This is to certify that the
dissertation entitled

AN ADAPTIVE REPRESENTATION FOR A GENETIC ALGORITHM IN SOLVING FLEXIBLE JOB-SHOP SCHEDULING AND RESCHEDULING PROBLEMS

presented by

Prakarn Unachak

has been accepted towards fulfillment
of the requirements for the

Ph.D. degree in Computer Science

Major Professor's Signature

August 24, 2010

Date

MSU is an Affirmative Action/Equal Opportunity Employer

AN ADAPTIVE REPRESENTATION FOR A GENETIC ALGORITHM IN SOLVING FLEXIBLE JOB-SHOP SCHEDULING AND RESCHEDULING PROBLEMS

By

Prakarn Unachak

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILSOPHY

Computer Science

2010

ABSTRACT

AN ADAPTIVE REPRESENTATION FOR A GENETIC ALGORITHM IN SOLVING FLEXIBLE JOB-SHOP SCHEDULING AND RESCHEDULING PROBLEMS

By

Prakarn Unachak

In a modern manufacturing system, it is imperative that production go on efficiently. A good scheduler must allocate resources to processes with minimum waste while fulfilling all constraints of the scheduling environment. The Job Shop Scheduling Problem (JSSP) is among the most popular scheduling problems. The Flexible Job Shop Scheduling Problem (FJSP) relaxes the restrictive machine assignment of JSSP, moving it closer to a real-world application. However, it is still far from a real-world manufacturing environment, in which disruptions such as machine failure must be taken into account.

The goal of this dissertation is to create a Genetic Algorithm (GA) approach to FJSP that can adapt to disruption to reflect more closely the real-world manufacturing environment. We hope that by using just-in-time machine assignment and adapting scheduling rules, we can achieve the robustness and flexibility we desire.

The adaptive representation (AdRep) was tested in both a static environment and a disruption-prone dynamic environment. In the static environment, benchmark problems and published results were compared with the result of our approach to test its utility there. Although not as scalable as some approaches usable only for static cases, our approach discovered all the best-so-far published results on a series of commonly used

benchmark problems in a strong way—it consistently produced, in almost every run, all of the points on the Pareto front produced when FJSP is formulated as a multi-objective problem, whereas most of the other approaches maximized only a single one of the objectives. Then, in the dynamic model (i.e., in which machines break down), we compared our adaptive method to two benchmark algorithms: a right-shifting rescheduler and a prescheduler. A right-shifting rescheduler repairs schedules by delaying affected operations until the disruption is over. A prescheduler works on each disruption scenario separately, treating disruptions like prescheduled downtime. Experiments showed that our approach was able to adapt to disruptions in a manner that minimized lost time.

To my family

ACKNOWLEDGEMENT

First, I would like to thank my advisor, Dr. Erik D. Goodman, who has provided me all these years with advice and support. I am very grateful for many years he has guided me to this day. I would not have finished this dissertation without him.

Furthermore, I would like to thank the other members of my committee, Dr. Charles A. Ofria, Dr. William F. Punch, and Dr. Eric Torng. I would like to thank Dr. Punch especially for introducing me to Genetic Algorithms.

I also would like to thank my friends. You have made this stranger at home far away from his home.

Finally, I would like to thank my family, especially my parents. Without their love, support and understanding, I would not be able to complete this study. I would especially like to thank my parents for their unwavering support and exceptional patience. The long years have come to pass.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Problem Statement

A J × M Flexible Job Shop Scheduling Problem (FJSP) instance consists of a number J of jobs in a manufacturing environment of M machines. Each job has a process order that specifies an ordering in which the operations must be performed (precedence constraints). There is at least one machine (and sometimes, perhaps several) that can process each operation, sometimes at differing costs. One machine can process only one operation at a time (resource constraint), and it will continue processing that operation until it is finished (non-preemption constraint). The goal is to construct a schedule, by first assigning each operation a machine, then allocating runtime for it, so as to optimize the objective function(s), such as makespan—time from when the first operation starts to when the last operation finishes—while satisfying all the constraints. The environment can be *static* or *dynamic*. In a static environment, once the scheduling starts, there is no change in the system. (no machine breakdowns or arrival/cancellation of jobs). A schedule can be expected to proceed unimpeded to its planned outcome. On the other hand, in a dynamic environment, disruptions such as machine breakdowns or new job arrivals can occur, either deterministically or stochastically. A good scheduler should be able to handle such disruptions in a way that the repaired solutions are typically not unnecessarily worse than the original solutions.

In this dissertation, the focus is on 1) minimizing three objectives: *makespan (C_M), critical machine workload (W_M), and total machine workload (W_T)* in static environment, (whereas many heuristics seek to minimize only a single one of them), and 2) minimizing makespan and *repaired schedule makespan (C_{RMax})* in a dynamic environment with disruption due to machine unavailability (unscheduled breakdown).

1.2 Goals of Research

1) Design a flexible representation for FJSP.

   Currently, few approaches to FJSP seem to focus on the flexibility of the machine selection, also known as routing, process. By delaying the routing decision until absolutely necessary and providing simple heuristics to assist the routing process, while employing adaptive scheduling to sequence the operations based on the states of the environment, we hope to exploit the states of the system to achieve more stable solutions (solutions less impaired by machine failures and other disruptions).

2) Present a novel way of FJSP rescheduling.

   So far, there have been few attempts to expand rescheduling into FJSP. Since our proposed approach does not bind an operation to a machine, when a disruption occurs, rendering the machine unavailable, routing rules can reassign any affected operation to another machine. This can often be beneficial, especially when the disruption period is especially long, rendering right-shifting less useful.

3) Validate the adaptive representation in the static model.

There are quite a few papers on FJSP. Here, we will test the new adaptive representation against the current best published results on a number of benchmark problems. The goal is to show that there is no loss in static performance, or, if noticeable degradation is found, that improvements can be made. (The real advantage of the representation is to be its performance on the dynamic FJSP, rather than improvement on the already-well-optimized static case.)

4) Validate the adaptive representation in disruption-prone model.

To date, there has not been much work on the dynamic FJSP, especially against disruptions such as machine unavailability. The AdRep representation will be tested in a dynamic environment that is prone to machine unavailability disruption. The goal is to show that the AdRep can produce solutions that suffer minimally from disruptions, or at least in which the losses are small relative to those suffered by other scheduling approaches.

5) Performance Analysis of AdRep.

While validating the AdRep on published problem instances, we hope to measure its efficiency in solving the problems. Although it is good to create a good schedule builder that can adapt well to disruption, it is better to do so with reasonable cost. We have sought to learn how efficient AdRep is, and, when possible, have implemented improvements in its efficiency.

1.3 *Overview of the Dissertation*

This dissertation is organized as follows: Chapter 2 discusses the background of this dissertation, first on Genetic Algorithm, the evolutionary computation approaches used in this dissertation. Then the focus problem of the dissertation—FJSP and its parent problem, JSSP—are discussed. Then, we focus on previous approaches utilized to solve these two problems. Chapter 3 discusses the architecture of a proposed GA system, the Adaptive Representation (AdRep) and how it can be used to solve FJSP instances, in both the static and the disruption-prone dynamic cases. Chapter 4 describes the performance analysis of AdRep on the static FJSP. Chapter 5 is a performance analysis study of AdRep on the disruption-prone dynamic FJSP, comparing computational results to those of benchmark algorithms. Chapter 6 concludes the dissertation and suggests future directions for the research.

CHAPTER 2

BACKGROUND

2.1 Genetic Algorithm (GA)

A genetic algorithm (GA) is an evolutionary computation approach derived from Darwin's principle of survival of the fittest and the idea of sexual reproduction. They are well described in Goldberg's landmark 1989 book [1], as summarized next. Multiple individuals, representing solutions or instructions for constructing solutions (indirect representations), coexist in a virtual environment in which they will be evaluated. The objective function (makespan, for example) is used to determine which solutions are "fitter" than others. Fitter individuals have more chance to propagate their characteristics to the next generation. A *recombination function* combines the characteristics of two individuals to produce new solutions, which carry some mixture of their parents' characteristics. Sometimes the recombination produces more fit individuals, but frequently the offspring are less fit than their parents. To maintain the diversity of the population of solutions being explored, a *mutation function* makes changes to individuals to produce (genotypically) different individuals. This is one of the most popular approaches to JSSP.

In order to use a GA, or most EC frameworks, to solve a problem, we need a method to encode schedules (solutions of the problem) as *chromosomes*, as the individuals in a GA system are often called. Such an encoding is called *representation*. Choice of representation will affect other parts of the GA system. For FJSP, a solution can be a

direct representation, in which the representation is the schedule itself, or it can be an *indirect representation*, in which the representation constitutes a way to build a schedule, not the schedule itself. For example, in the FJSP context, a chromosome including direct machine assignments and an array of starting times for each operation is a direct representation, while if the an array of starting times were to be replaced by an array of priorities, the chromosome would become an indirect representation, since some "external" scheduling mechanism will be needed to build a schedule out of the priorities provided in the chromosome.

After we have a representation of the problem, we can start the GA system. The GA runs in steps, as shown in Figure 2.1 below:

1. *Initialization*. Initial population is randomly or stochastically generated.
2. *Evaluation*. Each individual's fitness is evaluated.
3. Population of the next generation is generated by
   3.1. *Selection*. With probability proportional to their fitness, two individuals are selected.
   3.2. *Recombination* or *Crossover*. Two selected individuals are recombined to produce offspring.
   3.3. *Mutation*. Small changes are applied to offspring. They are then put into the new population.
   3.4. Repeat until new population is full.
4. Evaluation fitness of individuals in the new population.
5. If *termination criterion* is met, results are print and the search terminates. Otherwise, assign new population as current population and go back to step 3.

Figure 2.1. Steps in a classical genetic algorithm

Initialization is usually random. The key is to have diversity in the population. However, pregenerated individuals, sometimes called *seeds*, or specialized sampling algorithms can also be introduced into the initial population if certain characteristics are

desired. Also, as often in scheduling problem, dispatching rules can be used to create the initial population. Approach to Localization (AL) [2] and Dispatching Rules approach can also be used to enhance the initial population.

Evaluation is done using a *fitness function* based on the search goals. However, by using raw fitness values, it is possible that one or a few individuals quickly dominate the population, leading to premature convergence of the population. Hence, fitness values may need to be scaled, or scale-invariant fitness methods such as tournament selection must be employed. The need for scaling does depend on the selection method employed, however. Many selection methods, such as tournament selection, do not need scaled fitnesses.

A **crossover operator** combines features from two parents and produces offspring that inherit parents' alleles. A **mutation operator** applies changes to the offspring, introducing some diversity into the population. Appropriate choice of both types of operators is dependent on the problem at hand and the representation used. Note that whether crossover operators and mutation operators produce valid results or not also depends on the representation used. For example, if a permutation-type chromosome is being used, a single-point crossover operator will not produce valid offspring. In some cases, a **repair mechanism** is required to convert an invalid chromosome into a valid one.

Selection assigns probability of being selected to become a parent to an individual. The probabilities can be proportional to fitness (proportional selection, or roulette-wheel selection), evenly distributed among individuals, or even based on ranking among individuals. Alternative selection schemes, such as randomly choosing k individuals and selecting the fittest one (also called *tournament selection*), can also be used.

Replacement dictates how the population of the next generation is derived. Replacement can either be generational, where no parents directly survive, or steady-state, where parents and offspring coexist and replacement decisions are made as each offspring is created. Sometimes we would want to preserve the fittest individual. In such cases, we use *elitism*, a policy that allows preserving the k fittest individuals automatically into the next generation. Particularly for multiple-objective searches, there are other replacement methods, such as the one used in NSGA-II [3], where individuals are successively sorted into non-dominated sets and selected for survival, then removed from the further sorting.

A **Termination Criterion** is used by a GA to decide when to quit. It can be a specified number of generations, a target fitness value, one of many diversity criteria, or the number of times the evaluation function has been called, for example.

2.2 *Flexible Job-Shop Scheduling Problem (FJSP)*

First introduced by Brucker and Schlie in 1990 [4], where a polynomial algorithm was developed to solve a 2-job problem, a $J \times M$ Flexible Job-shop Scheduling Problem consists of J number of jobs and M number of machines. Each job $i \mid i = 1..J$ has a sequence of operations $\left(o_{i1}, o_{i2}, \ldots, o_{in_i}\right)$, where n_i is the number of operations in job i. Each operation has at least one machine that can process it, at a cost that may vary from machine to machine. The goal of the problem is to schedule each operation on some machine, subject to all constraints, and to extremize some additional measures.

2.1.1 Definition

A more formal definition of FJSP is follows:

A $n \times m$ FJSP consists of:

- $J = \{J_i\}_{1 \le i \le n}$ is the set of jobs to be scheduled.
- Each job J_i is a set of operations $\{ o_{j1}, o_{j2}, \ldots, o_{jn_i} \}$ in a predetermined sequence, where o_{jx} is the x^{th} operation of job j.
- $M = \{M_k\}_{1 \le k \le m}$ is the set of machines available.
- An operation o_{jx} can be processed on machine $m_k \subseteq M$ for the time unit cost of $t_{j,x,k}$.

- *Precedence constraint*: Let $t_s(o_{jx})$ be the start time of processing o_{jx} and $t_f(o_{jx})$ be the finishing time of processing o_{jx}. Then $t_f(o_{jx}) \leq t_s(o_{jx+1})$. That is, in order to start its job successor o_{jx+1}. o_{jx} must finish first.

- *Resource constraint:* Let $m(o_{jx})$ be the machine on which o_{jx} will be processed. Then $(m(o_1) = m(o_2)) \rightarrow (t_f(o_2) \leq t_s(o_1) \mid t_f(o_1) \leq t_s(o_2))$. That is, a machine can only process one operation at a time.

- *Non-preemption constraint*: an operation cannot be pre-empted out of a machine: once it starts, it will be processed there until finished (i.e., no other operation can take over that machine until the one already started has finished).

If all operations can be processed by any machine (although with possibly different time costs), the FJSP instance is considered to have *total flexibility*. FJSP instances that belong to this category can also be called *T-FJSP*. However, if some operations can only be processed by a proper subset of all machines available, the problem instance has only *partial flexibility*. An FJSP instance of this type is sometime called *P-FJSP*.

2.1.2 Example of FJSP instance

Table 2.1 contains an example of a 3×3 FJSP instance. Each row represents an operation, where each table entry $t(O_{j,x}, M_m)$ is the runtime cost for machine M_m to process the operation $O_{j,x}$.

Table 2.1. An example of 3×3 FJSP

Job		Machine		
		M1	M2	M3
1	O1,1	5	6	1
	O1,2	3	6	2
	O1,3	10	6	9
2	O2,1	7	8	7
	O2,2	7	9	5
	O2,3	4	6	5
3	O3,1	4	6	5
	O3,2	6	4	5
	O3,3	3	9	6

In this example, the second operation of job 1, O, would take 6 time units to be processed by machine 2, but only 2 time units on machine 3.

When every operation is assigned a machine and a starting time, we have a solution: a schedule. A schedule is a valid one if it follows all the constraints specified by the FJSP instance. That is, operations in each job do not overlap and follow the precedence ordering (precedence constraint), every machine processes one operation at a time (resource constraint), and every operation is processed by a machine from start to end (non-preemption constraint). Figure 2.2 is a Gantt chart of a schedule example from solving the FJSP instance in Table 2.1.

