Notes on Ant Colony Optimisation in Erlang

Kenneth MacKenzie

May–June 2014

1 Introduction

Here are some notes on Ant Colony Optimisation for the SMTWTP in Erlang.

2 The Single Machine Total Weighted Tardiness Problem

We're going to look at the Single Machine Total Weighted Tardiness Problem (SMTWTP), which appears to have originated in [20]. In the SMTWTP, we're given an ordered sequence of *jobs*

$$J_1, \ldots, J_N \tag{1}$$

which are to be executed in order on a single machine without pausing between jobs. Each job J_j has

- A *processing time* (or *duration*) p_j
- A *weight* w_j
- A *deadline* (or *due date*) d_j.

We define the *tardiness* of J_j to be

$$T(j) = max\{0, C_j - d_j\},$$

where $C_j = \sum_{i=0}^{j} p_j$ is the completion time of job j (so $T(j)$ will be zero if job j completes before its deadline). The *total weighted tardiness* of the sequence $J_1, \ldots, J_N$ is

$$T = w_1T_1 + \cdots + w_NT_N. \tag{2}$$

Our goal is to arrange the sequence of jobs in such a way as to minimise the total weighted tardiness: essentially, we get penalised when jobs are finished late, and we want to do the jobs in an order which minimises the penalty. This problem is known to be NP-hard [11, 19, 18]. Typically, interesting problems are ones in which it's not possible to schedule all jobs within their deadlines.

2.1 Example data for the SMTWTP.

Many papers dealing with the SMTWTP make use of the ORLIB datasets originating in [26][1]. There are 125 instances each for problems of size 40, 50 and 100; for sizes 40 and 50 the best solutions are known, whereas for size 100 solutions which are conjectured to be the best ones are given. I'm fairly certain that these solutions must have been shown to be optimal by now, but I haven't found an explicit reference for this yet.

These instances are perhaps rather easy for current machines and techniques, and new instances are described in [14][2]; there are 25 "hard" instances of size 100, and 25 instances of size 1000. The new instances were generated using the technique described in [26] (and also in [21, 14], which may be easier to access). This technique is fairly simple, so we have produced an Erlang implementation which can generate instances of arbitrary size for use in scaling experiments.

3 Ant Colony Optimisation

3.1 Overview

Ant Colony Optimisation (ACO) is a "metaheuristic" which has proved to be successful in a number of difficult combinatorial optimisation problems, including the Travelling Salesman, Vehicle Routing, and Quadratic Assignment problems. A detailed description of the method and its applications can be found in the book [9][3]; a more recent overview can be found in [10]. There's a vast amount of research on this subject: an online bibliography at `http://www.hant.li.univ-tours.fr/artantbib/artantbib.php` currently has 1089 entries.

The ACO method is inspired by the behaviour of real ant colonies. Ants leave their colonies and go foraging for food. The paths followed by ants are initially random, but when an ant finds some food it will return to its home, laying down a trail of chemicals called *pheromones* which are attractive to other ants. Other ants will then tend to follow the path to the food source. There will still be random fluctuations in the paths followed by individual ants, and some of these may be shorter than the original path. Pheromones evaporate over time, which means that longer paths will become less attractive while shorter ones become more attractive. This behaviour means that ants can very quickly converge upon an efficient path to the food source.

This phenomenon has inspired the ACO methodology for difficult optimisation problems. The basic idea is that a (typically very large) search space is explored by a number of artificial ants, each of which makes a number of random choices to construct its own solution. The ants may also make use of heuristic information tailored to the specific

[1]These datasets can be downloaded from `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/wtinfo.html`.

[2]These datasets can be downloaded from `http://logistik.hsu-hh.de/SMTWTP`. The download also contains solutions generated by Geiger [15] which he says have been shown to be optimal by Shunji Tanaka, of Department of Electrical Engineering at Kyoto University: see `http://turbine.kuee.kyoto-u.ac.jp/~tanaka/SiPS/`.

