A Genetic Algorithm for the Two Machine Flow Shop Problem

Kumar Adusumilli
School of Computer Science
University of Nevada
Las Vegas, NV 89154
kumar.adusumilli@gmail.com

Doina Bein
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083
siona@utdallas.edu

Wolfgang Bein *
Center for the Advanced Study of Algorithms
School of Computer Science
University of Nevada
Las Vegas, NV 89154
bein@cs.unlv.edu

Abstract

In scheduling, the two machine flow shop problem $F2||\sum C_i$ is to find a schedule that minimizes the sum of finishing times of an arbitrary number of jobs that need to be executed on two machines, such that each job must complete processing on machine 1 before starting on machine 2. Finding such a schedule is $\mathcal{NP}$-hard [6]. We propose a heuristic for approximating the solution for the $F2||\sum C_i$ problem using a genetic algorithm. We calibrate the algorithm using optimal results obtained by a branch-and-bound technique. Genetic algorithms simulate the survival of the fittest among individuals over consecutive generations for solving a problem. Prior work has shown that genetic algorithms generally do not perform well for shop problems [21]. However, the fact that in the case of two machines the search space can be restricted to permutations makes the construction of effective genetic operators more feasible.

1 Introduction

John Holland [11] at University of Michigan conceived of genetic algorithms in the early 1970 in order to solve optimization problems, by using random search. Genetic algorithms are a class of adaptive heuristic search techniques which exploit gathered information to direct the search into regions of better performance within the search space. In terms of time complexity, compared with other optimization techniques such as integer linear programming, branch and bound, tabu search, they may offer a good approximation for the same big-O time when the state-space is large. (See also [8, 9, 16].)

Flow shop problems are a distinct class of shop scheduling problems [4, 5, 10, 13, 14], where n jobs ($i = 1, \ldots, n$) have to be performed on m machines ($j = 1, \ldots, m$) as follows. A job consists of m operations, the j^{th} operation of each job must be processed on machine j and has processing time p_{ij}. A job can start only on machine j if its operation is completed on machine $(j-1)$ and if machine j is free. The completion time of job i, C_i, is the time when its last operation has completed. This problem is denoted in the literature in $\alpha|\beta|\gamma$-notation (see e.g. [4]) as $Fm||\sum C_i$.

Consider an example of flow shop with three machines with the following data (Table 1).

Job i	p_{i1}	p_{i2}	p_{i3}
$J1$	1	2	3
$J2$	1	2	1
$J3$	1	1	1

Table 1. Three Machine Flow Shop

Figure 1 and Figure 2 show two feasible schedules for the

*Research supported by NSF grant CCR-0312093. Research conducted while on sabbatical leave from the University of Nevada, Las Vegas. Sabbatical support from UNLV is acknowledged.

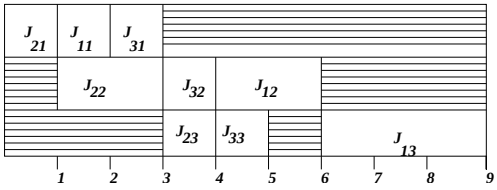


Figure 1. Flow Shop for 3 Machines, Case(i)

example. Note that in both schedules the order of the jobs differs across machines. For the case (i), we have $C_{\max} = 9$ and $\sum C_i = 18$. In case (ii), $C_{\max} = 8$ and $\sum C_i = 21$. Note that $\sum C_i$ is better than $C_{\max}$ in case (i) whereas it is the opposite in case (ii). The example suggests that things very much depend on the objective function.