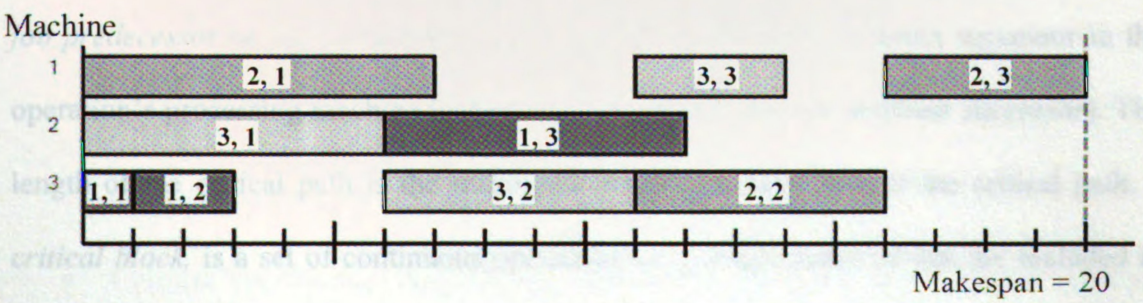

Figure 2.2. A schedule solution of the problem instance from Table 2.1

2.1.3 Job-Shop Scheduling Problem and Variants

The FJSP is a variant of a Job-shop Scheduling Problem (JSSP). In the JSSP, each operation needs to be processed by one specified machine. That is, the routing has been prespecified (no routing decision is needed). Various approaches to the JSSP have been surveyed in [5] and [6]. The JSSP is known to be NP-hard [7], making exhaustive search intractable in the general case. If all the routing decisions are made, an FJSP instance becomes a JSSP instance.

Many scheduling concepts and techniques employed to solve JSSP instances can be useful in solving FJSP instances as well. One such concept is the *critical path*. Derived from the longest path when a JSSP schedule is represented with a disjunctive graph, a critical path is a set of operations on a schedule, finishing at C_M, such that any delay in beginning any of those operations will cause a delay in completion of the final task at C_M. Any operation on the schedule that can be delayed by any amount without causing a delay in completion of the final operation is not on the critical path. From the schedule depicted on Figure 2.2, the critical path of that schedule is {(3, 1), (3, 2), (2, 2), (2, 3)}. In the critical path, an operation's predecessor or successor in the path will either be (1) a direct predecessor or direct successor in the operation's job precedence ordering (called *job predecessor* or *job successor*), or (2) a direct predecessor or direct successor in the operation's processing machine (called *machine predecessor* or *machine successor*). The length of the critical path is the makespan of the schedule. Part of the critical path, a *critical block,* is a set of continuous operations on a single machine that are included in the critical path.

Since the FJSP and JSSP share scheduling tasks, JSSP solution techniques also apply to the FJSP.

2.1.4 Objectives in Solving FJSP

The main objective of solving an FJSP instance is usually minimizing *makespan* (C_M). Makespan is the time measured from when the first operation in the problem instance starts its processing to when the last operation finishes its processing.

Or, let C_j be the completion time of job j. Makespan is:

$$C_M = \max_{1 \le j \le J} (C_j)$$

However, other objectives do exist, such as:

- **Tardiness.** If deadlines for jobs are given, tardiness can be used as an objective. If the completion time is later than the deadline, tardiness is the time period between the completion time and the deadline. If the completion time is earlier than or at the deadline, tardiness is 0.
- **Flow time.** Flow time is the time period between when a job is available to be scheduled and the job completion time. If all jobs are available at the beginning, flow time of a job is its completion time.
- **Total Workload.** (W_T) Total workload is the sum of all utilized time (workload) of all machines in the system.
- **Critical Workload.** (W_M) Critical workload is the maximum machine workload among all machines in the system. Critical workload cannot be greater than makespan.

W_T and W_M are usually not used as single objectives in scheduling. Instead, they are typically used in multiobjective optimization, together with makespan.

To solve an FJSP instance, there are two types of task that must be performed: *routing* and *scheduling*. Routing, sometime called machine assignment, is the task of assigning a machine to each operation. Scheduling, sometime called *sequencing*, is the task of setting each operation's start time, creating a valid schedule that will satisfy all constraints.

2.3 *Previous Approaches*

2.3.1 In Routing

There are two frameworks for combining routing and scheduling together: (1) a *concurrent* or *integrated approach*, where routing and scheduling are performed at the same time, and (2) a *hierarchical approach*, where all the routing decisions are handled first, then the scheduler processes what is now a JSSP instance.

There are a few ways to represent routing in a solution. First is a *fixed routing*, where machine assignment is hard-coded into the solution. This is by far the most popular way of representing routing. Care must be taken, however, in case of partially flexible FJSP instances, that an operation not be assigned to a machine that cannot process it. For example, Ho and Tay's approaches in [8] [9], and [10] utilized a bit string representation, with just enough bits per operation that there is one bit for each machine that is valid for that operation, preventing mis-assignment. Although not encoding the machine number directly, Xia & Wu [11] provided a sorted machine list based on runtime for each operation. The representation encodes which one in the list to pick. For an FJSP instance, the same encoding results in the same machine being assigned, regardless of the situation. However, there is an advantage in sorting machines based on runtime: by limiting the key in the chromosome to less than the number of machines available, one can essentially eliminate the machine-operation pairings with relatively high runtimes, heuristically improving the resulting schedule.

The other way does not encode routing information directly at all. For example, [12] proposed a representation that only consists of scheduling priority. Routing happens

when an operation is about to be scheduled, assigning the machine on which the operation can finish earliest.

Also, a heuristic can be useful to create an initial routing set for a fixed representation. Kacem *et al.*'s Approach by Localization (AL) [2] routes operations one by one, following the precedence constraints, selecting machines with least projected workload, and uses the choice made at each step to update the information used by the next operation, resulting in a routing with relatively balanced workload. Pezzella *et al.* [13] improved on AL by adding heuristics to determine which operation will be routed first. Ho *et al.*'s CDR-Popgen [9] expanded on AL by, instead of routing one operation at a time, routing one operation of each job instead, if one is available. Conflict is resolved by dispatching rules.

2.3.2 In Scheduling

In JSSP and FJSP, the simplest way to schedule an operation is to utilize *priority rules:* simple heuristics that base their decisions on some easily computed parameters of the current state of the partially-solved problem. Examples of those parameters, surveyed by Panwalkar and Iskandar [14], are shown in Table 2.2. Priority rules are quite efficient, but are very limiting when applied without other additional computations.

Table 2.2. Examples of priority rules

Rule	Description
Random	Select job in random order.
FIFO	First in, first out.
SR	Select job with shortest remaining processing time.
DD	Select job with earliest deadline.
NINQ	Select job for which the next operation will use the machine with shortest queue.

More current research usually combines priority rules with other approaches, such as a genetic algorithm, or combines it with a state descriptor, allowing selective deployment of priority rules based on the current state of the system. An example of such *adaptive scheduling* is described in [15]. Adaptive scheduling creates a decision system that takes current state of the scheduling environment into account in order to select the most appropriate dispatching rule for the time.

One of the most popular scheduling methods uses the GT algorithm, a theory-grounded approach proposed by Giffler and Thompson [16]. It limits the number of operations to be scheduled in each decision step to the ones that are in conflict with the earliest finisher, in the following sense: (1) they are assigned the same machine as the operation that can finish earliest, and (2) they can start before the operation that can finish earliest finishes. The outline of the GT algorithm is depicted in Figure 2.3 below. The GT algorithm guarantees to provide an *active schedule*—a schedule that cannot be improved unless an ordering between operations is changed (or in other words, left shifting cannot improve the schedule).

1. Start the set of schedulable operations, C, with the first operation of each job.
2. Calculate the completion time of all operations in C. Find the minimum completion time $t(C)$ among C. Let m^* be the machine where $t(C)$ is achieved.
3. Let G denote the conflict set of operations on set C that run on machine m^* and can start before $t(C)$.
4. Select an operation from G to schedule.
5. Delete the chosen operation from C. Include its immediate successor, if there is one, in C.
6. If C is empty, terminate. If not, return to step 2.

Figure 2.3. GT Algorithm

To use the GT algorithm, we need to find a way to decide which operation in the critical set is to be scheduled next. This is the main focus when combining other approaches with the GT algorithm.

The GT algorithm is not the only schedule generator, however. Ho and Tay [8] have proposed an alternative algorithm, Makespan Computation Algorithm. Operations are scheduled one by one, but insertion between already scheduled operations is allowed if there is a gap large enough to fit the processing time of the current operation in question, while not violating any constraint.

2.3.3 Local Search

Local search techniques are based on the idea that making small changes, called iterative improvements, to the initial solution can lead to at least near-optimal solutions in reasonable computation time. The current solution is a neighbor in the search space of the previously explored solution.

For local search to work, we need (1) an *initial solution* (sometime called a seed) to start the search process from, (2) a *neighborhood function*, or process to produce a set of solutions considered to be the current solution's neighbors, and (3) *selection methods* to determine the next current solution from the neighborhood. To define a neighborhood function, we need to define what a neighbor is to a problem. Neighbors, in general terms, are obtained by making small changes to the current solution. In other words, we need to define what changes to the current solution will be allowed. If a neighborhood function produces too many neighbors to be processed efficiently, we will need sampling techniques to limit search to the most significant (or promising, in some sense) neighbors. Neighborhood functions are usually dependent to the nature of the problem at hand. An example of a neighborhood function in FJSP is changing the order of operations to be processed on a given machine. Below are a few examples of local search techniques:

- *Hillclimbing*, also sometime called classic *iterative improvement*, will select the neighbor that makes the best improvement from the current solution. Hillclimbing will stop when no further improvement can be made. This can, however, lead to a local optimum, where, while it is not an optimal solution or necessarily an acceptable solution, no improvement can found among the neighbors. Additional mechanisms are required to escape local optima.
- To provide a measure of protection against stopping at local optima, a *threshold accepting algorithm* allows selection of a non-improvement neighbor if the difference between the current solution and that neighbor is below a certain threshold. In a threshold accepting algorithm, the threshold starts as a large, non-negative number and gradually decreases to (or toward) 0 in the end. For *simulated annealing*, the threshold values are $-T \ln u$, where T (temperature) is reduced gradually and u is a value derived from a uniform distribution over [0, 1].
- *Tabu (taboo) search* also allows selection of a non-improvement neighbor to avoid local optima. Furthermore, it maintains a taboo list, a list or equivalent memory structure that can be used to identify previously visited solutions, to avoid traversing the same path again during a specified number of future steps. Usually the taboo list will not contain all of the previously visited solutions, just the most recent ones.

Brandimarte [17] has proposed a two-way hierarchical approach to FJSP. First, the routing is performed, transforming an FJSP instance into a JSSP instance. Then, tabu search is performed to solve the scheduling problem. The scheduler then sends critical path information to the router, allowing the routing to focus on the machine assignment

of those operations in the critical path. Hurink *et al* [18] suggested a concurrent approach, where rerouting and swapping are moves in the same tabu search, also focusing on operations on the critical path. Chambers and Barnes [19] incorporated a long-term memory structure that remembers all visited solutions, keeping repetition to a minimum. Mastrolilli and Gambardella [20] have proposed a way to limit the possibility of where the reassigned operation can be inserted into the operation order of its new machine, which they used in their Tabu search technique. Gao *et al.* [21], used GA with variable neighborhood descent: changing the neighborhood structure by reassigning operations on the critical path.

Local search can also be useful when combined with an evolutionary computing approach, such as a genetic algorithm. An EC and Local search hybrid is called a *Memetic Algorithm.*

2.3.4 Particle Swarm Optimization

Modeled after the swarming pattern of migratory birds, Particle Swarm Optimization (PSO) explores a search space by making each individual in the population move toward the best-so-far individual in some way. In each iteration, individuals try to move closer to the best individual, exploring the search space. In order to do this, a *distance function*, a measure to determine how far apart the two individuals are, is required.

Xia and Wu [11] have proposed a representation for their PSO-SA hybrid algorithm. The chromosome contains only the routing policy. First, each operation has its own machine index, based on runtime. Each chromosome allele corresponds to an operation, indicating which machine it will choose based on its machine index. By disallowing large indices, the system prevents operations from choosing machines with relatively large

runtimes. After routing is done, scheduling is performed by using simulated annealing to order operations allocated to each machine.

Grobler *et al.* [12] devised a chromosome without routing policy for PSO, combining it with the GT algorithm. The chromosome contains only priorities for operations. When an operation is picked to be scheduled, a valid machine that allows the operation to finish processing the soonest is selected.

2.3.5 Genetic Algorithm

As mentioned in Section 2.1, to use a GA to solve a scheduling problem, you need a representation first. Below are examples of representations used to solve FJSP, either by GA or other EC approaches (some of which can be adopted for GA), are:

As mentioned in Section 2.1, to use a GA to solve a scheduling problem, a representation must first be developed. Below are examples of representations used to solve FJSP, either by GA or other EC approaches (some of which can be adopted for GA):

- The Parallel Jobs Representation has been proposed by Mesghouni *et al.* [22] A chromosome is a $J \times M$ table. Each row represents a job, where each entry is a pair value: *{machine, start_time}* for each operation in that job, where *machine* is the machine assignment for that operation and *start_time* is the time the operation starts running on its assigned machine. Effort is needed to make sure that the chromosome is valid (no conflicting *start_time* entries, for example) or a repair will be needed. This is a direct representation.
- An indirect representation containing a pair of chromosomes, A and B, was proposed by Chen *et al.* [23]. A is a string of machine assignments (routing

policy). B is a string of precedence orders, one for each machine, containing operations that it will perform. Note that the B-string, if used to construct a schedule directly, needs to be evaluated for validity and to see that it conforms to machine assignments in the A-String and does not violate the precedence constraints of the FJSP instance.

- Zhang and Gen [24] use a multi-stage operation-based GA (moGA) to simplify the chromosome. A moGA chromosome is basically a routing string, one locus for each operation. A Schedule is created by scheduling operations, one by one, to its earliest valid starting time.
- The Assignment Table is proposed by Kacem *et al.* [2]. The assignment table is an *Op (total number of operations)* × *M* table. Each row represents an operation. An entry corresponding to an assigned machine will contain runtime information (starting and completion times) of the operation. This is a direct representation.
- A string of machine indices was proposed [8] to cope with the fact that sometimes an operation can only be run on a subset of all machines. Each entry corresponds to an operation, where the value is the index of a list of valid machines for that operation. The scheduling part in [8] is represented by permutations with repetition of job numbers. [25] improved on this by adding another string which contains precedence constraints between each pair of operations.

- Gao *et al* [26] also uses a pair of chromosomes, where the machine assignment string is a fixed assignment similar to the A-string proposed by Chen *et al* [23], with permutations with repetition used for scheduling.

Due to the relative complexity of scheduling, almost all specialized representations required specialized crossover and mutation operators developed for the scheduling components of the chromosome. These operators are, however, very dependent on the representation used. Examples, along with those used in JSSP, are:

- *Enhanced order crossover* is used with permutation with repetition. [26][21], parents are first converted to conventional permutations, replacing job number with operation indices. Then, the order crossover is performed: a part of one parent is copied to the child, while the rest is filled with entries not found in that part, following the ordering from another parent.
- *Time horizon exchange* (THX) was proposed by Lin *et al* [27]. This crossover operator works on the schedule level by randomly selecting a crossover point. The crossover point is a certain time point in the schedule. Before the crossover point, the child retains identical operation placements from one parent; operations starting after the crossover point are rescheduled, using the GT algorithm, according to the temporal relationships among operations in the other parent.
- Another schedule-level crossover operator, *GT crossover* [28], is based on the GT algorithm, where, in each scheduling iteration, a parent is randomly selected and the operation in the conflict set that starts earliest in the selected parent's schedule is chosen for scheduling next.

2.3.6 Other EC Approaches

Artificial Immune Systems [29] mimic how immune systems in nature react to potentially harmful substances, known as antigens. Antibodies in the immune system will work as pattern recognizers to detect emergent patterns of antigens. Antibodies with higher match scores will pass on to the next generation. This approach is useful in preserving diversity of individuals by developing antibodies that will detect common characteristics among antigens. Ong *et al.*'s ClonaFlex [30] imitates the vertebrate immune system, where the most effective antibody (the schedule with the least makespan) is cloned and mutated rapidly. Diversity is maintained by removing the best individual to a separate (archival) population.

Ant Colony Algorithms, or Ant Colony Optimization (ACO) imitate how ants deploy pheromones to establish pathways from nest to food sources. In this approach, the problem must be defined as a graph. Multiple agents then traverse the graph, cooperating by communication using virtual pheromones. First, agents make solutions. Then pheromones are applied to solutions. Popular subparts of solutions get more pheromones. However, these pheromones degrade over time except for the parts of best-so-far solutions. This preserves common parts among best-so-far solutions. Liouane *et al.* [31] used ACO in routing an FJSP instance and combined it with tabu search, assigning the strength of pheromones to makespans achieved by tabu search using the routings obtained by artificial ants.

