Scaling Populations of a Genetic Algorithm for Job Shop Scheduling Problems using MapReduce

Di-Wei Huang
Department of Computer Science
University of Maryland
dwh@cs.umd.edu

Jimmy Lin
iSchool
University of Maryland
jimmylin@umd.edu

ABSTRACT

Inspired by Darwinian evolution, a genetic algorithm (GA) approach is one of the popular heuristic methods for solving hard problems, such as the Job Shop Scheduling Problem (JSSP), which is one of the hardest problems where there lacks efficient exact solutions. It is intuitive that the population size of a GA may greatly affect the quality of the solution, but it is unclear how a large population helps in finding good solutions. In this paper, a GA is implemented to scale the population using MapReduce, a framework running on a cluster of computers that aims to provide large-scale data processing. The experiments are conducted on a cluster of 414 machines, and population sizes up to 10^7 are inspected. It is shown that larger population sizes not only tend to find better solutions, but also require fewer generations. It is clear that when dealing with a hard problem like JSSP, an existing GA can be improved by scaling up populations, whereby MapReduce can handle massive populations efficiently within reasonable time.

Categories and Subject Descriptors

I.2.8 [**Computing Methodologies**]: ARTIFICIAL INTELLIGENCE—*Problem Solving, Control Methods, and Search* [Heuristic methods, Scheduling]; D.1.3 [**Software**]: PROGRAMMING TECHNIQUES— *Concurrent Programming* [Distributed programming]

Keywords

Job shop scheduling, genetic algorithms, MapReduce

1. INTRODUCTION

In solving hard problems where there lacks efficient exact solutions, heuristic approaches, including genetic algorithms (GAs) [5], simulated annealing [1], and tabu search [9], have become popular alternatives. Among them, GAs can be easily parallelized to scale its computing ability because of its intrinsic parallelism, and hence offer great potential toward solving hard problems. GAs represent potential solutions by strings of symbols, or linear chromosome, and simulate the process of natural selection, crossover, and mutation among a population of chromosomes, as inspired by Darwinian evolution. Fitnesses of chromosomes are assessed based on the quality of the solutions they represent, and the fitter chromosomes are given higher probability of survival and reproduction. In this manner, a good solution is likely to be evolved after a number of generations.

The emergence of the MapReduce framework [6] provides new opportunities to empower GAs with the ability to handle large populations (e.g., millions of individuals). Responding to the need to process huge volumes of data over the growing Internet, MapReduce was designed to support parallel large-scale data processing on a cluster of commodity hardware, which is also known as cloud computing. It aims to provide seamless scalability such that as more machines are plugged into the cluster, the computing capability grows almost linearly. As shown in Figure 1, the framework consists of two types of components: the *mappers* and the *reducers*, which execute map and reduce tasks by invoking user-defined map and reduce functions, respectively. Each mapper and reducer can be located on separate machines. In a MapReduce job, each mapper processes a portion of the input data in the form of key-value pairs, where each pair is sent as input to the map function. The map function produces zero or more intermediate key-value pairs. These intermediate data are then shuffled, sorted, and sent to the reducers. How these intermediate data are dispatched is controlled by another user-defined component called the *partitioner*. The reducers process the intermediate data and output key-value pairs as the final results. The intermediate data sent to a reducer are aggregated and sorted by the keys, where the reduce function is called once for every key. As proposed in [18], GAs can be fitted into this framework by computing each generation as a separate MapReduce job. Since MapReduce can handle input data of large sizes (e.g., terabytes), it is possible to encode the population of GAs as the input/output data, and benefit from a large population.