[3]This can be downloaded from the internet.

problem. Individual solutions are compared, and information is saved in a structure called the *pheromone matrix* which records which records the relative benefits of the various choices which were made. This information is then used to guide a new generation of ants which construct new and hopefully better solutions. After each iteration, successful choices are used to reinforce the pheromone matrix, whereas pheromones corresponding to poorer choices are allowed to evaporate. The process finishes when some termination criterion is met: for example, when a specified number of iterations have been carried out, or when the best solution has failed to improve over some number of iterations.

Detailed information about the biological and computational aspects of the ACO paradigm can be found in Dorigo and Stützle's book [9]. We will outline the specifics of the method in the case of the SMTWTP in the next subsection. Note, though, that ACO is not regarded as the best technique for the SMTWTP problem. Various papers (see [2, 13], for example) suggest that the Iterated Dynasearch method of [5] is superior; see also [16] and [3].

3.2 Application to the SMTWTP.

A number of strategies have been proposed for applying the ACO method to the Single Machine Total Weighted Tardiness Problem. Our initial implementation is based on [1],[7], and [21], which give sequential ACO algorithms for solving the SMTWTP.

We have a collection of N jobs which have to be scheduled in some order. The pheromone matrix is a $N \times N$ array τ of floats, with τ_{ij} representing the desirability of placing job j in the ith position of a schedule.

The algorithm involves several constants:

- $q_0 \in [0, 1]$ determines the amount of randomness in the ants' choices.
- $\alpha, \beta \in \mathbb{R}$ determine the relative influence of the pheromone matrix τ and heuristic information η (see below).
- $\rho \in [0, 1]$ determines the pheromone evaporation rate.

The algorithm performs a loop in which a number of ants each construct new solutions based on the contents of the pheromone matrix, and also on heuristic information given by a matrix η (different or each ant); however, the entries η_{ij} are only used once each, and are calculated as the ants are constructing their solutions: it is never necessary to have the entire matrix η in memory.

Ant behaviour. In detail, each ant constructs a new solution iteratively, starting with an empty schedule and adding new jobs one at a time. Suppose we are at the stage where we are adding a new job at position i in the schedule. The ant chooses a new job in one of two ways:

- With probability q_0, the ant deterministically schedules the job j which maximises $\tau_{ij}^{\alpha}\eta_{ij}^{\beta}$. Various choices of heuristic information η have been proposed; here, we have used the *Modified Due Date* (MDD) [1] given by

$$\eta_{ij} = 1/max\{T + p_j, d_j\}$$

where T is the total processing time used by the current (partially–constructed) schedule. This rule favours jobs whose deadline is close to the current time, or whose deadline may already have passed. Other heuristics commonly used for the SMTWTP are *Earliest Due Date* (EDD) [12] and *Apparent Urgency* (AU) [25].

- With probability $1 - q_0$, a job j is chosen randomly from the set U of currently-unscheduled jobs. The choice of j is determined by the probability distribution given by

$$P(j) = \frac{\tau_{ij}^{\alpha}\eta_{ij}^{\beta}}{\sum_{k \in U} \tau_{ik}^{\alpha}\eta_{ik}^{\beta}}$$

In the sequential version of the algorithm, an ant can perform a *local pheromone update* every time it adds a new job to its schedule. This involves weakening the pheromone information for the job it has just scheduled, thus encouraging other ants to explore different parts of the solution space. In our concurrent implementation, several ants are working simultaneously, and we have omitted the local update stage since it would involve multiple concurrent write accesses to the pheromone matrix, leading to a bottleneck.

Global Pheromone Update. At the start of the algorithm, the elements of the pheromone matrix are all set to the value

$$\tau_0 = \frac{1}{AT_0},$$

where A is the number of ants (as usual), and T_0 is the total weighted tardiness of the schedule obtained by ordering the jobs in order of increasing deadline (this is called the *Earliest Due Date* schedule).

After each ant has finished, their solutions are examined to select the best one (based on lowest total tardiness: see Equation 2). The pheromone matrix is then updated in two stages:

- The entire matrix τ is multiplied by $1 - \rho$ in order to evaporate pheromones for unproductive paths.

