer Academic Publishers, 1987.
- [3] Salwani Abdullah, Edmund K. Burke, and Barry Mccollum. An investigation of variable neighbourhood search for university course timetabling. In *The 2nd Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA)*, pages 413–427, 2005.
- [4] Masri Binti Ayob and Ghaith Jaradat. Hybrid ant colony systems for course timetabling problems. In *Proceedings of the 2nd conference on data mining and optimization, Universiti Kebangsaan Malaysia*, pages 120–126. IEEE, 2009.
- [5] Imed Bouazizi. Ara - the ant colony based routing algorithm for manets. In *Proceedings of the 2002 International Conference on Parallel Processing Workshops*, ICPPW 02 series, page 79, Washington, DC, USA, 2002. IEEE.
- [6] P Brucker and S Knust. In complexity results for scheduling problems. In *Robust and Online Large-Scale Optimization, volume 5868 of Lecture*, Last update: 29.06.2009.
- [7] Edmund Burke, Kirk Jackson, Jeff Kingston, and Rupert Weare. Automated university timetabling: the state of the art. *The Computer Journal*, 40:565–571, 1997.
- [8] Edmund K. Burke, Dave Elliman, Peter H. Ford, and Rupert F. Weare. Examination timetabling in british universities: a survey. In *Selected papers from the First International Conference on Practice and Theory of Automated Timetabling*, pages 76–90, London, UK, UK, 1996. Springer-Verlag.

- [9] Edmund K Burke, Barry Mccollum, Amnon Meisels, Sanja Petrovic, and Rong Qu. A graph-based hyper-heuristic for educational timetabling problems. *European Journal of Operational Research*, 176:177–192, 2007.
- [10] Oscar Castillo, Patricia Melin, Janusz Kacprzyk, and Witold Pedrycz, editors. *Soft computing for Hybrid Intelligent Systems*, volume 154 of *Studies in Computational Intelligence*. Springer, 2008.
- [11] C. Y. Cheong, K. C. Tan, and B. Veeravalli. A multi-objective evolutionary algorithm for examination timetabling. *J. of Scheduling*, 12(2):121–146, April 2009.
- [12] M. Chiarandini, M. Birattari, K. Socha, and O. Rossi-Doria. An effective hybrid algorithm for university course timetabling. *Journal of Scheduling*, 9(5):403–432, October 2006.
- [13] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.
- [14] D. de Werra. An introduction to timetabling. *European Journal of Operational research*, (19), 1985.
- [15] M. Dorigo. *Optimization, learning and natural algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992.
- [16] Marc Dorigo and Socha Krzysztof. An introduction to ant colony optimization. *IRIDIA Technical Report Series*, 2006.
- [17] Marco Dorigo, Mauro Birattari, and Thomas Sttzle. Ant colony optimization - artificial ants as a computational intelligence technique. *IEEE COMPUT. INTELL. MAG*, 1:28–39, 2006.
- [18] Marco Dorigo and Christian Blum. Ant colony optimization theory: a survey. *Theor. Comput. Sci.*, 344(2-3):243–278, November 2005.
- [19] Marco Dorigo and Thomas Sttzle. The ant colony optimization meta-heuristic: algorithms, applications, and advances. In *Handbook of Meta-heuristics*, pages 251–285. Kluwer Academic Publishers, 2002.
- [20] Tuan-Anh Duong and Kim-Hoa Lam. Combining constraint programming and simulated annealing on university exam timetabling, 2004.
- [21] R. W. Eglese. Simulated annealing: a tool for operational research. *European Journal of Operational Research*, (46), 1990.
- [22] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing*, 5(4):691–703, 1976.

- [23] Stefka Fidanova. Simulated annealing for grid scheduling problem. In *Proceedings of the IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing, JVA '06*, pages 41–45, Washington, DC, USA, 2006. IEEE Computer Society.
- [24] Juan Frausto-Solís, Federico Alonso-Pecina, and Jaime Mora-Vargas. An efficient simulated annealing algorithm for feasible solutions of course timetabling. In *Proceedings of the 7th Mexican International Conference on Artificial Intelligence: Advances in Artificial Intelligence, MICAI '08*, pages 675–685, Berlin, Heidelberg, 2008. Springer-Verlag.
- [25] M. Gendreau. An Introduction to Tabu Search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, chapter 2, pages 37–54. Kluwer Academic Publishers, 2003.
- [26] Walter J. Gutjahr. First steps to the runtime complexity analysis of ant colony optimization. *Comput. Oper. Res.*, 35(9):2711–2727, September 2008.
- [27] Juraj Hromkovic. *Algorithmics for hard Problems: introduction to combinatorial optimization, randomization, Approximation, and heuristics*. Springer-Verlag, Berlin, Heidelberg, 2010.
- [28] Richard M. Karp. Reducibility among combinatorial problems. In Michael Jnger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008*, pages 219–241. Springer Berlin Heidelberg, 2010.
- [29] Graham Kendall. Artificial intelligence methods: Simulated annealing - introduction, 2002. course run at the The University of Nottingham within the School of Computer Science and IT.
- [30] Philipp Kostuch. The university course timetabling problem with a three-phase approach. In *Proceedings of the 5th international conference on Practice and Theory of Automated Timetabling, PATAT'04*, pages 109–125, Berlin, Heidelberg, 2005. Springer-Verlag.
- [31] Ketan Kotecha, Gopi Sanghani, and Nilesh Gambhava. Genetic Algorithm for Airline Crew Scheduling Problem Using Cost-Based Uniform Crossover. pages 84–91. 2004.
- [32] M Lundy and A Mees. Convergence of an annealing algorithm. *Math. Program.*, 34(1):111–124, January 1986.
- [33] Vittorio Maniezzo, Luca Maria Gambardella, and Fabio De Luigi. Ant colony optimization, April 09 2004.