To assess the ability of GAs with large populations, the Job Shop Scheduling Problem (JSSP) is chosen as the target problem to be solved. It has drawn much attention not only for its practical applications in operation research, but also for its computational complexity. The objective is to schedule $|J|$ jobs on $|M|$ machines such that the makespan, i.e., the overall time needed to complete all jobs, is minimized. Each job consists of an ordered list of operations, each of which requires being processed by a certain machine for a certain uninterrupted duration. The ordering of operations represents precedences or dependencies among them. Typically, each job contains $|M|$ operations requiring different machines. Two constraints must be satisfied when scheduling an operation of duration d at time t: (1) all precedent operations are completed before t; (2) no other operations are scheduled to the required machine from t to $t+d$. The Traveling Salesman Problem (TSP), a well known strong NP-complete problem, is a special case of JSSP [16].

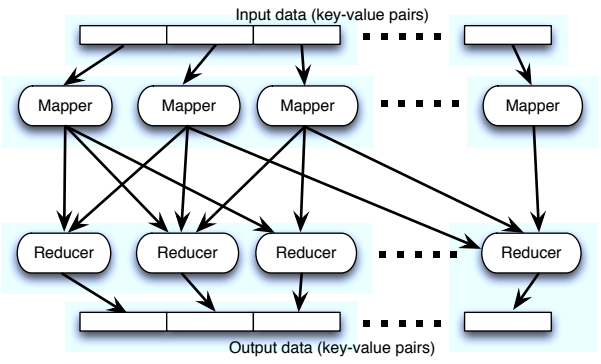

Figure 1: The architecture of the MapReduce framework

Therefore, JSSP is much harder than TSP and is among the hardest combinatorial optimization problems. Since there are no efficient exact solutions to date, a heuristic approach is needed.

In this study, a GA with massive populations for solving JSSP is implemented using MapReduce. The GA is non-trivial in that it includes encoding/decoding chromosomes, building schedules, performing local searches, handling tournament-base selections, and processing non-random crossover. As it has been suggested by theoretical research [7, 19] that GAs with large population sizes are advantageous in solving hard problems, our GA for JSSP is given massive populations and run on a cluster of 414 machines. The experimental results show the effects of having massive populations, and confirm that large populations indeed help in finding good solution. Another experiment is done to show the execution time decreases as the size of a cluster grows.

2. ALGORITHMS

This section describes in detail the algorithms used in this paper and how they are implemented with MapReduce.

2.1 Representation

The operation-based representation is adopted [4, 3] to encode and decode the chromosomes. Consider a set of jobs $J = \{0, 1, 2, ...\}$, where each job $j \in J$ contains N_j operations ($j = 0, 1, ..., |J|-1$). A chromosome contains $\sum_{j \in J} N_j$ genes that are job names (i.e., members of J), where each job j appears exactly N_j times. The job name appearing at each gene represents an operation belongs to the job, where the actual operation is determined by the order of occurrence of that job name, i.e., the kth occurrence of job j represents the kth operation of j. For example, with $J = 2$ and $N_0 = N_1 = 3$, a chromosome may look like:

$$[0, 0, 1, 1, 0, 1]$$

where the first operation of job 0 comes first, followed by the second operation of job 0, followed by the first operation of job 1, and so on. Notice that any permutations of the genes will always yield valid schedules if the operations are added to the schedule in the order of their appearance in the chromosome.

The data structure used to store an individual of the GA is shown in Figure 2. This key-value structure is used as the input, the intermediate, and the output data of MapReduce. The key part contains an ID $\in [0, 1)$ assigned to each individual uniformly at random, and the value part contains a makespan value, a generation value, and the chromosome. The makespan value stores the length of the schedule implied by the chromosome, that is, the length of time between the execution of the earliest operation and the completion of the latest one. The fitness can be evaluated directly through the makespan value, where a lower makespan value makes a fitter individual. The generation value facilitates tracing of evolution by storing the number of generation descended from the original population. Finally, the chromosome is stored as an array of integers.