- The path leading to the best solution S is reinforced. For every pair (i, j) with job j at position i in S, we replace τ_{ij} by

$$\tau_{ij} + \rho/T,$$

  where T is the total weighted tardiness of the current best solution.

There are many alternative strategies in the literature. For example, all ants may be allowed to contribute to the pheromone update (possibly weighted according to the quality of their solution); different evaporation strategies may be used; a different reinforcement factor may be used; the entries of τ may be constrained to lie within some specified maximum and minimum values τ_{min} and τ_{max}. See [9] (especially §3.3.3) for details and references.

Parallelisation. To a large extent the ACO algorithm is naturally parallel: we have a number of ants constructing solutions to a problem independently, and if we have multiple processors available then is would seem sensible to have ants acting in parallel on separate processors rather than constructing solutions one after the other. There is some literature on this (see [4, 22, 23, 28, 27] for example), but perhaps not as much as I would have expected.

Local Search. The basic ACO algorithm generally gets close to a local minimum, but may not find it precisely. The quality of solutions can be improved by performing a *local search* [6, 8] just before the global update stage. This involves looking at solutions in some neighbourhood of the current best solution, and moving to the one which improves the solution most. This process is repeated until no improvement is found. Various neighbourhood structures have been considered: for example, one may consider schedules which differ by a single interchange of jobs, or one where a single job is removed and inserted at a different position. Currently, our Erlang implementation includes an option for a naive version of local search (techniques such as dynamic programming can be used to improve the efficiency of the search considerably, but we haven't yet implemented this). However, we have turned this off for the experiments reported below because a considerable amount of computation is required and number of iterations is unpredictable: this leads to variations in execution time which obscure the influence of the factors which we're really interested in, such as the number of parallel ants.

4 Implementation in Erlang (Single Machine)

Our initial implementation runs on a single Erlang node, preferably running on an SMP machine. We represent the pheromone matrix as an ETS (or DETS) table which stores each row of the matrix as an N-tuple of real numbers. This is accessible to all processes running in the Erlang VM.

The program takes three arguments. The first argument is the name of a file containing input data for the SMTWTP problem: this consists of an integer N and 3 sets of N integers, representing the processing times, the weights, and the deadlines for each job. The second argument is the number A of ants which should be used, and the third argument is the number K of iterations of the main "loop" of the program (ie, the number of generations of ants).

After reading its inputs, the program creates and initialises the table representing the pheromone matrix, then spawns M ant processes. The program then enters a loop in which the following happens:

- The ants construct new solutions (as described above) in parallel.
- When an ant finishes creating its solution, it sends a message containing the solution and its total tardiness to a supervisor process.
- The supervisor compares the incoming messages against the current best solution.

- After all A results have been received, the supervisor uses the new best solution to update the pheromone matrix.
- A new iteration is then started: again, each ant constructs a new solution.
- After K iterations, the program terminates and outputs its best solution.

4.1 Difficulties with Erlang

Two factors make Erlang somewhat unsuitable for implementation of the ACO application.

Firstly, a considerable amount of integer and floating-point calculation (including exponentiation) is performed. There will be some overhead in performing this in the Erlang VM, rather than directly in C, for example.

More seriously, it is necessary to allocate large amounts of heap space during the calculation. As mentioned above, it is necessary to calculate some heuristic information, conceptually stored in a matrix η. In fact, we store this information in a list of pairs $\{j, \eta_{ij}\}$ where j is the (integer) identifier of a job, and η_{ij} is the heuristic information indicating the desirability of scheduling job j at the current position i in the schedule. We only require information for unscheduled jobs j, but η_{ij} depends on the jobs which have already been scheduled. This means that we must construct a new list every time an ant schedules a new job, and thus during the construction of a single schedule each ant must allocate $N + (N-1) + \ldots + 2 = (N^2 + N - 2)/2$ list cells (each containing a pair $\{j, \eta_{ij}\}$); when we get to the last job, there's only one choice, so we don't have to allocate a list of length 1). For $N = 1000$, this is 500,499 heap cells which have to be allocated every time an ant constructs a solution (and remember that we have multiple ants, each constructing a new solution for every iteration of the main loop of the ACO algorithm). In contrast, in C we could use a single array of length N in conjunction with an array saying which jobs are currently unscheduled, and reuse these every time we add a job to the schedule. This would be a massive saving in heap allocation, and it may well be worth using Erlang's native interface to implement a simple library of mutable arrays with `get` and `set` functions.