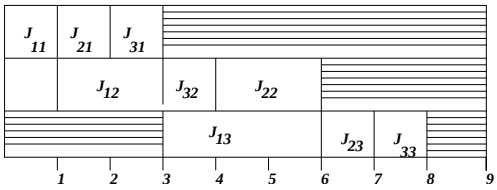


Figure 2. Flow Shop for 3 Machines, Case(ii)

The work here focuses on the case $m = 2$ where the objective is to minimize the sum of completion time ($\sum C_i$), or equivalently the average completion time; thus we consider the flow shop problem $F2||\sum C_i$ with n jobs. Flow shop problems are well studied; section 6.6 of [4] give a comprehensive overview of results. Non withstanding this, there is still wide interest in the problem, even when $m = 2$. For example, very recently, Oulamara [18] considered makespan minimization for no-wait ow shop problems on two batching machines. (For Batching machines the completion time of a job is the completion time of the batch the job is part of.) Independently, Liaw [15] developed heuristic for minimizing the makespan for two-machine no-wait job shop problems. In this setting operations must be performed without any interruption on machines and without any waiting in between machines. We also mention that Allaoui *et al.* [1] studied the problem of scheduling n immediately available jobs in a flow shop composed of two machines in series with the objective of minimizing the makespan. Blazewics *et al.* [3] have studied the variant of the problem where a total weighted late work criterion and a common due date ($F2|d_i = d|Y_w$) is given.

Genetic algorithms for shop problems were extensively studied by Wall [21] in the context of adaptive approaches to resource-constrained scheduling. However the approach did not work well for general problems; Wall reports: "Performance on the job shop problems was less encouraging." For the $F2||\sum C_i$ problem, however, Theorem 3.6 of Brucker [4] shows that there is an optimal solution where both machines have the same scheduling order for the jobs. Thus an optimal schedule may be represented by a job permutation and a permutation fully describes the solution. Computing the order is $\mathcal{NP}$–hard (Garey *et al.* [6]). Still, the fact that in the case of two machines the search space is restricted to permutations makes the construction of effective genetic operators more feasible.

We note that in contrast, the problem $F2||C_{\max}$ is to find a schedule, which minimizes the $C_{\max} = \max\{C_i, i = 1, \ldots, n\}$ (the so called *makespan*). For arbitrary processing times, this problem is the only flow shop problem that is polynomially solvable. The optimal solution is given by Johnson's algorithm (Johnson [12]).

Contributions. We first show that the schedule produced by Johnson's algorithm, which was designed for makespan minimization, can be arbitrarily bad for average completion $\sum C_i$. We construct a genetic algorithm for the $F2||\sum C_i$ problem and we benchmark the algorithm using various sets of jobs, with the optimal schedules obtained by using a branch-and-bound technique. Implementations are written under a Linux Fedora environment, and run under GNU g++ compiler in conjunction with GAlib.

The chromosome structure used in our genetic algorithm is the same as in the Traveling Salesman Problem (TSP) [17] i.e. an array of unique integers. Another crossover that can be used is the PMX of Goldberg and Lingle [7].

Outline. In Section 2 we discuss Johnson's algorithm and show that it can produce solutions arbitrarily far from optimal for $F2||\sum C_i$ problem. In Section 3 we briefly review the concept of genetic algorithms and describe a genetic algorithm used for solving $F2||\sum C_i$. The implementation of the algorithm utilizes GAlib, the object-oriented library of Matthew Wall [20] developed at MIT. Comparative results for solutions offered by Johnson's algorithm, branch-and-bound (optimal), and our genetic algorithm are presented in Section 4. We conclude in Section 5.

2 Using Johnson's Algorithm

Johnson's algorithm gives an optimal solution to the $F2||C_{\max}$ problem and all the jobs are scheduled on the same order for both machines. It creates two partial schedules, L and R. The final schedule T (the same for the both machine) is obtained by concatenating L and R (see Algorithm 1).

From the set X of all jobs that are not scheduled yet, at time t consider the job i that has the smallest processing time for either machine: the smallest value of p_{i1} or p_{i2} where $i \in \{1, \ldots, n\}$. If job i has smallest p_{i1} value then job i is removed from X and added to the tail of L i.e, $L \circ i$ and if otherwise job i is added to the front of R i.e, $i \circ R$.