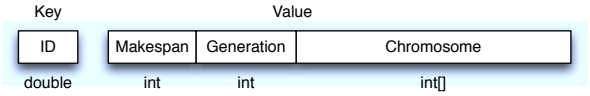

Figure 2: The key-value structure to store an individual

2.2 The genetic algorithm

The main GA is implemented in the MapReduce framework where each generation of the GA is done by a MapReduce job. The data structure shown in Figure 2 is used as the input and output key-value pairs of both the map and reduce phase. The map phase evaluates the fitnesses of the population, where the schedules are built according to the chromosome, and local search is performed to find the makespans. The fittest individual is recorded. The partitioner dispatches the resulting individuals to random reducers by referring to their randomly generated IDs. The reduce phase processes selection and crossover, and produces a new generation population as the output. After the new population has been generated, a new MapReduce job is created for this new generation. This process is continued until a satisfactory solution, if not optimal, to the JSSP is found.

2.2.1 *Mapper: Fitness Evaluation*

The algorithm for the map phase is shown in Algorithm 1, which aims at evaluating the fitnesses (i.e., the makespans) of an individual. The map function is invoked once per individual in parallel on multiple mappers. The mapper first obtains an ordered list of operations by decoding the chromosome. A schedule is then built based on the list, where each job in the list is placed on the schedule at the earliest possible time. A local search is performed to fine-tune the resulting schedule [14]. First, the critical path of the schedule is identified as blocks of continuous operations. For each block, if swapping the first two operations or the last two operations yields a shorter makespan then accept it, otherwise undo the swapping. Notice that swapping the first two operations or the last two operations will not improve the schedule, and thus can be omitted. Once a new schedule is obtained by swapping operations, local search is performed again on the new schedule until no improvement can be made. The mapper then outputs the individual with the makespan updated.

In addition to evaluating and outputting individuals, each mapper keeps track of the best individual it has seen. At the end of the mapper's lifecycle, the best individual is emitted with the ID set to a special value null.

Algorithm 1 The map function of a single generation of the GA. An ID is required as the key, and an individual is required as the value, as shown in Figure 2.

```
function map(key, value)

begin
  opList ← decode(value.chromosome);
  for each operation op in opList do
      comment: add op to schedule at the earliest available spot
      schedule.add(op); od
  schedule.local_search()
  value.makespan ← schedule.getMakespan();
  output(key, value);
  if best.makespan > value.makespan
    then best ← value; fi
end
finalization
output(null, best);
```

2.2.2 Partitioner

The partitioner assigns the individuals emitted from the mappers to the reducers according to the IDs, as characterized by Equation 1, where $h(\cdot)$ is a hash function and r is the number of reducers. The null IDs are always sent to the first reducer (i.e., reducer #0). The first reducer is therefore responsible for comparing the best individual from different mappers, and determining the best of the best individual across the whole population. Otherwise, normal IDs are used as input to a hash function to determine which reducer to send to. Since IDs are generated at random, each individual is sent to a random reducer.

$$reducer \leftarrow \begin{cases} 0, & \text{if } ID = \text{null} \\ h(ID)\%r, & \text{otherwise} \end{cases} \quad (1)$$

2.2.3 Reducer: Selection and Reproduction

The algorithm for the reduce phase is shown in Algorithm 2, which selects good individuals and produces descendants by crossing over their chromosomes. The reduce function is invoked once per individual ID in parallel on multiple reducers. The first reducer, i.e., the reducer #0, which receives the best individuals from each mapper, records the best among them. This is the best solution found in this generation of the GA. In the following, an approximation of tournament selection is adopted, in which s individuals are chosen randomly from the population and the fittest one among them is selected for crossover. In this study, s is set to 5 empirically. Since the individuals are sent to the reducers at random, and their IDs by which the reducers sort them are also random, their order in the input sequence to a reducer is arbitrary and without regard to the fitnesses. It is then reasonable to use a sliding window (indicated by the variable *window* in Algorithm 2) of size s, go through the input sequence of key-value pairs, and select the fittest one within the window, to approximate the random choices of s individuals in the tournament selection. Notice that since the window has to wrap around when it reaches the end of the input sequence, the first s individuals have to be buffered (indicated by the variable $firstWindow$) for processing after the reducer has seen all individuals.