Addendum. I tried to implement mutable arrays using NIFS, but actually it slowed the whole thing down by about another 50%. I'm guessing (but I haven't read the NIF source code to make sure) that the problem here is that it's possible to get floats out of Erlang objects and into C arrays, but there's quite a lot of overhead because you have to unbox an Erlang heap object to get a C double when you write an element of the array, and then when you read an element you have to re-box it into an Erlang object again (and this of course requires some new memory to be allocated in the heap).

One could try to overcome this by storing pointers to boxed objects in the array, but I think that wouldn't work because as far as I can see there's no way to tell the Erlang garbage collector that you're keeping a pointer to something that it owns, and then there's a danger that your pointer would become invalid because the garbage collector might delete or move the object you're pointing to. In short, I think that was a dead-end: they really don't want you making mutable data structures.

4.2 Behaviour

The execution time of the program is linear in K and quadratic in N, due to the allocation issues discussed above, and also due to the fact that since the entire $N \times N$ matrix τ has to be traversed by each ant and later rewritten in the global update stage.

It's more difficult to see how the execution time is influenced by M (the number of ants), and we carried out extensive experiments on a number of different machines to examine this. We used two different versions of the program, one with the pheromone matrix stored in an ETS table and the other using a DETS table. ETS (Erlang Term Storage) tables are kept in memory and disappear when the VM shuts down; tables are visible to all processes running in the same VM, but are not visible to other VMs (even ones running on the same host). DETS tables as stored on disk and are persistent, but have the advantage that they can be accessed by nodes running on different machines if the machines happen to share a disk (over NFS, say). The code for the two versions was largely identical, with only minor changes required for the two types of table.
Detailed information is given later in Appendix A, but the basic result is that for the ETS version the execution time appears to vary linearly with the number of ants. One might expect that the execution time would be roughly constant when the number of ants is less than the number of CPUs, and then would increase linearly. Oddly, this doesn't appear to happen: in most cases the trend is entirely linear, with no visible change when the number of ants exceeds the number of processors.

For the DETS version, execution times were also more or less linear, but execution times were many times slower (by a factor of 6 to 20, depending on the machines involved and the size of the input). I initially tried this with the DETS table stored on an NFS filesystem, but then later tried it with the table stored in the `/scratch` partition on the local disk: this way generally slightly faster (but never more than 2%), so it would appear that using DETS tables is not a good way for the ACO application to share data. Persistence is also something of a nuisance because if a program happens to crash or is terminated early, you're left with a disk copy of the DETS table which must be removed manually.

5 Distributed ACO techniques

Our main interest is in distributed applications, so I have also implemented a distributed version of the ACO program. A plausible strategy is to have separate colonies operating on separate machines, either completely independently or with some communication.

Some research has been done on systems involving multiple ant colonies in the context of the Travelling Salesman Problem (see [17, 24], for example). Here, several ant colonies construct solutions independently but occasionally exchange information. This has at least two advantages:

- Colonies which have got stuck at a local minimum have a chance to escape, or are just ignored.
- It appears that strategies which may be efficient for solving some ACO instances may not be so efficient for others. Using a multi-colony approach, different colonies can use different heuristics and different values of the parameters α, β, ρ, and q_0 mentioned above.