Genetic Programming (GP) is similar to genetic algorithms, but individuals in this approach represent explicit programs used to produce solutions. Individuals are usually represented by program trees, where nodes in a tree represent program primitives. Tay

and Ho [32] used GP to develop Composite Dispatching Rules (CDRs) for scheduling purpose.

2.4 *FJSP Rescheduling*

In a manufacturing environment, unexpected complications can occur. A machine can break down; a new job can arrive, with or without notice; a deadline can change, seemingly on a whim. Such changes are called *disruptions*. When a disruption occurs, some changes might be required to accommodate it into the current schedule. Such a process is called *rescheduling*, or *schedule repair*. Figure 2.4 depicts such a disrupted schedule. Operations (2, 2) and (3, 1), whose runtimes overlap with the disruption, are *directly affected* by the disruption. However, since other operations might be job successors or machine successors of the affected operations, their scheduling may be subjected to change as well.

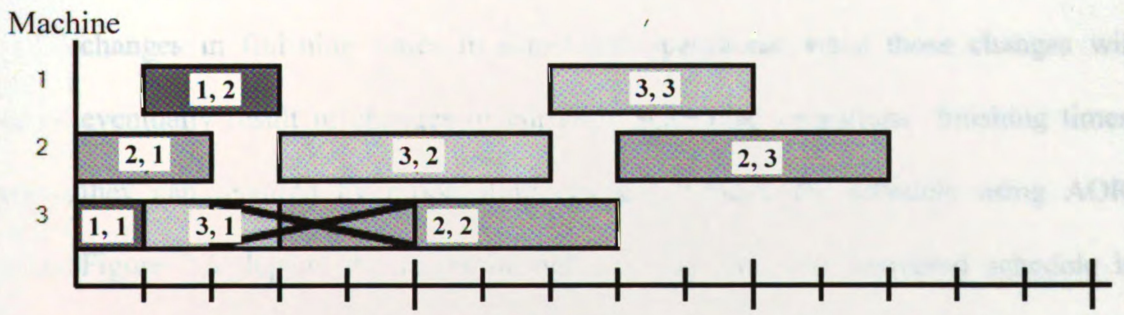

Figure 2.4. A Schedule affected by a disruption

In this case, the non-preemption constraint of FJSP is relaxed, since machines cannot process operations when they break down. Two different assumptions can be made regarding to disrupted operations:

1. They are resumable. They can resume their processings after the disruption is over.
2. They are not resumable. Their progresses are lost and they must restart anew.

Various work has been published on rescheduling in JSSP and its related manufacturing scheduling problems. Examples of such approaches, some of which were surveyed by Vieira *et al.* [33], are:

- *Affected Operations*. Affected operations are identified by first putting operations directly affected by the disruption into the set. Right-shifting is performed on the first operation, and then the algorithm checks (1) its direct job successor, and (2) its direct machine successor. If any of the two operations has an earlier start time than the new finish time, it is affected by the disruption and will be put into the set with updated starting and finishing times. The process is repeated recursively until no new operation is added to the set. Subramaniam and Singh have added preprocessing to adapt AOR to other disruptions [34], such as new job arrivals or changes in finishing times in scheduled operations; since those changes will eventually result in changes of currently scheduled operations' finishing times, they can repaired by propagating changes through the schedule using AOR. Figure 2.5 depicts the result of right-shifting from the disrupted schedule in Figure 2.4. Note that operations (2, 2), (2, 3), (3, 1), (3, 2), and (3, 3) are all affected, and have to be right-shifted, even though only operations (2, 2) and (3, 1) are directly affected.

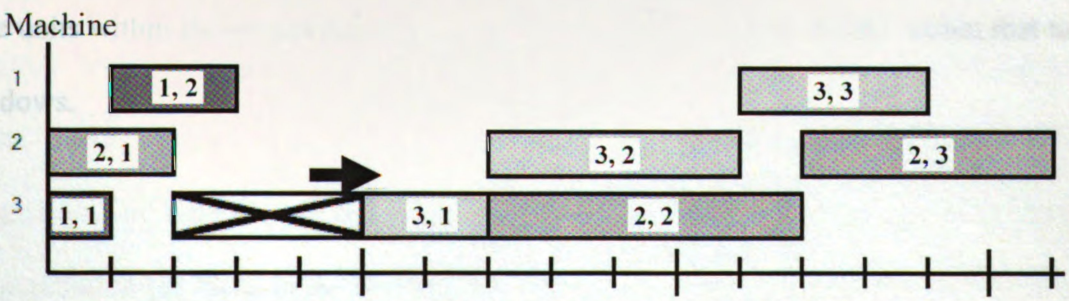

Figure 2.5. The Schedule after right-shifting

- Partial Rescheduling. Only the affected operations are rescheduled. Lin *et al.* [27], proposed GT Rescheduling to handle new job arrival. When a job arrives, the schedule stops at arrival time. The operations that have not started at that time, along with operations for the new job, are rescheduled using the GT algorithm. At the conflict resolution phase, the algorithm will either pick an operation from the new job, or follow the priorities of the old schedule.
- Total Rescheduling. The entire schedule is rebuilt. In EC, a new evolution cycle can take place for the affected schedule. This approach is more likely to find an optimal repaired schedule, but at higher computational cost. Of course, in real-world application, such an approach might even halt progress until a new schedule is found.

Unfortunately, only two previous works regarding scheduling in dynamic FJSP have been found. Gholami *et al.* [35] suggested using right-shifting with a FJSP representation proposed by Gao *et al.* [21]. Meanwhile, Grobler *et al.* [12] adapted PSO schemes utilized with static FJSP. Both approaches assumed disrupted operations are *resumable*. That is, affected operations can continue their processing after the disruption is over, without having to restart anew. Gao *et al* applies their approach in [26] to deal with FJSP with scheduled maintenance [36]. Maintenance tasks must take place for a predefined time units within the pre-specified time windows, but they can be shifted within that time windows.

2.5 *Multiobjective Optimization*

Sometime, there are needs of optimizing a solution not against just one objective function, but multiple ones, often of conflicting priority. A few ways to combine objectives together so that an individual can be ranked are described below:

First is aggregation, combining all objectives into a single weighted sum. Gao *et al.* [26] and Xia and Wu [11] used weighted objectives to combine normalized values of makespan, total workload, and critical workload together, favoring them in that order. Ishibuchi *et al.* [37] used randomized weights to combine objectives. Kacem *et al.* [38] used fuzzy logic to calibrate weights in dealing with the same set of objectives.

The next approach is population based. In each generation, subpopulations are created, one optimized for each objective; then the populations are recombined. Schaffer's Vector Evaluated Genetic Algorithms (VEGA) [39] is such an approach.

To understand the next approach, first we need to understand the concept of *domination* in the context of multiobjective optimization. An individual *dominates* another individual by (1) performing at least as well on every objective, and (2) performing better on at least one objective. This notion is used in defining Pareto sets—sets of solutions in which no solution dominates another.

A popular Pareto-based approach is the Non-dominated Sorting GA II (NSGA-II) [3]. NSGA-II works by sorting individuals into multiple non-dominated fronts. The first front is the Pareto front, a group of individuals not dominated by any individual in the population. The second front is a group of individuals not dominated by any individual in the population but those in the Pareto front, and so on. The first non-dominated front will be chosen first to move on to the next generation, then the second front and so on. Once

the remaining empty population quota is smaller than the size of the non-dominated front currently being considered, neighborhood distance is used, favoring the ones located in the more sparse parts of the front. The Strength Pareto Evolutionary Algorithm (SPEA) [40] is another Pareto-based approach. SPEA and its successor, SPEA2 [41], maintain a library of non-dominated individuals for the purpose of breeding. The library is updated every generation, adding new non-dominated individuals and removing dominated members of the library. If the capacity of the library is exceeded, a density function is used to assist in removing individuals in a way that maintains diversity. Ho and Tay used NSGA-II's fast ranking system to determine the best individuals to perform local search on [10].

2.6 *Summary*

Although there has been a good deal of research on the FJSP, most focus has been on the static problem, optimizing single objectives such as makespan. There has been very little work on FJSP rescheduling. Hence, current approaches in FJSP do not seem to support rescheduling as is. Some of the obstacles are:

1. Routing components of the representation are usually fixed. For the lifetime of an individual, each operation will be assigned a specific machine. This tends to function well enough in static environments, but in case of rescheduling, a fixed routing can be a hindrance. A routing that can adapt in case of unexpected change in the system will be more useful.
2. Although many novel approaches in routing exist, such as Approach by Localization (AL) or Composite Dispatching Rules (CDR), most approaches are only used to enhance the quality of a fixed routing of an initial population in a static problem. Heuristics like these can be useful for selecting a more suitable machine for an affected operation in the case of rescheduling. It is unlikely that there exists a single heuristic that performs well in every situation. A set of condition-action aggregates similar to those of adaptive scheduling [15] should be beneficial to rescheduling throughput.
3. Currently, those few works on FJSP that handle disruptions ([35] and [12]) use only right-shifting to deal with the disruptions. Although simple to implement, the performance of right-shifting is at the mercy of the length of the disruption. In order to avoid that, a schedule repair mechanism that allows reassignment of affected operations will be required.

CHAPTER 3

ADAPTIVE REPRESENTATION (AdRep)

As seen from the research mentioned in the previous chapter, routing decisions are mostly done before any scheduling starts. We believe that by delaying routing decisions until necessary and providing choices of basic heuristics to guide the routing process, we can improve machine assignments, increasing the quality of solutions as a whole. Also, since the AdRep representation does not bind an operation to a machine, it can also be used as a blueprint for rescheduling in the case of machines becoming unavailable due to disruption. In this chapter, the AdRep representation is described in Section 3.1. In Section 3.2, the rescheduling process using AdRep representation is discussed. Section 3.3 illustrates how AdRep works with a Genetic Algorithm.

3.1. Representation

Most work on using Evolutionary Computing to solve FJSP performs routing separately from scheduling, performing routing of all operations first, transforming the FJSP instance into a JSSP instance, and then performing JSSP scheduling. However, there may be some advantages in delaying the routing decisions as long as possible. If we use the GT algorithm with our approach, there is no need to designate a machine for an operation until that operation is being considered for scheduling, which is the time it is put into the schedulable operations set for the first time.

The proposed representation consists of 2 integer vectors. The first vector contains routing policy. The second vector contains scheduling policy. This representation is an

indirect one. It will be used by a GT algorithm to create a schedule. Figure 3.1 illustrates the chromosome of this representation.

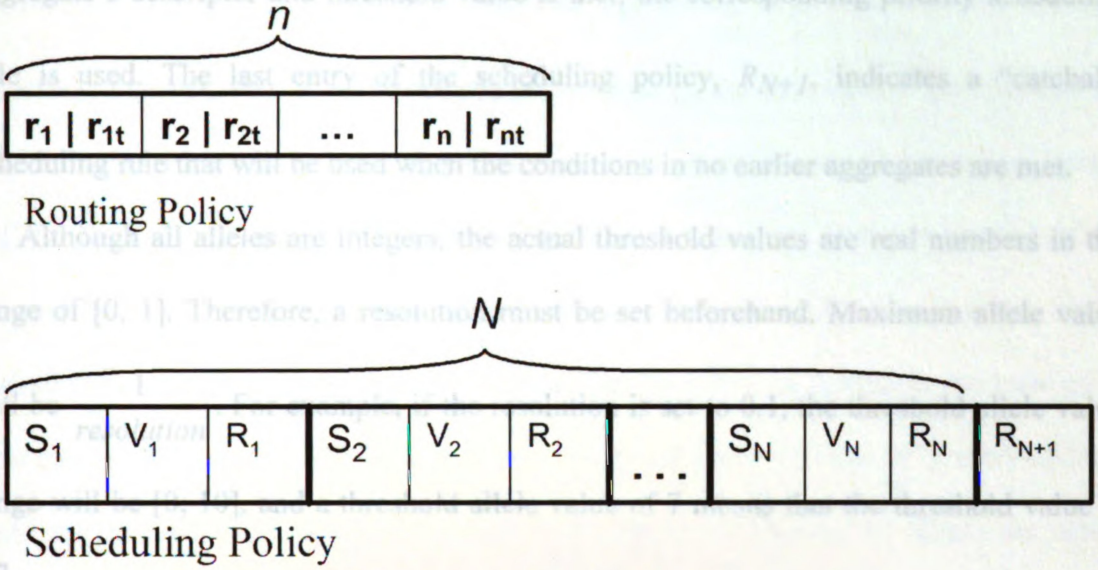

Figure 3.1. The AdRep chromosome

Since there is one routing policy entry for each operation, the length of the routing policy component of the chromosome is equal to the number of operations in the FJSP instance (n). The user specifies how many schedule aggregates will be in a chromosome (N). Usually, the length of the routing component is considerably greater than that of the scheduling component.

In the modified GT algorithm, machine assignment for an operation is delayed until that operation is ready to be scheduled, i.e., being put into the schedulable set. Then, a machine will be selected based on the routing rule indicated in the corresponding entry in the routing policy. Then, after the operation in the schedulable that can finish earliest is located, a conflict set is created. The conflict set consists of operations that (1) currently belong in the schedulable set, (2) are assigned the same machine as the earliest finisher, and (3) can start before the earliest finisher is projected to finish. One of the operations

from the conflict set will be selected to be scheduled. To do that, the adaptive rule aggregates are checked, from left to right. If the state condition indicated by the aggregate's descriptor and threshold value is met, the corresponding priority scheduling rule is used. The last entry of the scheduling policy, R_{N+1}, indicates a "catchall" scheduling rule that will be used when the conditions in no earlier aggregates are met.

Although all alleles are integers, the actual threshold values are real numbers in the range of [0, 1]. Therefore, a resolution must be set beforehand. Maximum allele value will be $\frac{1}{resolution}$. For example, if the resolution is set to 0.1, the threshold allele value range will be [0, 10], and a threshold allele value of 7 means that the threshold value is 0.7.

3.1.1. Routing

The routing policy vector consists of machine selection (routing) rules, one entry for each operation. Each entry utilizes two routing rules—the second rule (r_{it}), or *tiebreaking rule*, is used if there is a tie from the first rule. By utilizing two routing rules per operation instead of just one, or a more complex routing decision mechanism, we hope to achieve more powerful search, while not overfitting the routing decision. Table 3.1 contains the routing rules used in this new representation.

Table 3.1. Routing rules

Routing Rule	Description
Earliest Finish (EF)	Select the machine that will allow the operation to finish earliest.
Smallest Runtime (SR)	Select the machine with smallest runtime.
Smallest Workload (SW)	Select the machine with smallest workload.
Least Projected Crowding (LPC)	Select the machine with least Projected Crowding value

The EF rule is inspired by the GT algorithm, but will cover all possible machine choices. The SR rules are among the machine selection rules utilized by Subramaniam *et al.* in [42]. SW is an attempt to incorporate optimizing W_M and W_T into machine selection. The LPC rule is an attempt to incorporate global information into the routing process. LPC works by comparing values in entries of the Projected Crowding Vector (PCV). There is one entry in the PCV entry for each machine. At the beginning of the scheduling process, the PCV entry for a machine is determined by counting how many operations have shortest runtimes on that machine. Note that an operation can have more than one machine that provides the least runtime, resulting in multiple PCV entry updates. Once an operation is scheduled, crowding information is updated to reflect the actual schedule being constructed, removing entries in the machines the operation has least runtime on, but is not scheduled in, and adding itself into the PCV entry of the machine it is scheduled to run on. By choosing a machine with a smaller projected crowding value, workload is expected to be distributed more evenly. At the end of the evaluation, PCV resets itself to its initial value, ready for the next evaluation. Figure 3.2 shows how PCV updates itself.