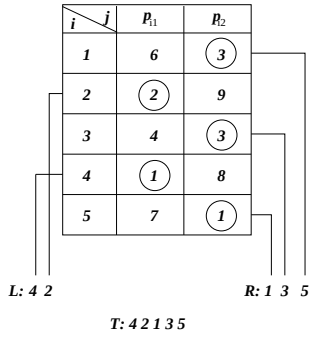
Algorithm 1 Johnson's Algorithm

1. $X := \{1, \ldots, n\}$; $L := \emptyset$; $R := \emptyset$;
2. **while** $X \neq \emptyset$ **do**
   BEGIN
3. Find job i that has smallest p_{i1} or p_{i2}.
4. **if** p_{i1} is the smallest **then** $L := L \circ i$ **else** $R := i \circ R$;
5. $X := X \setminus \{i\}$
   END
6. $T := L \circ R$

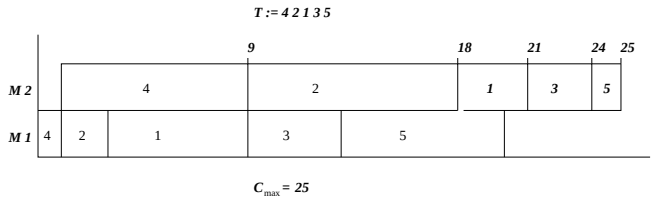
This is done until X becomes empty (all the jobs have been scheduled in T and R).

Initially let $X = \{1, \ldots, i, \ldots, n\}$ be the set of all jobs.

The example in Figure $3(a)$ shows how Johnson's algorithm works for a set of 5 jobs, where i represents the job number and j represents the machine. The optimal schedule is presented in Figure $3(b)$.



(a) Selecting the jobs



(b) Optimal schedule T of the jobs

Figure 3. Johnson's Algorithm for $n = 5$

To show that Johnson's algorithm gives an arbitrarily large solution for the $F2||\sum C_i$ problem, consider the following flow shop that has n jobs (see Table 2). The value ϵ is considered very small and the value k very large. We refer to the n^{th} job as the "large" job.

Job i	p_{i1}	p_{i2}
1	ϵ	ϵ
2	ϵ	ϵ
$\vdots$		
n	$\epsilon/2$	k

Table 2. Example of A 2-machine Flow Shop Problem

For the data given in Table 2, it is obvious that the optimal schedule for $\sum C_i$ would schedule the large job last, after jobs $1, \ldots, (n-1)$. Thus $\sum C_i$ is equal to

$$\begin{aligned}\sum C_i &= C_1 + C_2 + \ldots + C_n \\ &= 2\epsilon + 3\epsilon + \ldots + n\epsilon + (n\epsilon + \epsilon/2 + k) \\ &= \frac{n(n+3)-1}{2}\epsilon + k \\ &= \text{lower order terms} + k\end{aligned}$$

Johnson's algorithm schedules the large job first, followed by jobs $1, \ldots, (n-1)$. Thus $\sum C_i$ is equal to

$$\begin{aligned}\sum C_i &= C_1 + C_2 + \ldots + C_n \\ &= (\epsilon/2 + k) + (k + (\epsilon)) + (k + (\epsilon + \epsilon)) + \ldots \\ &+ (k + (\epsilon + \cdots + \epsilon)) \\ &= nk + \frac{n(n+1)+1}{2}\epsilon \\ &= nk + \text{lower order terms}\end{aligned}$$

If n is arbitrarily large, then Johnson's algorithm gives an arbitrarily bad solution.

3 Using Genetic Algorithms

In a genetic algorithm a fixed size set of individuals (called *generation*) is maintained within a search space, each representing a possible solution to the given problem. The individuals in the generation go through a process of evolution. A *fitness score* is assigned to each solution representing the abilities of an individual to "compete". The individual with the optimal (or near optimal) fitness score is sought. The individuals with lower values are removed and newer ones, added by the "breeding" process – by combining information from the parents' components – are added.

After an initial population is randomly generated, the algorithm evolves through three operators: *selection* represents the paradigm of survival of the fittest, *crossover* mimics mating between individuals, and *mutation* introduces random modifications.