This is encouraging, because it suggests that there may be real benefits to be obtained by using a distributed approach. One might consider sharing entire pheromone matrices between colonies (which would be expensive, due to high communication due to the large size of the matrices), but it seems that it's better to share only *the best solution*: for example, [24][§4] says

> *The results of Krüger, Middendorf, and Merkle (1998) for the Traveling Salesperson problem indicate that it is better to exchange only the best solutions found so far than to exchange whole pheromone matrices and add the received matrices, multiplied by some small factor, to the local pheromone matrix.*

This makes sense, because instead of restarting each colony from an identical point after every synchronisation, it allows colonies to influence each other while retaining some individuality. This reduces the amount of data transfer significantly: instead of having to transmit N^2 floats, we only have to transfer N integers.

A number of different strategies have been proposed for exchanging information: for example, propagating the global best solution to every colony, or arranging the colonies in a ring where each colony shares its best solution with its successor. The relative merits of several strategies are considered in [24].

5.1 Implementation strategy

The fact that pheromone matrices don't have to be shared in their entirety is encouraging. The experiments described in Appendix A below show that ETS tables are substantially better than DETS tables for storing pheromone information, and if individual colonies don't have to share too much information then each colony can keep its pheromone information in its own ETS table.

My initial approach has been to implement individual colonies using essentially the same code as the SMP version of the program described earlier. A number of Erlang VMs are started on different machines in a network (possibly with more than one VM per machine), and each of these will run a single colony. There is a single master node which starts the colonies and distributes information (like the input data and parameter settings) to them; each colony runs a specified number of ants for a specified number of generations, and then reports its best solution back to the master. The master chooses the best solution from among the colonies and then broadcasts it to all of the colonies, which use it to update their pheromone matrices and then start another round of iterations. This entire process is repeated some number of times, after which the master shuts down the colonies (but leaves their VMs running) and reports the best result. The current implementation is quite simplistic, only using basic features of Distributed Erlang; all communication is done via PIDs (not global names) and there's no attempt at fault-tolerance.

Initial experiments give good results, with optimal solutions being found for almost all of the 40-job instances from ORLIB in a few seconds (for high-quality results, we can use an optional (and inefficiently implemented) local search strategy which increases solution time by 50% or so for the small problems we've looked at so far) . It'll be more challenging to test the program on larger examples.

Currently, each colony uses the same parameter settings, but the code allows this to be easily changed at the point where the master initialises the colonies; I hope to experiment with this shortly. It would also be interesting (especially from the point of view of SD Erlang) to experiment with other communication topologies.

Fault tolerance How fault-tolerant do we wish to be? In some situations it's probably not too important if a single colony fails, since the others will just be able to carry on independently. This depends strongly on the information-exchange strategy though. For example, in the ring-shaped topology mentioned above, the failure of a single colony would be disastrous, since it would break the communication process completely. On the other hand, the inital strategy which we have adopted (communicating a single global best solution between multiple colonies) has a dedicated process which collects solutions from the colonies and then distributes the best solution to all the colonies. Failure of this supervising process would be problematic since the colonies would then need to come to a consensus on who should take over the supervision.

Another issue which might become interesting in a distributed multi-colony setting is that there may be some heterogeneity in the systems. If one machine is five times as fast as the others, will it spend 80% of its time idle while it waits for the others to catch up? Perhaps the answer to this would be to run multiple colonies on separate Erlang nodes on the fast machine, or to get it to perform a larger number of iterations of its main loop. This kind of information might be something which we could discover using SD-Erlang's *attributes.*

6 Measuring Scalability

One issue which we haven't mentioned yet is the question of how we measure scalability. Standard measures of scalability consider speedups obtained from running an algorithm on multiple cores or machines, but this may not be appropriate for the ACO application. In contrast to problems which can be divided into sub-problems which can be solved separately, we don't necessarily run any *faster* by making our application distributed, but we may improve the *quality* of our solution since multiple colonies can construct solutions independently, and there's more chance of finding a good solution with more colonies.