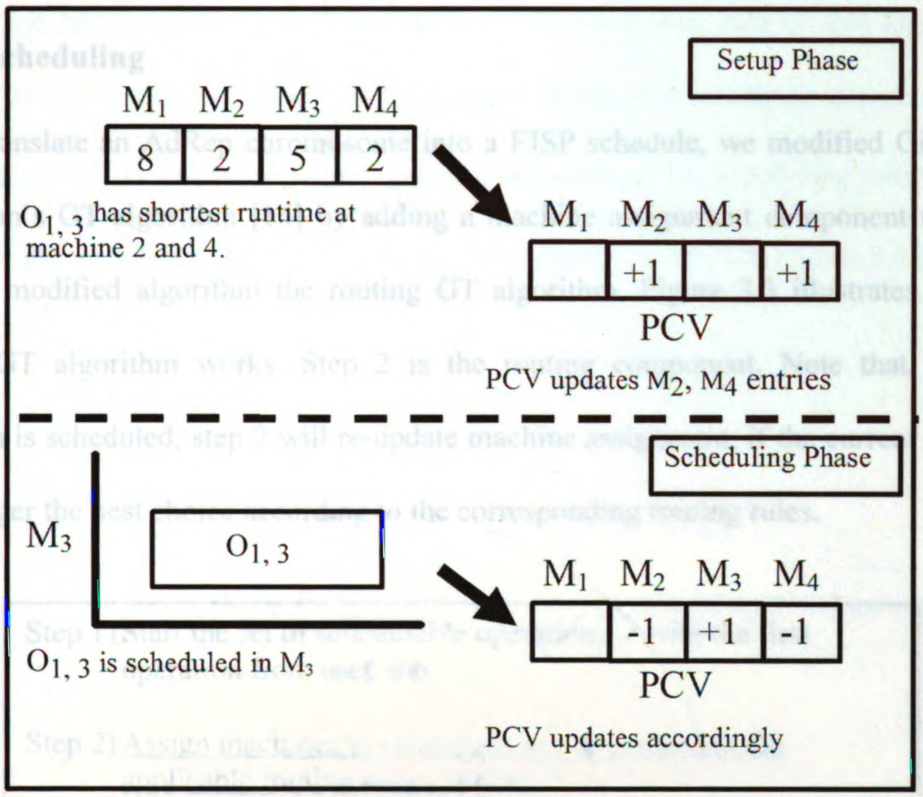

Figure 3.2. How Projected Crowding Vector works

Since our routing policy is a set of routing rules, not direct machine assignments, as long as there is a way to exclude invalid machines from routings, AdRep will be able to handle P-FJSP instances without producing any invalid chromosomes.

Note that, since the routing component makes up the majority of the chromosome, the number of routing rules available can greatly affect search space, and convergence might suffer as a result. It might be necessary to limit some pairings of the routing rules to limit dimensionality somewhat.

3.1.2. Scheduling

To translate an AdRep chromosome into a FJSP schedule, we modified Giffler and Thompson's GT algorithm [16] by adding a machine assignment component to it. We call this modified algorithm the routing GT algorithm. Figure 3.3 illustrates how the routing GT algorithm works. Step 2 is the routing component. Note that, after an operation is scheduled, step 2 will re-update machine assignment, if the current selection is no longer the best choice according to the corresponding routing rules.

Step 1) Start the set of schedulable operations, *c* with the first operation from each job.

Step 2) Assign machines to operations in *c* according to the applicable routing policy entries.

Step 3) Calculate completion times for all operations in *c*. Find the earliest finisher. Let *m** be the machine the earliest finisher is assigned to.

Step 4) Create the conflict set *g*, from every machine in *c* that is assigned to *m** and can start before the earliest finisher can finish its runtime.

Step 5) Use condition-rule scheduling aggregates to select an operation in *g*. Schedule that operation and delete it from *c*. Add its job successor, if one exists, to *c*.

Step 6) If *c* is empty, terminate. If not, return to step 2

Figure 3.3. Routing GT Algorithm

At step 5 of the routing GT algorithm, to choose an operation in the conflict set to be scheduled, AdRep scheduling, a modified version of adaptive scheduling [15], utilizes the scheduling aggregates to make the decision. It starts by consulting the leftmost scheduling aggregate. If the condition specified by the state descriptor and the threshold

value of the first scheduling aggregate is met, the scheduling rule of that aggregate is used to determine the operation to be scheduled. If not, the scheduler moves on to the next aggregate. The catchall rule (R_{N+1}) at the last entry of the scheduling policy is used if none of the conditions specified in the former aggregates are met. Figure 3.4 illustrates how the scheduling process works.

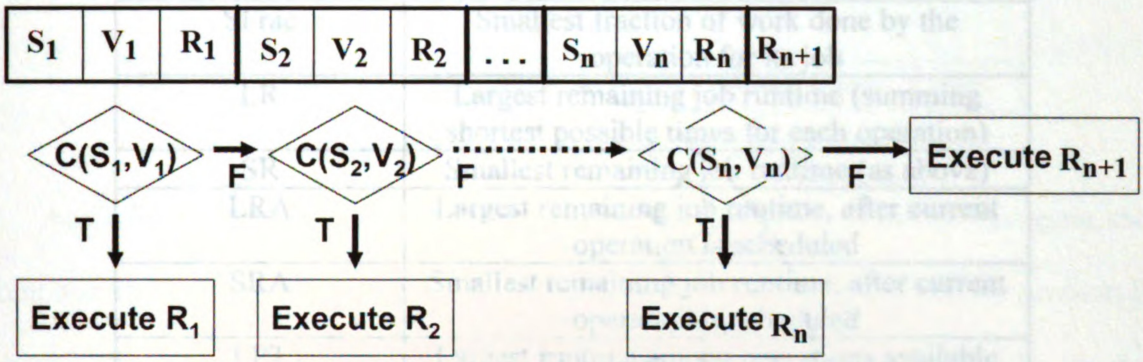

Figure 3.4. Order of execution of scheduling aggregates

State condition descriptors used in this dissertation are listed in Table 3.2. The scheduling dispatching rules used in this dissertation are listed in Table 3.3. Each descriptor is followed by a suffix –LE or –G., indicating whether to execute the rule when the condition is less-than-or-equal (-LE) or greater (-G) than the threshold value.

Table 3.2. State descriptors

State Descriptor	Description
RCW	Relative current machine workload
RTW	Relative average machine workload
CON	Machine contention
COM	Current Completion

RCW checks how well the machine has been utilized so far. RTW examines the utilization rate of all machines in the system. Machine Contention, which counts how many operations are assigned to it at this time, indicates the current competition for that machine. Completion rate checks the percentage of scheduled operation compared to the

total number of operations, indicating how much work relating to all available work has been done so far and how close it is to being finished.

Table 3.3. Scheduling priority rules

Priority Rule	Description
COp	Smallest current-runtime-to-shortest-runtime ratio
EFS	Earliest finishing operation
LFrac	Largest fraction of work done by the operation for its job
SFrac	Smallest fraction of work done by the operation for its job
LR	Largest remaining job runtime (summing shortest possible times for each operation)
SR	Smallest remaining job runtime (as above)
LRA	Largest remaining job runtime, after current operation is scheduled
SRA	Smallest remaining job runtime, after current operation is scheduled
LPT	Largest runtime among operations available for scheduling now
SPT	Smallest runtime among operations available for scheduling now
OPT	Operation with current runtime closest, proportion-wise, to its shortest value

3.1.3. Duplication Count

During the process of development, we noticed that AdRep suffered greatly from *false competition*, finding multiple distinct chromosomes that actually represent identical solutions (schedules). In order to alleviate this, a schedule-counter map was introduced to monitor the population during the evolutionary process. If an individual that produces the same schedule and fitness values as another already discovered is encountered, the counter for that schedule of that fitness value is increased. This counter is then used to reward individuals that produce novel schedules and thereby to attempt to reduce false competition, discouraging a schedule represented by multiple chromosomes from

crowding out other schedules in the population. Fitness values are also remembered in solving the dynamic FJSP, where different AdRep individuals might create identical initial schedules, but differ on rescheduling, resulting in different repaired makespan fitness values. This mechanism is similar to that of global memory in [19].

3.1.4. Chromosome Example

Here, in Figure 3.5, is a sample chromosome, where $n = 3$ and $N = 1$:

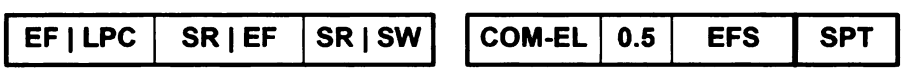

Figure 3.5. an example chromosome

According to the first entry of the routing policy component of the chromosome, the first operation selects a machine by choosing the one that allows it to finish the soonest; if there is more than one such machine, the one with the least PCV value is chosen. On the other hand, the second operation, which corresponds to the second entry, favors the machine with the least runtime, breaking ties using the earlier finishing time. The last operation also prefers the machine with the shortest runtime. However, it will break a tie using a workload criterion, picking the one with the least workload if there is more than one machine with smallest runtime.

On the scheduling side, if there is more than one operation in the conflict set, the Routing GT algorithm first looks at completeness—how many operations have been scheduled compared to the total number of operations. If completeness is less than or equal to 0.5—not more than 50% of total operations have been scheduled yet—it selects the operation with earliest finishing time to be scheduled. If not, and there is no more scheduling aggregate, the catchall rule is used. In this case, it is the SPT rule. The operation with the smallest runtime is selected.

3.2. *Rescheduling*

A *machine-unavailability disruption* (henceforth will be referred just as "disruption") is a period of time when a certain machine is not available to process operations. It can come from many causes: machine breakdown, sudden operator sickness, or maintenance. In the scope of this dissertation, we will focus on this type of disruption. When a disruption occur, operations currently scheduled on the disrupted machine will have to either (1) delay their processing until the disruption is over, or (2) be reassigned to another machine.

Sometimes, during the course of a manufacturing day, there will be more than one disruption. A *disruption scenario* is a set of disruptions that occur in a single evaluation. Each scenario is composed of possibly multiple *disruption events,* each of which is a set of disruptions that start at the same time on different machines.

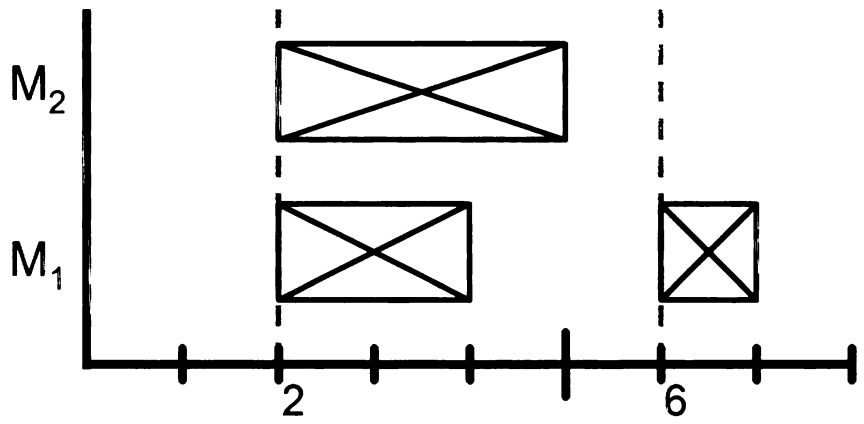

Figure 3.6. A disruption scenario

Figure 3.6 depicts a disruption scenario. Here, there are two disruption events, one at time 2, where both M_1 and M_2 break down. Note that both will not become operational

again at the same time. M_1 will come back at time 4, while M_2 will work again at time 5. The second event occurs at time 6, where M_1 breaks down again for 1 time unit.

When a disruption occurs, a *rescheduling* process is used to repair the schedule. The repaired schedule must avoid the disruption, still following all constraints, and minimizing repaired makespan. However, we want a scheduler that performs well when there is no disruption, as well. Therefore, we should also consider *initial makespan*, makespan from a disruption-free schedule. A good scheduler should produce good values for both objectives.

3.2.1. Disruption-prone FJSP

In the scope of this dissertation, the following assumptions have been made about the nature of disruptions:

- The disruption pattern is unknown to the AdRep rescheduler. Only reactive repair will be performed to cope with the disruptions.
- The end time of a disruption is known at the time it starts. The rescheduler can plan to place operations after the disruption's end time, with no risk that the disruption will continue past that time. There are no guarantees that the operation will be safe from a further disruption, however.
- Operations are not resumable. If an operation is affected by a disruption, it must restart its processing anew. There will be no partial completion gained from the processing interrupted by the disruption. However, the job will not lose work completed by the disrupted operation's predecessors. Only the work of the affected operation needs to be redone from its start.

3.2.2. AdRep and Rescheduling

The AdRep rescheduler will handle one *disruption event*: a collection of disruptions that start at the same time. If another disruption event occurs, the rescheduler needs to be called again. Figure 3.7 shows how the rescheduler works against a scenario.

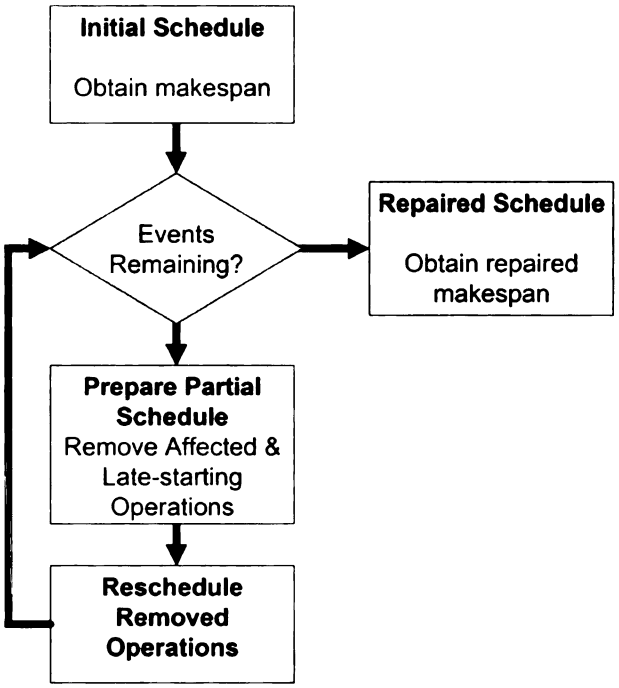

Figure 3.7. Rescheduling a disruption scenario

An operation is directly affected by the disruption if the following conditions are met:

1) It shares a machine with the disruption.
2) There is an overlap between the operation's runtime and the disruption. If the operation starts before the disruption ends and ends after the disruption starts, there is an overlap.

Two types of operations will be rescheduled: (1) those that are directly affected by the disruption(s), and (2) those that start at the same time as the disruption event, or after that. This will allow maximum flexibility in terms of rerouting. We use the Routing GT again to reroute and reschedule operations.

Figure 3.8 is an example of rescheduling in response to one disruption event.