When the winner of the tournament-based selection is determined, it is used in the reproduction and crossover procedure to generate a new descendant. That is, the chromosomes of the current and the previous winners are taken as the first and the second parents in the crossover, respectively. To preserve characteristics of the parents, a crossover that maintains partially temporal relations among operations (i.e., genes) is needed. One of the crossovers proposed in [15] is adopted (i.e., the "crossover 4"). The chromosomes of the first and the second parent are decoded as two lists (denoted as L_1 and L_2, respectively) of operations, and a continuous portion L_1' of L_1 is chosen at random. A new individual, *kid*, is created with a random ID and a list (denoted as L) of operations, which is initially identical to L_2. L_1' is then inserted to L at the same starting position it appears in L_1, followed by a sweep through L to remove operations contributed by L_2 that exist in L_1'. Finally, the chromosome of *kid* is updated to encode L.

Mutation with small probability is performed after the crossover. Three positions of distinct symbols are randomly selected from *kid*'s chromosome, and one of the six permutations among them is applied uniformly at random. As mentioned in [12], the importance of mutation recedes as the population grows. Since we are more concerned with large population sizes, the probability of mutation is set to a small value of 1%.

Algorithm 2 The reduce function of a single generation of the GA.

```
function reduce(key, values)

initialization
count ← 0; s ← 5;
begin
  if key = null then best ← arg min_{v∈values} v.makespan;
                     print(best); return; fi
  for each value in values do
      if count < s
        then window[count] ← value;
             firstWindow[count] ← value;
        else window[count % s] ← value;
             reproduction(); fi
      count ← count + 1; od
where
proc reproduction() ≡
  prevWinner ← winner;
  winner ← arg min_{i∈window} i.makespan;
  kid.chromosome ← crossover(prevWinner, winner);
  if random() < 0.01 then kid.mutate(); fi
  kid.makespan ← −1;
  kid.generation ← winner.generation + 1;
  output(random(), kid); .
end
finalization
for i ← 0 to s − 1 do
    window[(count + i) % s] ← firstWindow[i];
    reproduction(); od
```

2.3 Initialization

Initialization of the population is done by a separate MapReduce job without reducers. Although many of the previous studies use random initial populations, they may require more generations to find a good solution. This increases the overhead of MapReduce, because each MapReduce job running a generation requires a certain amount of time to initiate the mappers and the reducers, and to shuffle and sort the intermediate data over the network. For this reason, a good initial population is generated as suggested in [8, 15], which is outlined in Algorithm 3. The individuals generated in this manner always yield active schedules, in which no operation can be scheduled earlier without delaying some other operations or breaking a precedence constraint. The optimal solution of JSSP is always an active schedule.

Algorithm 3 The map function to generate initial population of size N.

function map(key, $value$)

J: the set of all jobs
N: the target size of population
$numOp$: the total number of operations
begin
  for $i \leftarrow 1$ to N do
    $schedule$.clear();
    $kid.chromosome \leftarrow \{\}$;
    $kid.generation \leftarrow 0$;
    comment: C: the set of schedulable operations
    $C \leftarrow$ {the 1st operation of job $j, \forall j \in J$};
    comment: $op.est$: the earliest schedulable time for op
    $op.est \leftarrow 0, \forall op \in C$;
    for $k \leftarrow 1$ to $numOp$ do
      $p \leftarrow \arg\min_{op \in C}\{op.est + op.processingTime\}$;
      $G \leftarrow \{op \in C$ s.t. $op.machine = p.machine$,
       and $op.est < p.est + p.processingTime\}$;
      $q \leftarrow G$.randomElement();
      $schedule$.add(q);
      $kid.chromosome = kid.chromosome + q$;
      C.remove(q);
      C.add(q.nextOperationInJob());
      update $op.est$ according to $schedule, \forall op \in C$; od
    output(random(), kid); od
end