We could of course make an entirely sequential version of our application, where each colony runs its first ant, then its second, and so on, for a number of generations; we would also run the colonies in sequence, comparing results once every colony has finished, and then starting a new iteration. Suppose we have the following:

- C colonies
- K_1 iterations of the main loop in the master process (each involving one activation of the colonies)
- K_2 generations of ants for each activation of the colony
- A ants per colony

Then if the time taken for a single ant to construct a solution is T_0, the total time required for a completely sequential execution of the system will be of the order of

$$CK_1K_2AT_0.$$

If we take fairly conservative figures for these, say $C = 40, A = 20, K_1 = 100, , K_2 = 50$, then we'd expect the purely sequential version to take roughly 4,000,000 times longer than a distributed version. This may be rather too large to measure. It might be more realistic to measure the overhead incurred by running multiple colonies in a distributed system in comparison with a single colony running on a single machine.

A completely different way to measure improvement might be in terms of solution quality, and this seems to be something that's done in some papers on Ant Colony Optimisation. For example, optimal solutions are known for the ORLIB instances mentioned above, and one measure of quality that's been used is to apply an ACO algorithm several times to each instance and then report the number of exact solutions found and the average deviation from the optimal cost: see [7, 21] for example.

Yet another technique which has been used (see [21]) is to measure the average time taken for the ACO algorithm to converge on the known optimal solution is found, but I'm not sure exactly how one would go about this. The behaviour of the SMP version of the program suggests that in many cases the algorithm will converge on a local minimum and stay there. Other runs with the same input may find the global best solution, but the stochastic nature of the ACO algorithm makes it hard to predict if this will happen, or if an execution which appears to have got stuck might in fact escape to a better solution if given sufficient time.

Perhaps a reasonable way to proceed would be to look at both of the first two methods. We could look at small runs of the SMP version and use this to predict how long a full run would take, and then compare that time with the time taken for the distributed version to perform a complete run. In tandem with this, we could use metrics of solution quality to show that distribution gives good solutions in reasonable time.

References

[1] A. Bauer, B. Bullnheimer, R.F. Hartl, and C. Strauss. An ant colony optimization approach for the single machine total tardiness problem. In *Evolutionary Compu-*

tation, 1999. CEC 99. Proceedings of the 1999 Congress on, volume 2, pages –1450 Vol. 2, 1999.

[2] Matthijs Den Besten, Thomas Sttzle, and Marco Dorigo. An ant colony optimization application to the single machine total weighted tardiness problem, 2000.

[3] W. Bozejko and M. Wodecki. A fast parallel dynasearch algorithm for some scheduling problems. In *Parallel Computing in Electrical Engineering, 2006. PAR ELEC 2006. International Symposium on*, pages 275–280, Sept 2006.

[4] Bernd Bullnheimer, Gabriele Kotsis, and Christine Strauss. Parallelization Strategies for the Ant System. In R. De Leone, A. Murli, P. Pardalos, and G. Toraldo, editors, *High Performance Algorithms and Software in Nonlinear Optimization*, volume 24 of *Applied Optimization*, pages 87–100. Kluwer, Dordrecht, 1997.

[5] Richard K. Congram, Chris N. Potts, and Steef L. van de Velde. An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. *INFORMS Journal on Computing*, 14(1):52–67, 2002.

[6] H. A. J. Crauwels, C. N. Potts, and L. N. van Wassenhove. Local search heuristics for the single machine total weighted tardiness scheduling problem. *INFORMS Journal on Computing*, 10(3):341–350, 1998.

[7] Matthijs den Besten, Thomas Stützle, and Marco Dorigo. Ant colony optimization for the total weighted tardiness problem. In Marc Schoenauer, Kalyanmoy Deb, Gnther Rudolph, Xin Yao, Evelyne Lutton, JuanJulian Merelo, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature PPSN VI*, volume 1917 of *Lecture Notes in Computer Science*, pages 611–620. Springer Berlin Heidelberg, 2000.

[8] Matthijs den Besten, Thomas Stützle, and Marco Dorigo. Design of iterated local search algorithms. In Egbert J. W. Boers, Jens Gottlieb, Pier Luca Lanzi, Robert E. Smith, Stefano Cagnoni, Emma Hart, Gnther R. Raidl, and H. Tijink, editors, *EvoWorkshops*, volume 2037 of *Lecture Notes in Computer Science*, pages 441–451. Springer, 2001.