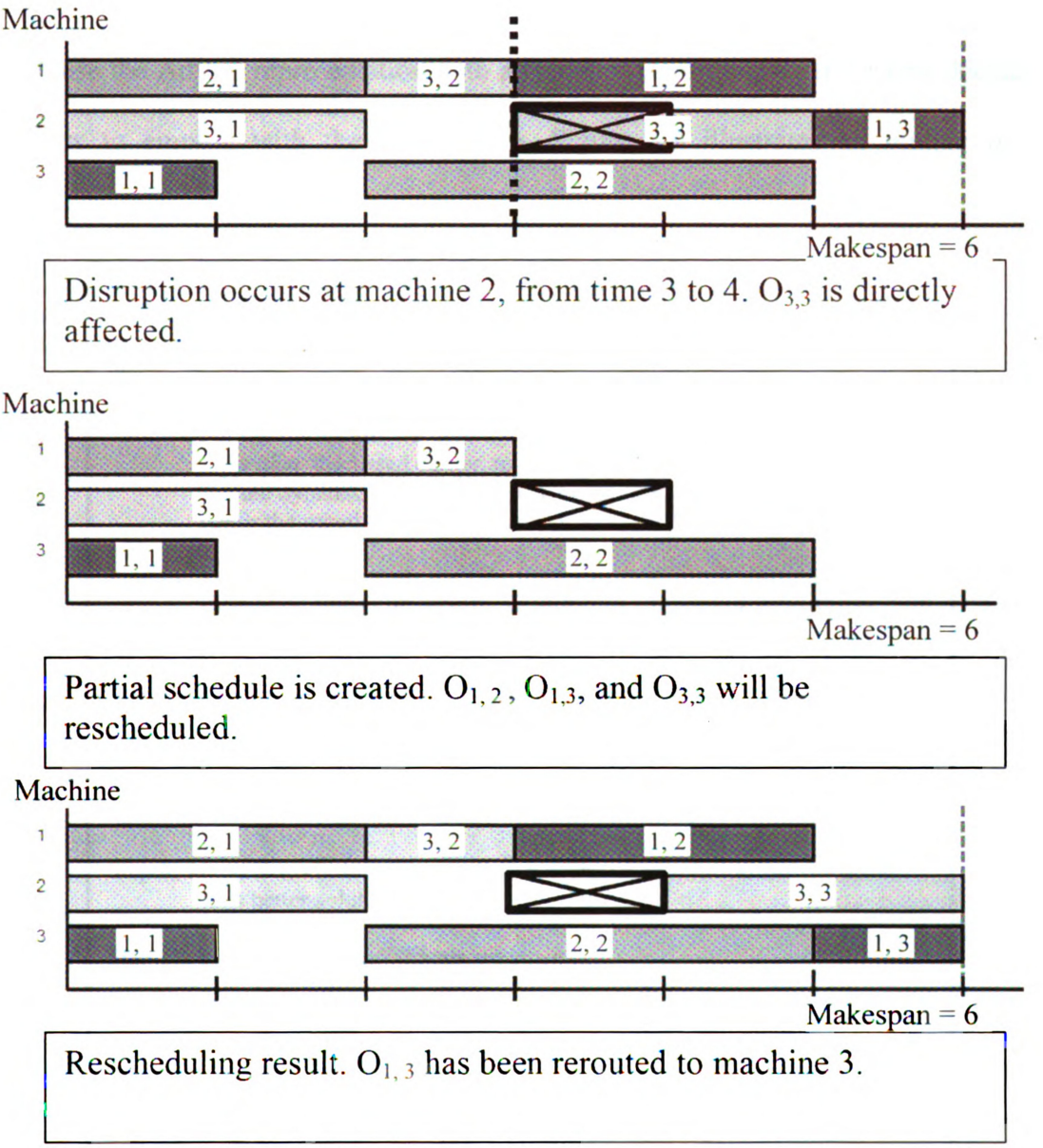

Figure 3.8. Rescheduling against an event

Note that, since there is a change in the current schedule environment, routing policy may assign an operation to a different machine. Also, the order of operations in a machine might change. This allows AdRep to adapt to disruption, preserving makespan as best as it can.

3.3. Using AdRep with a Genetic Algorithm

To use the AdRep representation with a Genetic Algorithm, we must first decide the operators to employ with the GA system. Figure 3.9 illustrates an outline of our framework.

Step 0) ***Setup.*** Parameters and data structures are initialized.

Step 1) ***Initialization.*** Initial population is generated randomly.

Step 2) ***Evaluation.*** Evaluation operator determines fitness of an individual. For the static case, we used the *Routing Giffler-Thompson (GT) algorithm,* discussed earlier, to create a schedule from a chromosome. For the dynamic case, Routing GT might be called multiple times to repair the schedule. All objectives involved will be discussed below.

Step 3) ***Selection.*** Individuals are selected for mating. Since NSGA-II is already an elitist replacement strategy, we use random selection to alleviate premature convergence.

Step 4) ***Crossover.*** Two individuals are recombined together, creating offspring that inherit some traits from each parent. We use two-point crossover.

Step 5) ***Mutation***. Small changes are applied to individuals, exploring the search space. Details will be discussed below.

Step 6) **Local Search**. Iterative search is performed on selected individuals to seek better solution.

Step 7) Newly created individuals are evaluated.

Step 8) ***Replacement Strategy*** decides which individuals will pass on to the next generation. We use the Non-dominated Sorting GA II (NSGA-II) [3], which was discussed earlier.

Step 9) **Migration.** Some individuals from one subpopulation are randomly selected to be relocated to another subpopulation.

Step 10) If the ***Termination Criterion*** is met, the GA run is stopped. If the max generation has been reached, the program terminates. If not, the program returns to Step 3.

Figure 3.9. The GA framework

Since our replacement strategy, NSGA-II, is inherently elitist, we use random selection to choose individuals for crossover and mutation, in order to maintain some diversity in the population.

AdRep representation is, in essence, an array of integers. The initial population can be generated randomly without the fear of illegal chromosomes, since all values in any allele have valid translations.

3.3.1. Crossover Operation

We use an adapted two-point crossover with the AdRep chromosome, applying crossover independently to each of the two parts of the chromosome. For the routing vector, it can be used without any modification. For the scheduling vector, however, care must be taken so that an adaptive rule aggregate is not divided by crossover. We should treat an aggregate as an atomic locus. Figure 3.10 shows an example of such crossover.

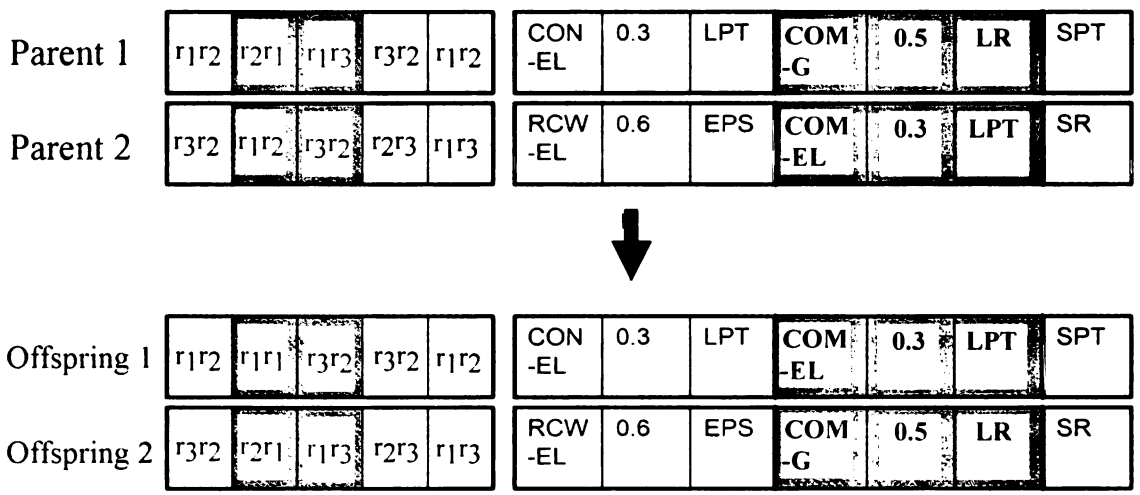

Figure 3.10. AdRep Crossover

3.3.2. Mutation Operation

For the routing policy vector, uniform mutation can be used simply to change the machine assignment of an operation. Mutating a routing allele changes the routing rule the corresponding operation will use to select a machine. For the adaptive scheduling policy, a few different mutations are used.

- The order between rule aggregates can be swapped, changing priority between them.
- Individual entries in an aggregate can also be changed. Since the threshold values tend to be represented with more resolution (higher cardinality of alleles) than the condition descriptor and scheduling rule, care must be taken that the threshold values be explored as evenly as the other parts of the scheduling aggregate.

A list of mutation operators is shown in Table 3.4.

Table 3.4. List of Mutation Operators

Mutation Operator	Description
Machine Reassign	Change values in routing policy vector.
Aggregate Swap	Exchange positions of two scheduling aggregates.
Descriptor Change	Change the descriptor of an aggregate.
Value Change	Change the threshold of an aggregate.
Priority Rule Change	Change priority rule of an aggregate.

3.3.3. Replacement Scheme – from NSGA-II

To determine which individuals will pass on to the next generation, we use the non-dominated sorting rule from the Non-dominated Sorting GA II (NSGA-II) [3] , a Pareto-front optimization replacement method. Non-dominated sorting works by sorting individuals into multiple non-dominated fronts. The first front is the Pareto front, a group of individuals not dominated by any individual in the population. The second front is a

group of individuals not dominated by any individuals in the population after those in the Pareto front are excluded, and so on. The first non-dominated front is chosen first to move on to the next generation, then the second front and so on. Once the empty space remaining in the next-generation population is smaller than the size of the non-dominated front currently being considered, neighborhood distance is used, favoring individuals located in the more sparsely populated parts of the front in determining which solutions survive.

3.3.4. Local Search

A local search technique has been devised to assist AdRep in covering the search space. We use a simple hillclimbing algorithm to make a series of small changes to the selected individual. The local search stops when no more improvement is being made or when the specified maximum number of steps, also known as *maximum local search depth*, has been reached.

Although local search can be useful, it can be expensive, since each neighborhood lookup in AdRep requires a new evaluation. Also, care must be taken to avoid premature convergence. A parameter (p_{Local}) will limit the probability that a selected individual is used as the starting point for a local search.

3.3.5. Objective Functions

In static, disruption-free cases, we will use three main objectives: (1) makespan, (2) maximum machine workload, and (3) total machine workload. Two helper objectives are also used: duplication counts, as mentioned in Section 3.1.3 and Guided Workload Distribution (GWD).

GWD rewards a schedule that distributes workloads more evenly among machines. However, only rewarding an even distribution will not guarantee a good schedule. Instead, GWD is based on the following formula:

$$GWD = C_M^{\,2} + W_T + W_M^{\,2} + \sigma_{workload}$$

where $\sigma_{workload}$ is the standard deviation of workloads among machines. This way, GWD will be bound by all three main objectives, eliminating individuals that distribute workload evenly among machines but do not perform as well in terms of the main objectives.

For disruption-prone dynamic cases, two objectives will be the main foci: makespan (C_M) and maximum repaired makespan (C_{RMax}). Average repaired makespan (C_{RAvg}) and duplication count will be used as helper objectives. The AdRep rescheduler will be tested against not one scenario at a time, but a disruption scenario set which contains a number of disruption scenarios. The greatest makespan from a repaired schedule caused by a scenario in the set will be used as the maximum repaired schedule makespan. The mean of all makespans from repaired schedules will be used as the average repaired makespan. Average repaired makespan is used as a helper objective to reward a rescheduler that can repair many schedules well, even if there are a few schedules it does not handle as well, resulting in high maximum repaired makespans. Figure 3.11 shows how AdRep interacts with disruption scenarios in the dynamic case.

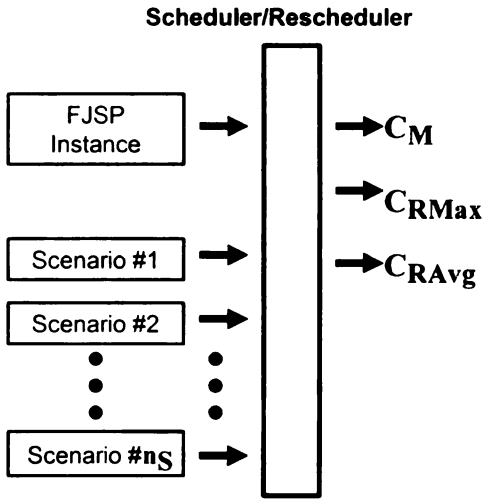

Figure 3.11. How AdRep derives fitness values in dynamic cases

3.3.6 The AdRep Framework

Combining everything together, we have an AdRep framework, as depicted in Figure 3.12 below:

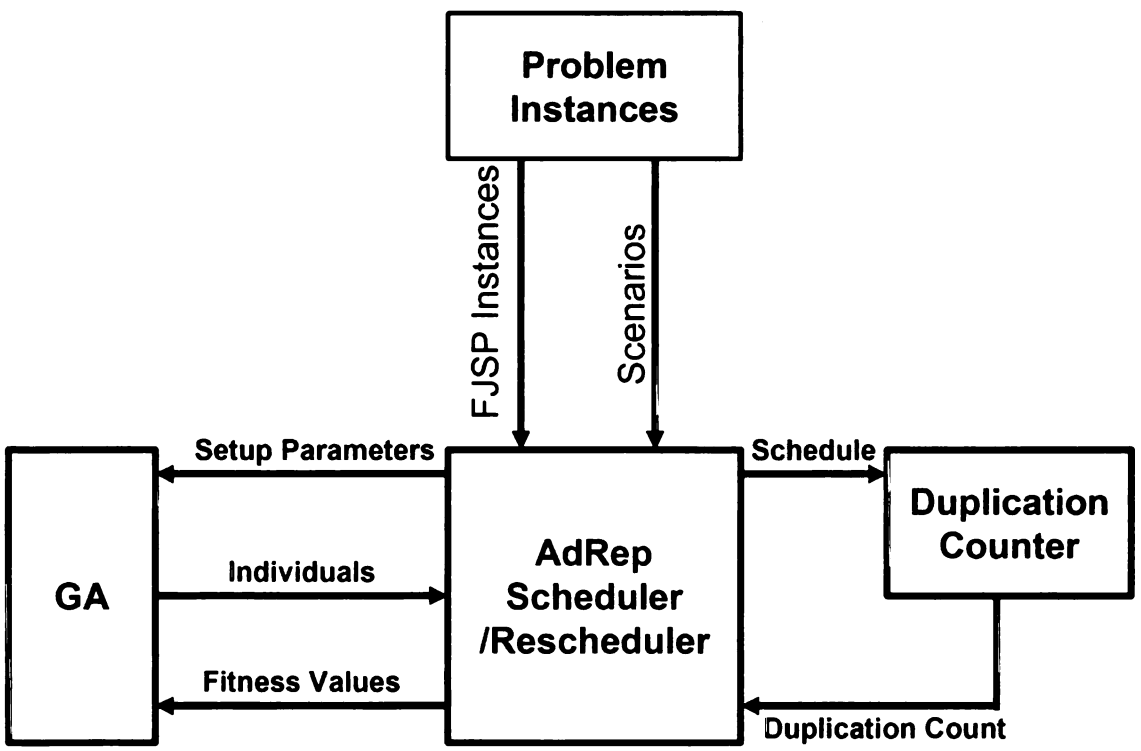

Figure 3.12. The AdRep framework

First, the AdRep scheduler (and rescheduler) reads in problem instances, including an FJSP instance, and, in case of dynamic instances, a set of disruption scenarios. Information from the problem instance is used to start the GA framework, which sends individuals to be evaluated. The duplication counter serves as a long-term memory which, when presented with a schedule, along with the schedule's fitness values (which might not be the same for identical schedules in the dynamic case), returns the duplication count, which is used to discourage repetition.

CHAPTER 4

PERFORMANCE ANALYSIS—STATIC FLEXIBLE JOB-SHOP SCHEDULING PROBLEMS

In this chapter, we will test our AdRep representation on published FJSP instances. The results will be compared with published Pareto fronts. We then will discuss the result in terms of effectiveness—whether the published Pareto fronts are matched or exceeded—and in term of efficiency—how efficiently AdRep performed.

In addition to the usual FJSP assumptions, the following assumptions are made in the problem instances that are used in this experiment:

- All machines are available at time 0.
- All jobs are released to be scheduled at time 0.
- There is no cost in moving jobs from one machine to another. The only cost that is incurred is the runtime cost from scheduling operations on machines.

Section 4.1 depicts benchmark FJSP instances that were used in the experiments, along with their published best solutions (points on their Pareto fronts). Section 4.2 discusses the parameters and design of experiments. The results of experiments are presented in Section 4.3. Section 4.4 concludes the chapter and discusses the results.

4.1. *Benchmark Problem Instances*

For the static model, we will use 3 FJSP instances reported in various publications. The first is an 8 × 8 FJSP instance with partial flexibility. [2] In this instance, there are 8 jobs with 27 operations. This instance is illustrated by Table 4.1, below. An 'X' entry indicates that the operation is not compatible with that particular machine and cannot be scheduled there.