To maintain diversity within the population and inhibit premature convergence, some characteristics of the "offsprings" are randomly modified. Mutation alone induces a random walk through the search space. A new generation is created once all combinations in the old population have been exhausted. Eventually, once the current population is not producing offsprings noticeably different from those in previous generation(s), the algorithm itself is said to have *converged* to a set of solutions to the problem at hand.

A genetic algorithm has the following structure:

1. Randomly initialize population (at time t).
2. Determine fitness of population (at time t).
3. Repeat the following until the best individual is found:
   (a) Select parents from population (at time t).
   (b) Perform crossover on parents creating population (at time $t+1$).
   (c) Perform mutation of population (at time $t+1$).
   (d) Determine fitness of population (at time $t+1$).

In the case of the 2-machine flow shop problem, an individual is represented by a permutation. The fitness of a permutation is the $\sum C_i$-value of the corresponding schedule. We have to define mutation and crossover. A mutation simply swaps two arbitrary elements of the permutation. For the crossover it is important to devise a mechanism that retains some features of the original two individuals in such a meaningful way that results in two new permutations. In the ordered crossover first used by Prins (see [19]), one takes a random subsequence of the first parent's permutation and insert it directly into the child. As described in Figure 4, the child is then completed by taking material from the second parent's permutation, where elements are inserted into the child in the order they occur in that parent, starting after the second cut location, and ignoring elements already inserted from the first parent.

Parent 1	$184\underbrace{637}\|25$	random slice 637
Parent 2	$3\underline{5}2\underline{7}\underline{1}\underline{8}\|6\underline{4}$	add underlined
Child	$218\underline{637}\|45$	child is a valid permutation

Figure 4. Example of Ordered Crossover

We have implemented a genetic algorithm for the 2-machine flow shop problem using Matthew Wall's GAlib [20]. GAlib is a C++ library developed at MIT, designed to assist in the development of genetic algorithm applications. Our programs are written under a Linux Fedora environment, and run under GNU g++ compiler in conjunction with GAlib. When programming using GAlib, one works primarily with two classes: a genome class and a genetic algorithm class. A genome instance represents a single individual in the population of solutions. The genetic algorithm defines how the solution will be evolved. In addition to defining these two classes, an objective function is needed. The three necessary steps to develop an application using GAlib are:

- define a representation
- define the genetic operators: initialize, mutate, and crossover
- define the objective function

In our case everything, except for the evaluation algorithm defining the objective function, was available readily in GAlib.

Genetic algorithms generally do not provide lower bounds. Branch-and-bound can be used as method to solve combinatorial optimization problems, by intelligently enumerating all feasible solutions.

Assume that there are subproblems of P which are defined by a subsets S' of the set S of feasible solution of P.

Three things are needed for a branch-and-bound algorithm.

1. **Branching:** S is replaced by smaller problems $S_i (i = 1, \ldots, r)$ such that $\cup_{i=1}^{r} S_i = S$.
2. **Lower Bounding:** An algorithm is available for calculating a lower bound for the objective values of all feasible solutions of a subproblem.
3. **Upper Bounding:** We calculate an upper bound $upperBound$ of the objective value of P. The objective value of any feasible solution will provide such an upper bound. If the lower bound of a subproblem is greater than or equal to $upperBound$, then this subproblem cannot yield a better solution.

In the case of the $F2||\sum C_i$ problem, we use the branch-and-bound algorithm presented in Brucker [4]. A natural way to branch is to choose the first job to be scheduled at the first level of branching tree, the second job at the next level, and so on. Thus the basic idea of this algorithm is to consider subproblems, where r jobs have been scheduled. Algorithm Branch-and-Bound summarizes these basic ideas.