[9] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization.* Bradford Company, Scituate, MA, USA, 2004.

[10] Marco Dorigo and Thomas Stützle. Ant colony optimization: Overview and recent advances. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, pages 227–263. Springer US, 2010.

[11] Jianzhong Du and Joseph Y.-T. Leung. Minimizing total tardiness on one machine is np-hard. *Mathematics of Operations Research*, 15(3):483–495, 1990.

[12] Hamilton Emmons. One-machine sequencing to minimize certain functions of job tardiness. *Operations Research*, 17(4):pp. 701–715, 1969.

[13] Martin Josef Geiger. The single machine total weighted tardiness problem – is it (for metaheuristics) a solved problem ? *CoRR*, abs/0907.2990, 2009.

[14] Martin Josef Geiger. New instances for the single machine total weighted tardiness problem. Technical Report Research Report 10-03-01, March 2010.

[15] Martin Josef Geiger. On heuristic search for the single machine total weighted tardiness problem – some theoretical insights and their empirical verification. *European Journal of Operational Research*, 207(3):1235 – 1243, 2010.

[16] A. Grosso, F. Della Croce, and R. Tadei. An enhanced dynasearch neighborhood for the single-machine total weighted tardiness scheduling problem. *Oper. Res. Lett.*, 32(1):68–72, January 2004.

[17] H. Kawamura, M. Yamamoto, K. Suzuki, and A. Ohuchi. Multiple Ant Colonies Algorithm Based on Colony Level Interactions. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E83-A(2):371–379, 2000.

[18] Eugene L. Lawler. A "pseudopolynomial" algorithm for sequencing jobs to minimize total tardiness. In B.H. Korte P.L. Hammer, E.L. Johnson and G.L. Nemhauser, editors, *Studies in Integer Programming*, volume 1 of *Annals of Discrete Mathematics*, pages 331–342. Elsevier, 1977.

[19] Jan Karel Lenstra, AHG Rinnooy Kan, and Peter Brucker. Complexity of machine scheduling problems. *Annals of discrete mathematics*, 1:343–362, 1977.

[20] Robert McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6(1):1–12, 1959.

[21] Daniel Merkle and Martin Middendorf. An ant algorithm with a new pheromone evaluation rule for total tardiness problems. In *Proceedings of EvoWorkshops 2000, volume 1803 of LNCS*, pages 287–296. Springer Verlag, 2000.

[22] R. Michel and M. Middendorf. An Island Model Based Ant System with Lookahead for the Shortest Supersequence Problem. In A.E. Eiben, T. Bäck, H.-P. Schwefel, and M. Schoenauer, editors, *Proceedings of the Fifth International Conference on Parallel Problem Solving from Nature (PPSN V)*, pages 692–701. Springer-Verlag, New York, 1998.

[23] R. Michel and M. Middendorf. An ACO Algorithm for the Shortest Common Supersequence Problem. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimisation*, pages 51–61. McGraw-Hill, London, UK, 1999.

[24] Martin Middendorf, Frank Reischle, and Hartmut Schmeck. Multi colony ant algorithms. *Journal of Heuristics*, 8(3):305–320, May 2002.

[25] T.E. Morton, R.M. Rachamadugu, and A. Vopsalainen. Accurate myopic heuristics for tardiness scheduling. Technical Report GSIA Working Paper 36-83-84, 1984.

[26] C. N. Potts and L. N. Van Wassenhove. Single machine tardiness sequencing heuristics. *IIE Transactions*, 23(4):346–354, 1991.

[27] NR Srinivasa Raghavan and M Venkataramana. Parallel processor scheduling for minimizing total weighted tardiness using ant colony optimization. *The International Journal of Advanced Manufacturing Technology*, 41(9-10):986–996, 2009.

[28] Marcus Randall and Andrew Lewis. A parallel implementation of ant colony optimization. *Journal of Parallel and Distributed Computing*, 62(9):1421–1432, 2002.