Table 4.1 An 8 × 8 FJSP instance with partial flexibility

		M1	M2	M3	M4	M5	M6	M7	M8
J1	O1,1	5	3	5	3	3	X	10	9
	O1,2	10	X	5	8	3	9	9	6
	O1,3	X	10	X	5	6	2	4	5
J2	O2,1	5	7	3	9	8	X	9	X
	O2,2	X	8	5	2	6	7	10	9
	O2,3	X	10	X	5	6	4	1	7
	O2,4	10	8	9	6	4	7	X	X
J3	O3,1	10	X	X	7	6	5	2	4
	O3,2	X	10	6	4	8	9	10	X
	O3,3	1	4	5	6	X	10	X	7
J4	O4,1	3	1	6	5	9	7	8	4
	O4,2	12	11	7	8	10	5	6	9
	O4,3	4	6	2	10	3	9	5	7
J5	O5,1	3	6	7	8	9	X	10	X
	O5,2	10	X	7	4	9	8	6	X
	O5,3	X	9	8	7	4	2	7	X
	O5,4	11	9	X	6	7	5	3	6
J6	O6,1	6	7	1	4	6	9	X	10
	O6,2	11	X	9	9	9	7	6	4
	O6,3	10	5	9	10	11	X	10	X
J7	O7,1	5	4	2	6	7	X	10	X
	O7,2	X	9	X	9	11	9	10	5
	O7,3	X	8	9	3	8	6	X	10
J8	O8,1	2	8	5	9	X	4	X	10
	O8,2	7	4	7	8	9	X	10	X
	O8,3	9	9	X	8	5	6	7	1
	O8,4	9	X	3	7	1	5	8	X

The second is a 10 × 10 FJSP instance with full flexibility. [2] In this instance, there are 10 jobs with 30 operations. This instance is illustrated by Table 4.2.

Table 4.2 A 10 × 10 FJSP instance with total flexibility

		M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
J1	O1,1	1	4	6	9	3	5	2	8	9	5
	O1,2	4	1	1	3	4	8	10	4	11	4
	O1,3	3	2	5	1	5	6	9	5	10	3
J2	O2,1	2	10	4	5	9	8	4	15	8	4
	O2,2	4	8	7	1	9	6	1	10	7	1
	O2,3	6	11	2	7	5	3	5	14	9	2
J3	O3,1	8	5	8	9	4	3	5	3	8	1
	O3,2	9	3	6	1	2	6	4	1	7	2
	O3,3	7	1	8	5	4	9	1	2	3	4
J4	O4,1	5	10	6	4	9	5	1	7	1	6
	O4,2	4	2	6	8	7	4	6	9	8	4
	O4,3	7	3	12	1	6	5	8	3	5	2
J5	O5,1	7	10	4	5	6	3	5	15	2	6
	O5,2	5	6	3	9	8	2	8	6	1	7
	O5,3	6	1	4	1	10	4	3	11	13	9
J6	O6,1	8	9	10	8	4	2	7	8	3	10
	O6,2	7	3	12	5	4	3	6	9	2	15
	O6,3	4	7	3	6	3	4	1	5	1	11
J7	O7,1	1	7	8	3	4	9	4	13	10	7
	O7,2	3	8	1	2	3	6	11	2	13	3
	O7,3	5	4	2	1	2	1	8	14	5	7
J8	O8,1	5	7	11	3	2	9	8	5	12	8
	O8,2	8	3	10	7	5	13	4	6	8	4
	O8,3	6	2	13	5	4	3	5	7	9	5
J9	O9,1	3	9	1	3	8	1	6	7	5	4
	O9,2	4	6	2	5	7	3	1	9	6	7
	O9,3	8	5	4	8	6	1	2	3	10	12
J10	O10,1	4	3	1	6	7	1	2	6	20	6
	O10,2	3	1	8	1	9	4	1	4	17	15
	O10,3	9	2	4	2	3	5	2	4	10	23

The third is a 15 × 10 FJSP instance with total flexibility. [38] In this instance, there are 15 jobs with 56 operations. This instance is illustrated by Table 4.3 on the following page.

Table 4.3 A 15 × 10 FJSP instance with total flexibility

		M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
J1	O1,1	1	4	6	9	3	5	2	8	9	4
	O1,2	1	1	3	4	8	10	4	11	4	3
	O1,3	2	5	1	5	6	9	5	10	3	2
	O1,4	10	4	5	9	8	4	15	8	4	4
J2	O2,1	4	8	7	1	9	6	1	10	7	1
	O2,2	6	11	2	7	5	3	5	14	9	2
	O2,3	8	5	8	9	4	3	5	3	8	1
	O2,4	9	3	6	1	2	6	4	1	7	2
J3	O3,1	7	1	8	5	4	9	1	2	3	4
	O3,2	5	10	6	4	9	5	1	7	1	6
	O3,3	4	2	3	8	7	4	6	9	8	4
	O3,4	7	3	12	1	6	5	8	3	5	2
J4	O4,1	6	2	5	4	1	2	3	6	5	4
	O4,2	8	5	7	4	1	2	36	5	8	5
	O4,3	9	6	2	4	5	1	3	6	5	2
	O4,4	11	4	5	6	2	7	5	4	2	1
J5	O5,1	6	9	2	3	5	8	7	4	1	2
	O5,2	5	4	6	3	5	2	28	7	4	5
	O5,3	6	2	4	3	6	5	2	4	7	9
	O5,4	6	5	4	2	3	2	5	4	7	5
J6	O6,1	4	1	3	2	6	9	8	5	4	2
	O6,2	1	3	6	5	4	7	5	4	6	5
J7	O7,1	1	4	2	5	3	6	9	8	5	4
	O7,2	2	1	4	5	2	3	5	4	2	5
J8	O8,1	2	3	6	2	5	4	1	5	8	7
	O8,2	4	5	6	2	3	5	4	1	2	5
	O8,3	3	5	4	2	5	49	8	5	4	5
	O8,4	1	2	36	5	2	3	6	4	11	2
J9	O9,1	6	3	2	22	44	11	10	23	5	1
	O9,2	2	3	2	12	15	10	12	14	18	16
	O9,3	20	17	12	5	9	6	4	7	5	6
	O9,4	9	8	7	4	5	8	7	4	56	2
J10	O10,1	5	8	7	4	56	3	2	5	4	1
	O10,2	2	5	6	9	8	5	4	2	5	4
	O10,3	6	3	2	5	4	7	4	5	2	1
	O10,4	3	2	5	6	5	8	7	4	5	2
J11	O11,1	1	2	3	6	5	2	1	4	2	1
	O11,2	2	3	6	3	2	1	4	10	12	1
	O11,3	3	6	2	5	8	4	6	3	2	5
	O11,4	4	1	45	6	2	4	1	25	2	4
J12	O12,1	9	8	5	6	3	6	5	2	4	2
	O12,2	5	8	9	5	4	75	63	6	5	21
	O12,3	12	5	4	6	3	2	5	4	2	5
	O12,4	8	7	9	5	6	3	2	5	8	4
J13	O13,1	4	2	5	6	8	5	6	4	6	2
	O13,2	3	5	4	7	5	8	6	6	3	2
	O13,3	5	4	5	8	5	4	6	5	4	2
	O13,4	3	2	5	6	5	4	8	5	6	4

Table 4.3. (cont'd)

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M1
J14	O14,1	2	3	5	4	6	5	4	85	4	5
	O14,2	6	2	4	5	8	6	5	4	2	6
	O14,3	3	25	4	8	5	6	3	2	5	4
	O14,4	8	5	6	4	2	3	6	8	5	4
J15	O15,1	2	5	6	8	5	6	3	2	5	4
	O15,2	5	6	2	5	4	2	5	3	2	5
	O15,3	4	5	2	3	5	2	8	4	7	5
	O15,4	6	2	11	14	2	3	6	5	4	8

4.1.1. Pareto fronts of benchmark problem instances

The Pareto fronts of all three instances, established by [26] and [10], are shown in Table 4.4. Each row is a set of objective values obtained from a solution. Our AdRep schedulers are expected to produce *all* Pareto-optimal results in the Pareto set for a problem instance in each run. Prior to this work, only [10] has captured the whole Pareto front, but did not report the frequency of capture.

Table 4.4. Pareto Fronts of Benchmark Problem Instances

Problem Instance	Solutions		
	C_M	W_M	W_T
8x8	14	12	77
	15	12	75
	16	11	77
	16	13	73
10x10	7	5	43
	7	6	42
	8	5	42
	8	7	41
15x10	11	10	93
	11	11	91

4.2. *Design of Experiments*

AdRep is implemented in the Open Beagle framework (http://beagle.gel.ulaval.ca/). Open Beagle is an open-source multi-platform object-oriented C++ framework for Evolutionary Computation (EC). With Open Beagle, the user builds an EC system by creating an XML structure, calling on provided EC operators. The user only needs to implement primarily components that are not widely in use, usually including the evaluation operator. Open Beagle is compliant with the C++ ANSI/ISO 3 standard. As long as the user-created components are also compliant with the C++ ANSI/ISO 3 standard, the framework can used across platforms. The experiments in this chapter and the next chapter were run in a Linux environment. More information on the Open Beagle framework can be found in [43].

Table 4.5. Experiment Parameters

Problem	**Description**	**8×8**	**10×10**	**15×10**
Populations	Size of population	3 × 200	3 × 800	3 × 800
Max Generation	Maximum generation termination criteria	100	1,000	2,000
N	Number of scheduling aggregates	7		
Resolution	*Resolution* of threshold allele	0.1		
p_{cross}	Probability of crossover	0.5	0.8	
p_{ind}	Probability that an individual will be mutated	0.3	0.4	
p_{route}	Probability that a mutation will occur to a routing allele	0.4	0.1	
p_{agg}	Probability that a mutation will occur to a scheduling aggregate	0.5	0.3	
p_{swap}	Probability that a scheduling aggregate will be swapped	0.3	0.4	
$p_{threshold}$	Probability that threshold value allele will be mutated	0.7		
n_{mig}	Number of individuals that will migrate	1		
gen_{mig}	Interval of migration	10		
p_{Local}	Probability that an individual will be performed local search	0.0	0.001	
d_{Local}	Local search depth	0	50	

Table 4.5 contains the experiment's parameters. These parameters are derived from preliminary experiments that showed the most reliable results. Each problem instance was run 100 times. The number of generations for the AdRep scheduler to reach the best published makespan and number of generations to establish the published Pareto front are observed. The run terminates when the maximum generation is reached.

Since dimensionality of routing policy can greatly affect the performance of the AdRep scheduler and rescheduler, we limited the rule pairs to 8 of a possible 12, eliminating the pairs with greatest similarity (runtime and runtime, workload and workload). Table 4.6 contains the pairings that were used in this chapter and the next. Descriptions of the routing rules can be found in Table 3.1.

Table 4.6. Routing Rule pairs used

Initial Rule	Tiebreaking Rule
EF	SW
EF	LPC
SR	SW
SR	LPC
SW	EF
SW	SR
LPC	EF
LPC	SR

4.3. *Results*

For problem instances 8×8 and 15×10, all experiment runs reached the published Pareto front. For the 10×10 instance, 94 out of 100 runs reached the published Pareto front, and the other six failed to find one point on the front: an individual with (C_M=7, W_M=5, W_T=43). The results are recorded in Table 4.7. It is noteworthy that AdRep was successful in establishing the entire Pareto Front in 98% of all runs made.

Table 4.7. Experiment Results

Problem Instance	Generations to Pareto Front			
	Minimum	Maximum	Mean	Std
8×8	5	57	21.63	9.03
10×10	23	857	240.32	176.5
15×10	100	1923	683.24	418.44

Figure 4.1 contains a schedule derived from an AdRep solution for the 15×10 problem instance. It has C_M = 11, W_M = 11 and W_T = 91, which make it a member of the published Pareto front.

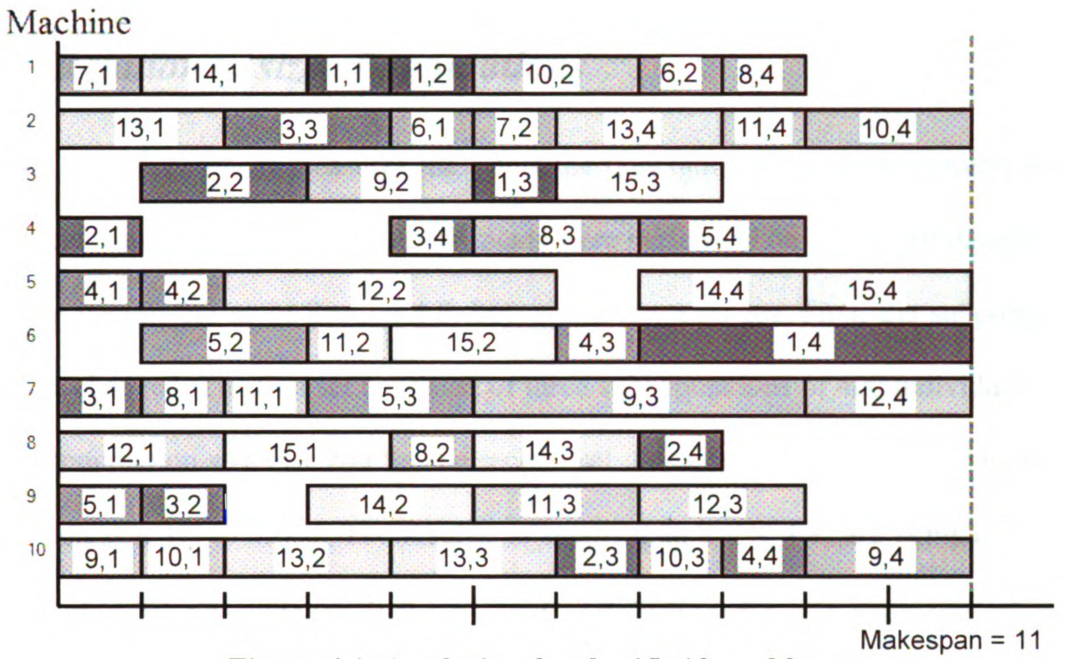

Figure 4.1. A solution for the 15×10 problem

Although the AdRep scheduler did reach the published Pareto front in almost all runs, there is some concern about its scalability, because of the procedures used to obtain this level of robustness of search. It took only about 8 seconds to finish an 8×8 run, however, it took approximately 8 minutes and 32 minutes to finish 10×10 and 15×10 runs, respectively. Even taking into account population size and maximum number of generations, there is still apparently exponential growth in required runtime.

4.4. *Relation to size of populations*

The next experiment measures the effect the population size has on convergence to the Pareto front. Two new sets of experiments were performed on the 15×10 instance. We used the parameters as in Section 4.2, but with a few changes. First, the subpopulation size was changed. The first set consisted of three subpopulations of 400 individuals each. The subpopulation size is 1,200 for the second set. Also, for the 3×400 trials, maximum generation was set to 4,000. The results, as reported in Table 4.8, are compared below with those of the 800 subpopulation size reported in Section 4.3.

Table 4.8. Experiment Results for 15×10 FJSP instance with various population size

Population	Number of successful trials	Mean Generations to Pareto Front	Standard Deviation
3×400	98	1,408.27	825.92
3×800	100	683.34	418.44
3×1,200	100	510.19	278.61

Two of the trials with the 3×400 subpopulations reached maximum generation before finding the last solution point on the published Pareto front. This behavior has been observed before when the population is not large enough for AdRep to converge reliably to the front. The Wilcoxon Rank-sum test between the 3×800 set and the 3×1200 set yielded a two-sided *P*-value of 0.0053. Therefore, the numbers of generations required for these two sets to reach the Pareto front are statistically significantly different. However, each generation with a population size of 3×1200 performs 50% more evaluations than does a generation with a population size of 3×800. Therefore, in terms of number of evaluations performed, the population size of 3×800 requires fewer evaluations to reach the Pareto front than does the population size of 3×1200. One can see that, although the variability in number of generations required with a population size

of 3×800 is about 30% higher, in terms of evaluations, the variability is not nearly so high. The mean number of evaluations required to reach the front is about 10% lower for the case of 3×800, and it is still able to find the Pareto front in 100% of the cases tested. Therefore, depending on the parallelization scheme used (the number of evaluations that are done in parallel), the best choice for population size may change. For example, if all evaluations are run on a serial machine, or one population is run on each processor, then the population size of 3×800 will generally produce the Pareto front sooner. However, in a massively parallel environment in which each individual to be evaluated in each population is sent to a separate processor for simultaneous evaluation, and assuming synchronization at the end of each generation, the population size of 3×1200 will produce the Pareto front sooner.