3. EXPERIMENTS

The JSSPs listed in Table 1 are tested. This problem set can be obtained from the OR-library [2]. These problems are by no means an exhaustive list of all available problems, but they are chosen to represent various difficulty levels, and because their optimal solutions are known. FT10 and FT20 were first proposed by [13] and have become standard benchmark problems. LA40 [11], a somewhat tricky problem, is concerned with scheduling 15 jobs on 15 machines. The hardest problem, SWV14 [17], consists of 50 jobs where intensive contention for machines can be expected. This study does not put emphasis on proposing innovative algorithms or on outperforming other solutions to JSSP, but shows the effects of a GA running large populations in parallel, as a potential enhancement to existing solutions. Two experiments are conducted. The first experiment shows how population sizes affect the GA in approaching a good solution; the second one shows how the running time can be reduced by scaling the size of the cluster.

Table 1: Profiles of JSSP instances

Name	#Jobs	#Machines	Optimal Makespan
FT10	10	10	930
FT20	20	5	1165
LA40	15	15	1222
SWV14	50	10	2968

3.1 Effects of the Population Size

The first experiment was run on a cluster provided by Google and managed by IBM, shared among a few universities as part of NSF's CLuE (Cluster Exploratory) Program and the Google/IBM Academic Cloud Computing Initiative. The cluster used in our experiments contained 414 physical nodes; each node has two single-core processors (2.8 GHz), 4 GB memory, and two 400 GB hard drives. The entire software stack (down to the operating system) was virtualized; each physical node runs one virtual machine hosting Linux. Experiments used Java 1.6 and Hadoop version 0.20.1. Population sizes $p = 10^5$, 10^6, and 10^7 were run with 1000 mappers and 100 reducers.

The results are shown in Figure 3. As the population size increases, fewer generations are required to converge. Particularly in Figure 3(a), only 4 generations are required to reach the optimal makespan when $p = 10^7$, while 18 and 26 generations are required when $p = 10^6$ and 10^5, respectively. The same observation can be made in Figure 3(b), where 19 generations are required to reach the optimal makespan when $p = 10^7$, while 25 are required by $p = 10^6$. In Figure 3(c), although both experiments with $p = 10^5$ and 10^6 converge at the same local minimum, the latter approaches it with fewer generations. Since MapReduce incurs overhead for every generation, it is desirable to find solutions with fewer generations. This can be achieved by using a larger population as shown in the results.

In addition, GAs with larger populations are more likely to find good solutions. In Figure 3(b), the experiment with $p = 10^5$ converges at a local minimum 1178, while the ones with larger population sizes yield the optimal makespan of 1165. Similarly, in Figure 3(c), both experiments with $p = 10^5$ and 10^6 converge at 1252, while a better makespan 1233 is found by scaling the population size to 10^7. In Figure 3(d), however, the effects of increasing population sizes are not phenomenal. The reason may be that this problem is too hard to be solved within a few tens of generations, but more generations incur more overhead in MapReduce. More experiments with $p > 10^7$ on a larger cluster have to be done to further investigate this issue.

3.2 Effects of the Cluster Size

This experiment runs the GA on Amazon's Elastic Compute Cloud (EC2) clusters of different sizes, and the completion time for each generation is observed. The GA is given a population of 10^4 individuals to solve LA40. For each cluster size, the GA is run for 10 generations, and the average execution times and the standard deviations are plotted in Figure 4. The execution time for a cluster of one machine is