As an example, consider Figure 5. Here, the number of jobs is 4. For example the node (23) represents the fact that jobs 2 and 3 are fixed in this order and jobs 1 and 4 could still be in any order after jobs $2, 3$. In general, suppose we are at node at which the jobs in the set $M \subseteq \{1, \cdots, n\}$ have been scheduled, where $|M| = r$. The cost of this schedule, which we wish to bound, is

$$S = \sum_{i \in M} C_i + \sum_{i \notin M} C_i$$

For the second sum, Brucker [4] derives two possible lower bounds based on assumptions:

Algorithm 2 The Branch-and-Bound Algorithm

1. $lowerBound, upperBound = feasiblesolution$ GENERATE_NODES(a, i)
2. IF $i = n$ THEN $currentSolution = calsch(n, a)$ END IF
3. IF $currentSolution < upperBound$ THEN UPDATE $upperBound$
   ELSE
   (a) CALCULATE $lowerBound$
   (b) IF $lowerBound \geq upperBound$ THEN CUT
   ELSE
   i. FOR $i + 1$ TO n DO
   BEGIN
   ii. SWAP
   iii. CALL GENERATE_NODES$(a, i+1)$ END FOR
   END IF
   END IF

1. Every job $i \notin M$ completes its processing without delay from machine 1.
2. Every job $i \notin M$ starts its processing on machine 2 without delay from machine 2.

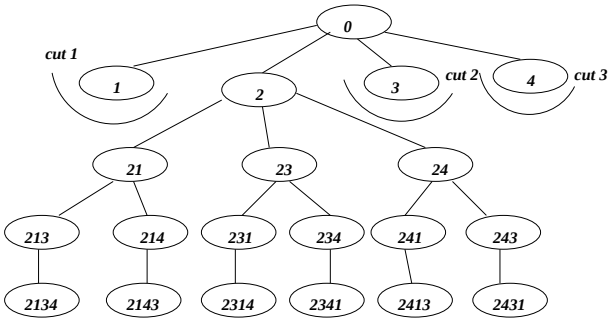


Figure 5. $n = 4$**, Branch and Bound Tree after Pruning**

We note that the branch-and-bound algorithm is exponential in its run time, and, unlike the genetic algorithm cannot be used for larger values of n. But it is useful to calibrate the genetic algorithm.

4 Simulations and Results

The following results are developed using Johnson's algorithm (JA), branch-and-bound (BB), and a genetic algorithm (GA) for two machine flow shop scheduling problem.

Two assumptions are made:

1. When implementing branch-and-bound, we calculate an initial feasible solution which is the sum of completion time all the processes in the ascending order.

2. When implementing a genetic algorithm, the mutation probability is 0.01 and the crossover probability is 0.85. These parameters were found after extensive experimentation. Lower crossover probabilities slowed convergence and other mutation probabilities did not work well. The choice of these parameters was also guided by our earlier work on traveling salesman problems [2].

The following results are obtained by applying Johnson's algorithm, branch-and-bound algorithm and a genetic algorithm to randomly chosen p_{i1} and p_{i2} values. When more runs are executed for a GA, the results are separated by commas.

Table 3 contains randomly selected p_{i1} and p_{i2} for up to 20 jobs.

Job i	p_{i1}	p_{i2}
1	6	3
2	2	9
3	4	3
4	1	8
5	7	1
6	4	5
7	7	6
8	9	3
9	6	4
10	7	2
11	5	8
12	9	3
13	2	6
14	4	3
15	6	5
16	3	6
17	9	2
18	4	6
19	8	5
20	8	1

Table 3. Random p_{i1} **and** p_{i2} **for** n **up to** 20

For $n = 5$ and randomly selected p_{i1} and p_{i2} given in Table 3, by running JA, BB, and GA algorithms the results for the objective function $\sum C_i$ are presented in Table 4.

$n = 5$	JA	BB	GA with gen=150, pop=50
$\sum C_i$	97	83	83

Table 4. $\sum C_i$ **Results for** $n = 5$

For $n = 7$ and randomly selected p_{i1} and p_{i2} given in Table 3, by running JA, BB, and GA algorithms the results for the objective function $\sum C_i$ are presented in Table 5.