4.5. Summary

In this chapter, we applied our scheduling approach, using an adaptive representation, to static FJSP instances. Our approach, AdRep, uses just-in-time routing heuristics with adaptive scheduling to take into account the current state of the scheduling environment, in order to make better routing and scheduling decisions.

We tested our approach against three published FJSP instances, with the goal of reaching the published Pareto front of each problem instance, proving the ability of our approach to discover the best-so-far solutions of these problem instances. AdRep succeeded in reaching the target Pareto fronts in 98% of all runs (100% for each of two cases, and 94% for the other). However, it suffered somewhat in terms of scalability. As the size of problem increased, the resource and runtime requirements increased at a faster rate.

In Section 4.4, we tested AdRep with various population sizes, one larger than the former test, the other smaller. We found two instances in which the trial with the smaller population size did not converge to the known Pareto front. Although the larger population size did decrease generations required to converge, the number of evaluations required for each generation increased. Therefore, it is evident that, depending on the level and type of parallelization available for the problem, the ideal population size may change.

AdRep, although functional in creating schedules in a static environment, is mainly designed for rescheduling. In the next chapter, we will test the ability of AdRep to repair schedules with machine-unavailability disruptions.

CHAPTER 5

PERFORMANCE ANALYSIS —DISRUPTION-PRONE FLEXIBLE JOB-SHOP SCHEDULING PROBLEMS

In this chapter, the effectiveness of AdRep in regards to rescheduling will be tested. Two benchmark algorithms will be employed for comparison: a right-shift rescheduler and a prescheduler, which rebuilds schedules from the start at each disruption scenario.

Section 5.1 contains the design of experiments. Section 5.2 discusses the test cases used in the experiments. Section 5.3 describes the two benchmark algorithms. The results of the experiments are recorded in Section 5.4. Section 5.5 concludes the chapter.

5.1 Design of Experiments

As mentioned in Section 3.2.1, the following assumptions are made in regards to the nature of the rescheduling problem under consideration:

- The disruption pattern is unknown to the AdRep rescheduler. Only reactive repair will be perform to cope with the disruptions.
- A disruption has a predetermined end time, known to the rescheduler at the start of the disruption (e.g., it is possible to estimate how long it will take to repair or return a given machine to service, once it has broken down).
- There is no partial completion credit. If an operation is interrupted by a disruption, it must start anew, either on the same machine or another, running for the whole runtime cost as specified in the problem instance.

In this dissertation, a scenario set will contain 10 disruption scenarios. Two objectives will be the targets of optimization; (1) *initial makespan*, makespan derived from a specified disruption-free FSJP instance, and (2) *maximum repaired makespan*, the highest makespan of the ten repaired schedules from the ten scenarios in the scenario set. Two helper objectives will be used, *average repair makespan* and *duplication count*.

Table 5.1 contains the parameters for the experiment. Note that local search was not used for AdRep in this experiment.

Table 5.1. AdRep parameters for dynamic case

Problem	**10×10**	**15×10**
Populations	2×500	3×500
Max Generation	300	
N	7	
p_{cross}	0.8	
p_{ind}	0.3	
p_{route}	0.1	
p_{agg}	0.3	
p_{swap}	0.4	
$p_{threshold}$	0.7	
n_{mig}	1	

5.2 *Test Cases*

To test the effectiveness of AdRep in rescheduling, we used a disruption scenario set based on 10×10 and 15×10 problem instances from Chapter 4. Two types of scenarios were used: the "small" set contained disruptions with smaller duration, but occurring more often, while the "large" set had disruptions with longer durations but less frequent occurrence. Fifty scenarios sets were created per disruption type per problem instance, making 200 problem instances in total. Table 5.2 contains the characteristic parameters of the test cases.

Table 5.2. Characteristics of the Disruption Test Cases

Problem Instance	Best Initial Makespan	Disruption Characteristic	Disruption Duration	Disruptions per Scenario
10×10	7	Small	1 – 2	4 – 6
		Large	3 – 4	1 – 2
15×10	11	Small	1 – 3	4 – 6
		Large	5 – 8	1 – 2

AdRep was run 100 times against randomly generated problem instances from each disruption type from each problem instance. At each generation, a new set of test scenarios was generated stochastically, based on the characteristics provided, so that the GA cannot overfit a fixed set of scenarios. The best individuals were tested against the 50 pre-generated test cases. Best results from each run were recorded.

5.3 *Benchmark Algorithms*

Two rescheduling approaches are used as benchmarks to AdRep rescheduling: a Right-shifting (RS) rescheduler and a (non-causal) Prescheduler (PS) rescheduler. The Prescheduler approach is providing a limiting case in the following sense: it "knows" about the breakdowns in the scenario at the beginning of the run, so is essentially scheduling them as additional, pre-planned operations. Therefore, a "reactive" scheduler using the same scheduling approach would not be expected to match the behavior of the Prescheduler, which provides an ideal against which a scheduler without advance information about breakdowns can be compared. (Below, the PS will be compared against the AdRep scheduler, which uses a different approach, so the PS does not necessarily bound the AdRep performance.) Both RS and PS schedulers used fixed machine assignments for routing and permutation with repetition for scheduling. By utilizing the GT algorithm, the scheduling chromosomes are used to decide which operation in the conflict set is to be scheduled first, by finding the leftmost unused allele that corresponds to one of the jobs that one of the operations in the conflict set belongs to.

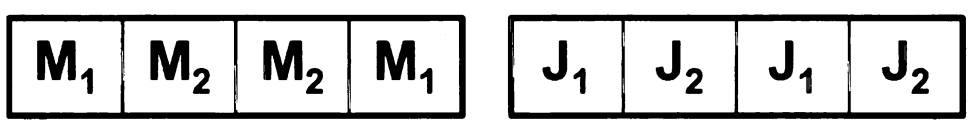

Figure 5.1. Chromosome used by benchmark algorithm

Figure 5.1 shows a chromosome of an individual from a 2×2 FJSP instance with 4 operations. The left chromosome is the routing chromosome, while the right one is the scheduling chromosome. From this pair of chromosomes, we know that, for example, the operation $O_{1,2}$ is assigned to machine 2. And, in the scheduling process, if there is a

conflict in the first iteration of the scheduling, operation $O_{1,2}$ will be selected over operation $O_{2,1}$, based on the value of the first allele in the scheduling chromosome.

Table 5.3. Evolution Operators for Benchmark Chromosome

	Routing	**Scheduling**
Initialization	Random	Shuffle
Crossover	Uniform	Modified Order
Mutation	Uniform	Swap

Table 5.3 contains some operators used in GA evolution for the chromosomes used by the two benchmark algorithms. Shuffle initialization starts with generating an ordered array with repetition, with number of repetitions for each value equal to the number of operations in the corresponding job. For example, if there are 3 operations in job 2, there must be 3 alleles with value 2. Then the array is shuffled, swapping allele values randomly.

Modified order crossover is based on the order crossover [44] operator commonly used with a permutation representation. First, half of a two-point crossover is performed on one parent, copying part of a chromosome to a child. Then, the operator takes note of how many entries are still needed for each value, and the order in which they appear is taken from the other parent and filled into the rest of the chromosome. Figure 5.2 illustrates how Modified order crossover works. The gray allele values are taken from the first parent, leaving one of each of the values missing. The underlined alleles in the second parent are the ones from which the left of the child's chromosome (the white alleles) takes their ordering.

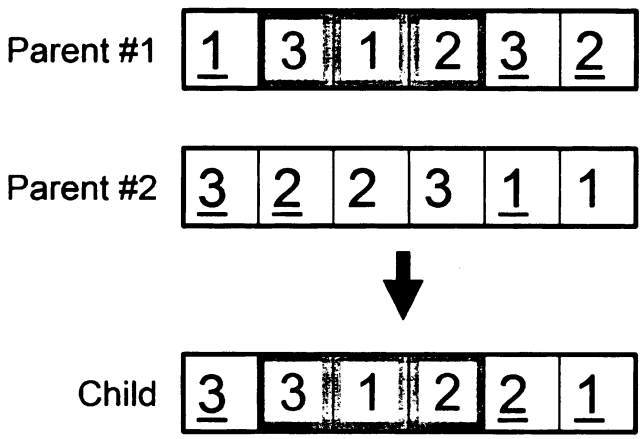

Figure 5.2. Modified Order Crossover

The non-dominant sorting algorithm from NSGA-II is used for the replacement strategy, and random selection decides which individuals are selected for breeding. Also, hillclimbing local search is employed to speed up the convergence.

The two benchmark methods utilize the same representation and are evolved in the same framework. The difference between the two methods is how they deal with disruptions.

5.3.1 Right-shift Rescheduler

For the right-shift (RS) rescheduler, the rescheduler starts from a disruption-free FJSP instance using the GT algorithm to create an initial schedule. In a disruption scenario, when a disruption occurs, the rescheduler performs the affected operation rescheduling [45]: it right-shifts directly the affected operations, delaying them until the disruption is over, then the rescheduler checks the affected operations' job and machine successors, right-shifting them if they are affected by their predecessors' updated finishing times. After a scenario is handled, the repaired makespan is recorded and the schedule is reset to being an initial schedule. The RS rescheduler then moves on to the next disruption

scenario in the set. C_{RMax} is collected the same way as in the AdRep rescheduler, from the scenario that inflicts the highest repair makespan. Also, like AdRep, the single RS rescheduler handles every scenario in the set.

For each disruption characteristic, the RS rescheduler is run 100 times. The ranges of best (lowest) maximum repaired makespans found in each run, along with the mean values, are reported in Table 5.6.

5.3.2 Prescheduler

For the prescheduler, each rescheduler works with only one scenario at a time, instead of a set of them as in the AdRep and right-shift reschedulers. The rescheduler treats all disruptions as *predetermined* downtime; that is, all disruptions are known to the rescheduler at the time of the initial scheduling. In a sense, the prescheduler performs *complete rescheduling*, evolving a new schedule to suit each disruption scenario. Operating as it does with full foreknowledge of all breakdowns, one would expect that its results would act as a lower bound on the disrupted makespans of any scheduler that does not make use of this foreknowledge of the breakdowns. However, we shall see that because the prescheduler is not using as powerful a representation and search operator pair, even its foreknowledge will not necessarily allow it to outperform the AdRep scheduler in coping with disruptions. Figure 5.3 illustrates how preschedulers work with a set of disruption scenarios.

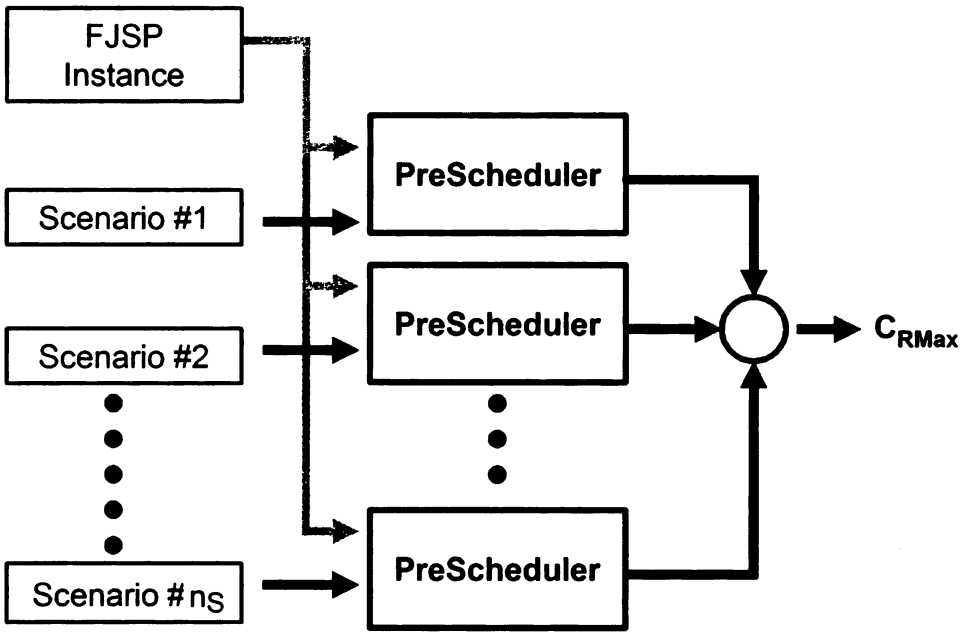

Figure 5.3. Prescheduling a set of scenarios

The prescheduler evolves very much like a static scheduler, only with predetermined periods in which it cannot schedule an operation. It uses the GT algorithm for creating a schedule, with an extra mechanism added in so that, when the finish time for each operation is calculated before the conflict set is computed, it delays any operation that cannot start at an available time before the disruption without having overlap, which is very similar to avoiding a scheduled downtime.

5.3.3 Experiment Parameters for the Benchmark Algorithms

AdRep and the RS Rescheduler are each run 100 times for each disruption characteristic and problem instance. The PS scheduler evolves a schedule directly for each individual scenario among the test cases, keeping the best of three runs for each scenario. The range of best maximum repaired makespan is reported in Table 5.6. The experiment parameters, as described in Table 5.4, are shown in Table 5.5.

Table 5.4. Description of Benchmark Experiment Parameters

Parameters	Description
Populations	Size of Population
Max Generation	Max Generation Termination Criteria
p_{cross}	Probability of crossover
p_{ind}	Probability of an individual to be mutated
p_{int}	Probability of an allele to be mutated
n_{mig}	Number of individuals that will migrate
p_{Local}	Probability of individual local search
d_{Local}	Local search depth

Table 5.5. Experiment Parameters for Benchmark Algorithm

Parameters	RS		PS	
	10×10	15x10	10x10	15x10
Populations	2 x 500	3 x 500	2 x 500	3 x 800
Max Generation	300			
p_{cross}	0.8			
p_{ind}	0.4			
p_{int}	0.1			
n_{mig}	1			
p_{Local}	0.01			
d_{Local}	50			

5.4 *Results*

The results of the experimentation comparing the AdRep and the two benchmark algorithms on scenarios with disruption are reported in Table 5.6.

Table 5.6. Results of the Rescheduling Experimentation

Problem Instance	Best Initial Makespan	Disruption Characteristic	RS Makespan		PS Makespan	AdRep Makespan	
			Range	Average		Range	Average
10×10	7	Small	11 – 13	11.15	9	10 – 14	10.9
		Large	12 – 13	12.05	9	10 – 13	10.79
15×10	11	Small	15 – 19	16.53	15	15 – 20	17.13
		Large	19 – 21	19.95	(16)	15 – 22	17.34

AdRep surpassed the RS rescheduler in most test cases, performing worse for only one disruption type on one problem. Almost all best AdRep solutions also obtained the best initial makespan when run without disruption. The exceptions were 11 runs (of 100) of the large disruption characteristic on the 15×10 problem instance and 3 runs (of 100) of the small disruption characteristic on the same problem instance. In these cases, the solution with the best repaired makespan produced the initial makespan of 12.

Unexpectedly, the PS rescheduler did not match AdRep in some runs of the large disruption characteristic for 15×10 instance, resulting in worse maximum repaired makespans, *even though it was scheduling with foreknowledge of the disruptions*. That result is enclosed in parentheses in the table. Further investigation revealed that the individuals produced by AdRep that surpassed the PS rescheduler suffered from larger than usual total workloads, which might be a reason the PS rescheduler had difficulty reaching that solution.

Figure 5.4 shows a schedule created by an individual obtained from an AdRep run. The repaired schedule by the same individual when a disruption scenario containing

disruptions at Machine 3 from time 3 to time 9, and on machine 10 from time 5 to time 12, is depicted in Figure 5.5.