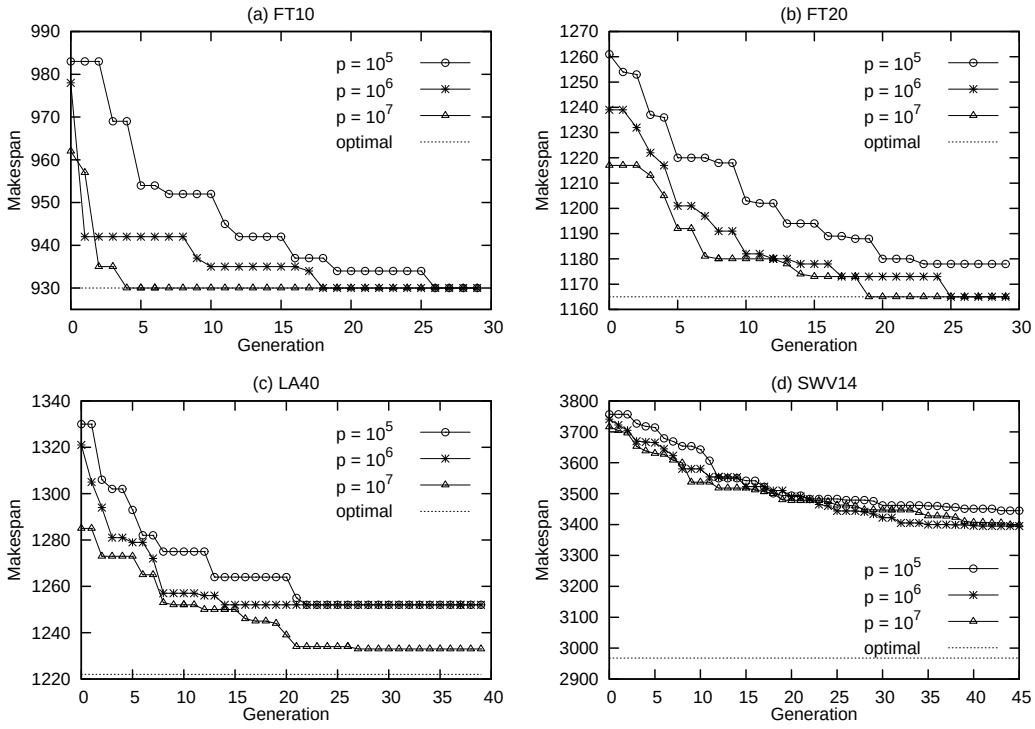

Figure 3: The results of GA with various population sizes p for the problems (a) FT10, (b) FT20, (c) LA40, and (d) SWV14

shown as a baseline for comparison with other clusters with multiple machines. As the number of machine instances in the cluster increases, the running time decreases as a result of increasing computing power. It is therefore beneficial to increase the cluster size when running GAs with large population sizes.

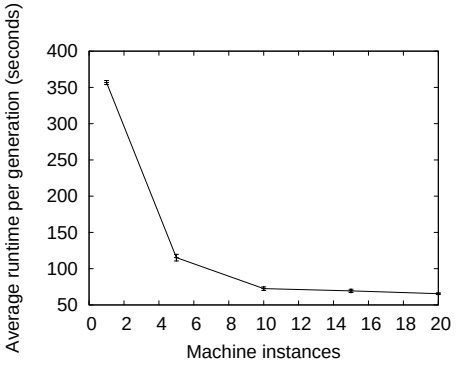

Figure 4: The effects of cluster size

For comparison, a typical profile of execution time for one generation is listed in Table 2. The GA is given a population of size 10^4 to solve LA40. Only 1 mapper and 1 reducer are used. The total job completion time and the cumulative running time for the map and the reduce functions are recorded. Most of the execution time (i.e., $\approx 95\%$) is spent on running the map and the reduce functions, while the remaining portion ($\approx 5\%$) of the time is labeled as "overhead".

Table 2: A typical profile of execution time when running the GA for one generation to solve LA40, using one mapper and one reducer. The population size is 10^4.