- [34] Peter Merz and Bernd Freisleben. A comparison of memetic algorithms, tabu search, and ant colonies for the quadratic assignment problem. In *Proc. Congress on Evolutionary Computation, IEEE*, pages 2063–2070. Press, 1999.
- [35] N Metropolis, A Rosenbluth, M Rosenbluth, A Teller, and E Teller. Equations of state calculations by fast computing machines. *The Journal of Chemical Physics*, (21):1087–1091, 1953.
- [36] Sara Miner, Saleh Elmohamed, and Hon W. Yau. Optimizing timetabling solutions using graph coloring. In *NPAC REU program*, 1995.
- [37] Carter M.W., Laporte G., Chinneck J.W., and GERAD. *A general examination scheduling system*. Les Cahiers du GERAD. 1992.
- [38] Eric Poupert and Yves Deville. Simulated annealing with estimated temperature. *AI Commun.*, 13(1):19–26, October 2000.
- [39] Olivia Rossi-doria, Michael Sampels, Mauro Birattari, Marco Chiar, Marco Dorigo, Luca M. Gambardella, Joshua Knowles, Max Manfrin, Monaldo Mastrolilli, Ben Paechter, Luis Paquete, and Thomas Stutzle. A comparison of the performance of different metaheuristics on the timetabling problem. In *In: Proceedings of the 4th International Conference on Practice and Theory of Automated Timetabling (PATAT 2002)*, pages 329–351. Springer, 2003.
- [40] Krzysztof Socha, Joshua Knowles, and Michael Sampels. A max-min ant system for the university course timetabling problem. In *Proceedings of the Third International Workshop on Ant Algorithms, ANTS '02*, pages 1–13, London, UK, UK, 2002. Springer-Verlag.
- [41] Krzysztof Socha, Michael Sampels, and Max Manfrin. Ant algorithms for the university course timetabling problem with regard to the state-of-the-art. In *In Proc. Third European Workshop on Evolutionary Computation in Combinatorial Optimization (EvoCOP 2003)*, pages 334–345. Springer Verlag, 2003.
- [42] Thomas Stutzle and Marco Dorigo. Aco algorithms for the quadratic assignment problem. In *New Ideas in Optimization*, pages 33–50. McGraw-Hill.
- [43] P. Surekha, P.R.A. Mohanarajan, and S. Sumathi. Ant colony optimization for solving combinatorial fuzzy job shop scheduling problems. In *Communication and Computational Intelligence (INCOCCI)*, pages 295–300, dec. 2010.

- [44] El-Ghazali Talbi. *Metaheuristics: from design to implementation*. Wiley Publishing, 2009.
- [45] Peter J. M. Van Laarhoven, Emile H. L. Aarts, and Jan Karel Lenstra. Job shop scheduling by simulated annealing. *OR*, 40(1):113–125, January 1992.
- [46] D. F. Wong, H. W. Leong, and C. L. Liu. *Simulated annealing for VLSI design*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.
- [47] Anthony Wren. Scheduling, timetabling and rostering - a special relationship? In *Selected papers from the First International Conference on Practice and Theory of Automated Timetabling*, pages 46–75, London, UK, 1996. Springer-Verlag.
- [48] Qinghong Wu, Zongmin Ma, and Ying Zhang. Current status of ant colony optimization algorithm applications. In *Proceedings of the 2010 International Conference on Web Information Systems and Mining - Volume 02*, WISM '10, pages 305–308, Washington, DC, USA, 2010. IEEE Computer Society.
- [49] Jen yu Huang. Using ant colony optimization to solve train timetabling problem of mass rapid transit. In *Journal of Computer Information Systems*, 2006.