$n = 7$	JA	BB	GA with gen=150, pop=50
$\sum C_i$	182	150	150

Table 5. $\sum C_i$ Results for $n = 7$

For $n = 10$ and randomly selected p_{i1} and p_{i2} given in Table 3, by running JA, BB, and GA algorithms the results for the objective function $\sum C_i$ are presented in Table 6.

$n = 10$	JA	BB	GA with gen=150, pop=50
$\sum C_i$	331	292	297,292,294,295

Table 6. $\sum C_i$ Results for $n = 10$

For the particular case when the total processing time $p_{i1} + p_{i2} = n + 1$, the integer values for p_{i1} and p_{i2} vary in the set $\{1, \ldots, n\}$. For $n = 10$ and p_{i1} increasing as the job index increases (thus p_{i2} decreases) (Table 7), by running JA, BB, and GA algorithms the results for the objective function $\sum C_i$ are presented in Table 8.

Job i	p_{i1}	p_{i2}
1	1	10
2	2	9
3	3	8
4	4	7
5	5	6
6	6	5
7	7	4
8	8	3
9	9	2
10	10	1

Table 7. Increasing p_{i1} and Decreasing p_{i2}

$n = 10$	JA	BB	GA with gen=200, pop=50
$\sum C_i$	395	337	340,343,339,337

Table 8. $\sum C_i$ Results for $n = 10$

For $n = 10$ and p_{i1} decreasing as the job index increases (thus p_{i2} increases) (Table 9), by running JA, BB, and GA algorithms the results for the objective function $\sum C_i$ are presented in Table 10.

For $n = 10$ and p_{i1} increasing as the job index increases and then decreasing after $i > (n + 1)/2$ (Table 11), by running JA, BB, and GA algorithms the results for the objective function $\sum C_i$ are presented in Table 12.

For $n = 15$ and randomly selected p_{i1} and p_{i2} given in Table 3, by running JA, BB, and GA algorithms the results for the objective function $\sum C_i$ are presented in Table 13.

Job i	p_{i1}	p_{i2}
1	10	1
2	9	2
3	8	3
4	7	4
5	6	5
6	5	6
7	4	7
8	3	8
9	2	9
10	1	10

Table 9. Decreasing p_{i1} and Increasing p_{i2}

$n = 10$	JA	BB	GA with gen=200, pop=50
$\sum C_i$	395	337	343,343,342,343

Table 10. $\sum C_i$ Results for $n = 10$

Job i	p_{i1}	p_{i2}
1	1	10
2	2	9
3	3	8
4	4	7
5	5	6
6	5	6
7	4	7
8	3	8
9	2	9
10	1	10

Table 11. p_{i1} Increasing, then Decreasing

$n = 10$	JA	BB	GA with gen=200, pop=50
$\sum C_i$	490	430	431,434,431,431

Table 12. $\sum C_i$ Results for $n = 10$

$n = 15$	JA	BB	GA (gen=150, pop=50)	GA (gen=200, pop=50)
$\sum C_i$	717	601	615	620
			621	628
			620	627

Table 13. $\sum C_i$ Results for $n = 15$

For $n = 20$ and randomly selected p_{i1} and p_{i2} (Table 3) by running JA, BB, and GA algorithms the results for the objective function $\sum C_i$ are presented in Table 14.

From our simulations we observe that the results obtained by genetic algorithms are close to the results obtained by branch-and-bound, while the results of Johnson's algorithm are always worse.

$n = 20$	JA	BB	GA with gen=500, pop=50
$\sum C_i$	1247	1054	1096, 1087, 1103

Table 14. $\sum C_i$ Results for $n = 20$

Scalability. As noted before the branch-and-bound algorithm is exponential in its run time, and, unlike the genetic algorithm cannot be used for larger values of n. Its purpose is to calibrate the genetic algorithm. In our simulations we have used relatively small values of n and thus the genetic algorithm uses moderate populations sizes (less than 100) and converges quickly (less than 500 generations). Thus there is a reasonable expectation that the genetic algorithm will scale up favorably.