Job Completion Time	map()	reduce()	Overhead
356.067 (seconds)	331.155	7.191	17.721
100%	93.00%	2.02%	4.98%

4. CONCLUSION

In this study, a GA for JSSP is implemented using MapReduce, and experiments are run with various population sizes (i.e., up to 10^7) and on clusters of various sizes. Our implementation of GA with MapReduce is based on [18], while adding more GA features to cope with real-world problems, including local search, non-random crossover, and non-random initial populations. The chromosome representation and the schedule evaluation for JSSP also increase the complexity.

The effects of large populations are prominent, in that a larger population tends not only to find a better solution, but also to converge with fewer generations. The results confirm what was mentioned in [10, p. 198-200], but our experiments consist of a much harder problem and much larger populations. Moreover, having fewer generations is beneficial regarding the overall MapReduce overhead. Because for each run of MapReduce there exists certain initialization/shuffling overhead, having fewer generation, and hence

fewer iterations of MapReduce, reduces the overall overhead. The effects of cluster sizes is also presented, which show the speedup of execution time by increasing nodes in the cluster. This may serve as a rough guideline regarding what cluster size to use and what speedup to expect.

In general, GAs with massive populations provide a new possibility toward solving hard problems, and this can be achieved by using MapReduce running on a cluster of commodity hardware.

Acknowledgements

This work was supported in part by Google and IBM, via the Academic Cloud Computing Initiative (ACCI).

5. REFERENCES

[1] E. Aarts and J. Korst. *Simulated annealing and Boltzmann machines*. John Wiley & Sons New York, 1989.

[2] J. Beasley. OR-Library: Distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

[3] C. Bierwirth, D. Mattfeld, and H. Kopfer. On permutation representations for scheduling problems. *Parallel Problem Solving from Nature—PPSN IV*, pages 310–318, 1996.

[4] R. Cheng, M. Gen, and Y. Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms–I. Representation. *Computers and Industrial Engineering*, 30(4):983–997, 1996.

[5] L. Davis et al. *Handbook of genetic algorithms*. Citeseer, 1991.

[6] J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[7] S. Droste, T. Jansen, and I. Wegener. Upper and lower bounds for randomized search heuristics in black-box optimization. *Theory of Computing Systems*, 39(4):525–544, 2006.

[8] B. Giffler and G. Thompson. Algorithms for solving production-scheduling problems. *Operations Research*, 8(4):487–503, 1960.

[9] F. Glover and B. Melián. Tabu search. *Metaheuristic Procedures for Training Neutral Networks*, 36:53–69, 2006.

[10] J. Koza. *Genetic programming: on the programming of computers by means of natural selection*. The MIT press, 1992.

[11] S. Lawrence. Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (Supplement). *Ph.D. Thesis Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA*, 1984.

[12] S. Luke and L. Spector. A comparison of crossover and mutation in genetic programming. *Genetic Programming*, 97:240–248, 1997.

[13] J. Muth and G. Thompson. *Industrial scheduling*. Prentice-Hall, 1963.

[14] E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop problem. *Management Science*, 42(6):797–813, 1996.

[15] B. Park, H. Choi, and H. Kim. A hybrid genetic algorithm for the job shop scheduling problems. *Computers & industrial engineering*, 45(4):597–613, 2003.

[16] S. Reddi and C. Ramamoorthy. On the flow-shop sequencing problem with no wait in process. *Operational Research Quarterly*, 23(3):323–331, 1972.

[17] R. H. Storer, S. D. Wu, and R. Vaccari. New search spaces for sequencing problems with application to job shop scheduling. *Management Science*, 38(10):1495–1509, October 1992.

[18] A. Verma, X. Llorà, D. Goldberg, and R. Campbell. Scaling genetic algorithms using MapReduce. *Proceedings of the 2009 Ninth International Conference on Intelligent Systems Design and Applications*, pages 13–18, 2009.

[19] C. Witt. Population size versus runtime of a simple evolutionary algorithm. *Theoretical Computer Science*, 403(1):104–120, 2008.