5 Conclusion

In this paper, we propose a heuristic based on genetic algorithms to approximate the two machine flow shop problem $F2||\sum C_i$. To calibrate our genetic algorithm we show that for smaller numbers of jobs (n) the results are comparable with the optimal schedule (obtained by using branch-and-bound technique). In our simulations we obtain for small values of n a difference of 2% between the results obtained by our genetic algorithm and branch-and-bound algorithm.

Finally we show that the schedule produced by Johnson's algorithm can be arbitrarily bad for weighted average completion $\sum C_i$ for the $F2||\sum C_i$ problem.

References

[1] H. Allaoui, A. Artiba, and E. Aghezzaf. Simultaneoulsy scheduling n jobs and the preventive maintenance on the two-machine flow shop to minimize the makespan. *International Journal of Production Economics, In Press, Corrected Proof, Available online April 10, 2007*, 2007.

[2] Wolfgang Bein and Bradley Hendricks. Pitfalls with adaptive methods for combinatorial optimization: The traveling salesman - a case study. In *Proceedings of the International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences - METMBS*, volume 1, pages 243–248. CSREA Press, 2002.

[3] Jacek Blazewicz, Erwin Pesch, Margozata Sterna, and Frank Werner. The two-machine flow-shop problem with weighted late work criterion and common due date. *European Journal of Operational Research*, 165:408–415, 2005.

[4] P. Brucker. *Scheduling algorithms (Fourth edition)*. Springer-Verlag, Heidelberg, Germany, 2004.

[5] J. Du and J. Y.-T. Leung. Minimizing mean flow time in two-machine open shops and flow shops. *Journal of Algorithms*, 14:24–44, 1993.

[6] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1:117–129, 1976.

[7] D. Goldberg and R. Lingle. Alleles, loci and the traveling salesman problem. In *Proceedings of the International Conference on Genetic Algorithms and Their Applications, Pittsburg, USA*, pages 154–159, 1985.

[8] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[9] Richard M. Golden. *Mathematical Methods for Neural Network Analysis and Design*. MIT Press, Cambridge, Massachusetts, 1996.

[10] T. Gonzalez and S. Sahni. Flowshop and jobshop schedules: complexity and approximation. *Operations Research*, 26:36–52, 1978.

[11] John Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge, 1992.

[12] S. M. Johnson. Optimal two-and-three-stage production schedules with set-up times included. *Naval Research Logistic Quaterly*, 1:61–68, 1954.

[13] B. J. Lageweg, J. K. Lenstra, and A. H. G. Rinnoy Kan. A general bounding scheme for the permutation flow shop problem. *Operations Research*, 26, 1977.

[14] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete mathematics*, 1:343–362, 1977.

[15] C.-F. Liaw. An efficient simple metaheuristic for minimizing the makespan in two-machine no-wait job shops. *Computers and Operations Research, In Press, Corrected Proof, Available online February 27, 2007*, 2007.

[16] Melanie Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, Cambridge, Massachusetts, 1996.

[17] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A study of permutation crossover operators on the traveling salesman problem. In J. J. Grefenstette, editor, *Genetic Algorithms and Their Applications: Proceedings*

of the Second International Conference on Genetic Algorithms, Hillsdale, New Jersey, 1987. Lawrence Erlbaum.

[18] A. Oulamara. Makespan minimization in a no-wait flow shop problem with two batching machines. *Computers and Operations Research*, 34:1033–1050, 2007.

[19] Christian Prins. Competitive genetic algorithms for the open-shop scheduling problem. Technical report, Ecole des Mines de Nantes, 1999.

[20] Matthew Wall. *GAlib: A C++ Library of Genetic Algorithm Components*. Cambridge, Massachusetts, version 2.4 edition, 1996. http://lancet.mit.edu/ga/.

[21] Matthew Wall. *A Genetic Algorithm for Resource-Constrained Scheduling*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachussetts, 1996.