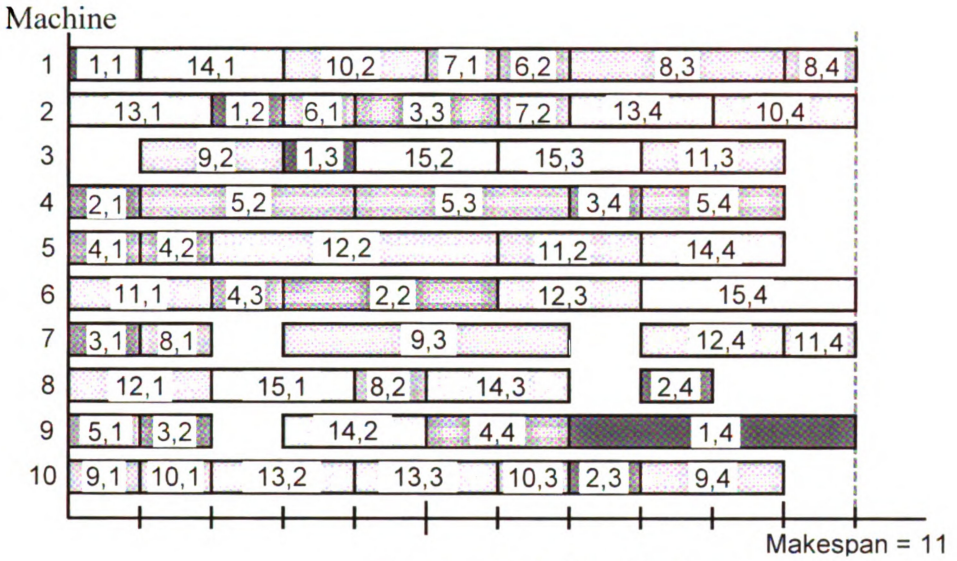

Figure 5.4. A C_M=11 Schedule

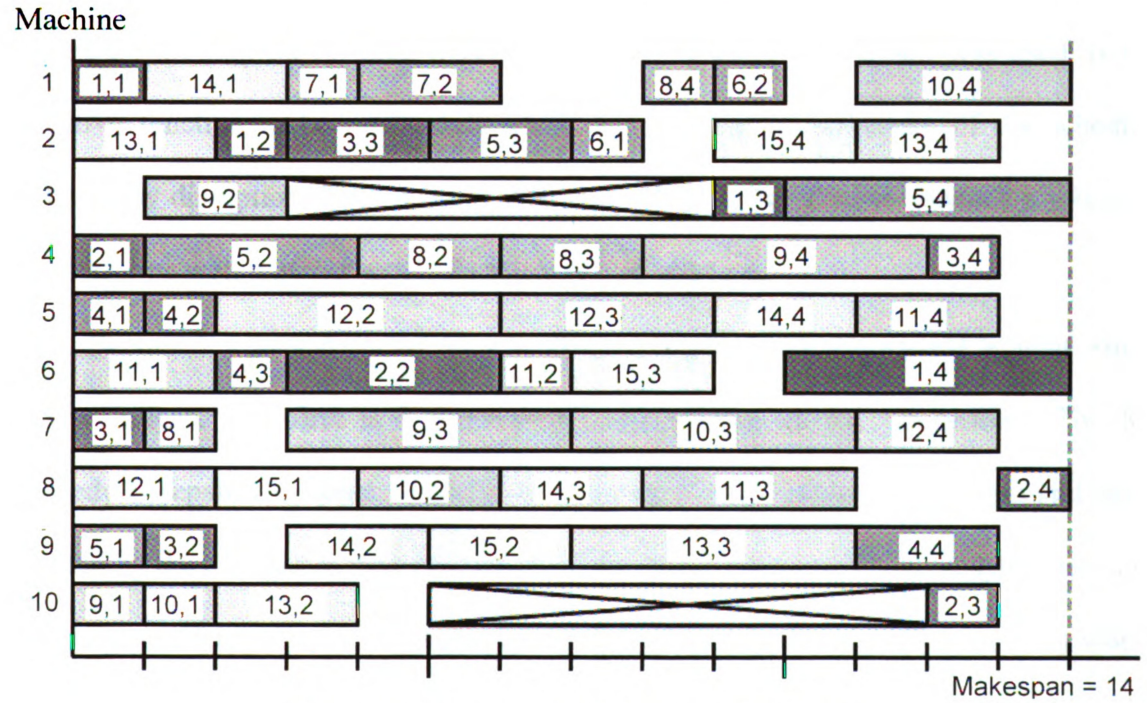

Figure 5.5. A C_{RMax}=14 Repaired Schedule

5.5 *Summary*

In this chapter, we investigated the performance of AdRep in terms of rescheduling from machine-unavailability disruption. Using a Routing GT algorithm, we can often repair the schedule by reassigning some affected operations (including ones scheduled to run later on the disrupted machine) away from the disrupted machine, depending on their routing policies.

To test how effective AdRep is in rescheduling, it was run against multiple disruption scenario sets, each a collection of breakdown scenarios. AdRep was evolving against multiple scenarios in a particular set at the same time—at each generation, a new set of scenarios was generated, reducing the likelihood of overfitting. Maximum repair makespan, the highest repaired makespan for any scenario in the set, was used as the objective function to be minimized, along with the initial makespan of the schedule without any disruption. That is, an AdRep individual is expected to produce a schedule that performs well whether *or not* there are any disruptions.

In Section 5.3 we introduced two benchmark rescheduling algorithms against which to test AdRep. The first is a Right-shifter rescheduler, or RS rescheduler. The RS rescheduler repairs disrupted schedules by delaying directly affected operations and those affected by the changes in their predecessors' finishing times, until the disruption is over.

The second benchmark algorithm is the prescheduler. The prescheduler (PS) works by treating each breakdown scenario as predetermined downtime, evolving to deal with each of them separately. By gaining disruption knowledge in advance and being allowed to evolve against each scenario separately, the PS algorithm is given a major advantage in competing against AdRep.

Section 5.4 contains the results of the experiments. AdRep performed noticeably better than the RS algorithm in most of the trials. However, the fact that the PS rescheduler performed worse in one case than AdRep, taking into account the advantage of foreknowledge of failures that the PS has, suggested that the PS algorithm suffers from some limitations relative to the AdRep algorithm in terms of exploring the search space.

CHAPTER 6

CONCLUSION

FJSP adds another layer of complexity into JSSP by the need to determine machine allocation to operations, in a process known as routing. This makes the already difficult task of scheduling a JSSP instance even more challenging. In this dissertation, we have developed a novel way to represent an FJSP solution, providing a new way to assign machines to operations and a more flexible scheduling component. Our approach has proven to be able to replicate the set of best published solutions of popular problem instances, and to do so reliably in each run of the algorithm. Although it suffers somewhat in term of scalability in the static case, the rescheduling results in the disruption-prone dynamic cases have been promising.

6.1. Summary

In this dissertation, our main objective has been to create a representation and genetic operators that are conducive to rescheduling, especially in the case of machine unavailability due to disruptions. However, the solutions should also perform well in the static (undisrupted) case. Our representation, Adaptive Representation (AdRep), consists of a routing component and an adaptive scheduling component. The routing component assigns a simple routing policy to an operation instead of directly assigning a machine to it. This allows the operation to be rerouted to another machine in case the scheduling environment has changed so drastically that the current machine is no longer suitable, while, in the static case, it enables the operation to be assigned to a viable machine in the

first place. The adaptive scheduling components use the current state of the system to direct the routing GT algorithm to select an operation to be scheduled.

To make sure AdRep performs well in the static FJSP, we have tested it against 3 published instances. These three, of varying sizes and dimensionalities, have been used in various experiments. The Pareto fronts of all three problems are well known, having been discovered (often by different authors) in work reported in the literature. In reliably reaching all of the points on these Pareto fronts, AdRep has shown that it is viable in the static case. The experiments have shown that, while suffering somewhat from scaling with increases in problem size, AdRep did succeed in reaching the Pareto front for almost all of the benchmark problems. We then tested AdRep with varying sizes of populations. The results showed that, while a decrease in population size can hurt the chance that the GA will locate the entire Pareto front, increasing the population size will eventually hit a point where, although the decrease in number of generations required to converge to the Pareto front still occurs, the increase in number of evaluations in each generation no longer supports a guarantee that a larger population size will enable faster convergence. Other factors, such as degree and method of parallelization, need to be taken into account in setting optimal population size.

To prepare AdRep for rescheduling, one need only create the partial schedule from the one disrupted by machine unavailability. By removing directly affected operations and those that start after the disruption, and preparing them to be rescheduled, they constitute a partial schedule that the AdRep rescheduler can use as a starting point. By re-executing the routing GT algorithm, rerouting operations as needed, a repaired schedule is created. This means that, in practice in a flexible job shop, when a disruption occurs,

the new schedule to cope with the disruption can be produced almost instantaneously, by running only the scheduling algorithm, without needing to perform any new search.

To test the performance of the AdRep rescheduler, two benchmark algorithms were used: a right-shift rescheduler, and a prescheduler. The right-shift rescheduler and AdRep were evolved against a set of disruption scenarios, while the prescheduler evolved against one scenario at a time. In this experiment, AdRep has shown that it can produce a good initial schedule that outcompetes the right-shifting rescheduler in most cases. It even managed to outperform the prescheduler in some cases, even with the prescheduler's foreknowledge advantage.

6.2. *Contributions*

The main contributions of this dissertation are summarized as followed:

(1) In this dissertation, we have proposed a novel alternative to represent an FJSP solution. Also, we have put forward a simple scheme to assign machines to operations. By allotting a simple routing policy to each operation, instead of assigning them a machine outright, we can retain some flexibility and can gain some information about the schedule environment at the time that the operation is about to be scheduled, improving routing decisions.

(2) Furthermore, AdRep can be used as a measure to reschedule an FJSP schedule that suffers from disruption to machine unavailability. Since AdRep's repair mechanism is inherent in the chromosome itself, the disrupted schedule can be repaired just by rescheduling from the disruption starting time using the chromosome that it used for scheduling. This allows rerouting as needed, improving the quality of the repaired schedule.

(3) We also performed a performance analysis, confirming the ability of AdRep to match the published result in term of Pareto front in most cases. Also, we have tested AdRep against two benchmark rescheduling algorithm. The results prove the superiority of AdRep to produce repaired makespan, in comparison to right-shifting rescheduling. Furthermore, AdRep has been shown to perform not much worse than complete rescheduling.

(4) The work in this dissertation has resulted in two papers published to date. The author has presented "Adaptive representation for flexible job-shop scheduling and rescheduling", which entails a makespan-focused static FJSP scheduler and

dynamic FJSP rescheduler, at the 2009 World Summit on Genetic and Evolutionary (GECS) Computation at Shanghai, China, and published in the proceeding of that conference. The paper, "Solving Multiobjective Flexible Job-shop Scheduling Using an Adaptive Representation", which focused on the multiobjective static FJSP, was accepted as full paper, published, and presented in July, 2010, at the Genetic and Evolutionary Computation Conference 2010 in Portland, Oregon.

6.3. *Future Work*

(1) In static FJSP, although AdRep can achieve all best published results on the benchmark problems nearly always, its convergence efficiency suffers considerably when the size of the problem increases. An effort to improve AdRep efficiency should be worthwhile—for example, devising an effective local search operator for AdRep. A neighborhood structure will need to be analyzed for a meaningful neighborhood traversal to be done.

(2) There is still a considerable distance between FJSP and real-world scheduling applications. The following factors should be taken into account:

- Set-up costs to prepare a machine for an operation. This also includes the cost of reassigning an operation to another machine. If such a cost exists, a routing rule should take it into account. The number of rerouted operations can be used as an objective, for example.
- Deadlines. Soft deadlines can be handled by making tardiness one of the objectives. Hard deadlines, on the other hand, might require some modification to the scheduling system itself, limiting machine choices if a deadline is to be breached and giving priority to near-deadline operations to be scheduled first.
- Additions to the system such as new jobs or new machine arrivals. Adapting to added machines should not be difficult, providing the characteristics of the machine are known. Adapting to job arrivals, or instances where job number is not set, can be much more difficult. Perhaps jobs can be divided into

categories and AdRep can be based on operations in a job category instead of a particular job.

(3) Improving the AdRep data structure. Although AdRep is currently quite effective in creating a good scheduling solution, especially on the rescheduling capacity, a more efficient and more expressive data structure would be advantageous. As it is, routing components constitute a solid majority of the chromosome, since there is one allele per operation. A more compact routing component should improve efficiency, but care must be taken not to go too far, or performance will be lost. The current adaptive scheduling component might, however, be too restrictive. A more expressive structure could help in scheduling performance. One suggestion is to implement AdRep in two program trees, one for routing and one for scheduling. Coevolutionary GP could then be used to improve them.

REFERENCES

REFERENCES

[1] D. Goldberg, *Genetic algorithms in search, optimization, and machine learning*, Addison-Wesley, 1989. ISBN: 0201157675

[2] I. Kacem, S. Hammadi, and P. Borne, "Approach by Localization and Multiobjective Evolutionary Optimization for Flexible Job-Shop Scheduling Problems", *Systems, Man and Cybernetics, Part C, IEEE Transactions on*, 2002.

[3] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II", *IEEE Transaction on Evolutionary Computation*, Vol. 6, No. 2, 182-197, 2002.

[4] P. Brucker , and R. Schlie, "Job-Shop Scheduling with Multi-Purpose Machines", *Computing*, Vol. 45, No. 4, 369-375, 1990.

[5] A. Jain and S. Meeran, "Deterministic Job-Shop Scheduling: Past, Present and Future", *European Journal of Operational Research*, 1999.

[6] E. Hart, P. Ross and D. Corne, "Evolutionary Scheduling: A Review", *Genetic Programming and Evolvable Machines*, 2005

[7] M.R. Garey, D.S. Johnson, and R. Sethi, "The Complexity of Flow Shop and Job Shop Scheduling", *Mathematics of Operations Research*, Vol. 1, No. 2, May, 1976.

[8] N. Ho and J. Tay, "GENACE: An Efficient Cultural Algorithm for Solving the Flexible Job-Shop Problem", *Proceedings of the Congress on Evolutionary Computation*, Vol. 2, p.1759, 2004.

[9] N. Ho, J. Tay, and E. Lai, "An Effective Architecture for Learning and Evolving Flexible Job-Shop Schedules", *European Journal of Operational Research*, Vol. 179, 316-333, 2007.

[10] N. Ho, and J. Tay, "Using Evolutionary Computation and Local Search to Solve Multi-Objective Flexible Job Shop Problems", *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, 821-828. 2007.

[11] W. Xia ,and Z. Wu, "An Effective Hybrid Optimization Approach for Multi-Objective Flexible Job-Shop Scheduling Problems", *Computers & Industrial Engineering*, Vol. 48, No. 2, 409-425, 2005.

[12] J. Grobler, A. Engelbrecht, S. Kok, and S. Yadavalli, "Metaheuristics for the Multi-Objective FJSP with Sequence-Dependent Set-Up Times, Auxiliary Resources and Machine Down Time", *Annals of Operations Research*, 2009.

[13] F. Pezzella et al, "A Genetic Algorithm for the Flexible Job-shop Scheduling Problem", *Computer & Operation Research* , Vol. 35, No. 10, 3202-3212, 2008.

[14] S.S. Panwalkar and W. Iskander, "A Survey of Scheduling Rules", *Operations Research*, 1977.

[15] S.C. Park, N. Raman, and M.J. Shaw, "Adaptive Scheduling in Dynamic Flexible Manufacturing Systems: A Dynamic Rule Selection Approach", *Robotics and Automation*, IEEE Transactions on, 1997.

[16] B. Giffler and G.L. Thompson, "Algorithms for Solving Production-Scheduling Problems", *Operations Research*, 1960.

[17] P. Brandimarte, "Routing and Scheduling in a Flexible Job Shop by Tabu Search", *Annals of Operations Research*, Vol. 41, No. 3, 157-183, 1993.

[18] J. Hurink, B. Jurisch, and M. Thole, "Tabu Search for the Job-Shop Scheduling Problem with Multi-Purpose Machines", *OR Spectrum*, Vol.15, No.4, 205-215, 1994.

[19] J. Chambers, J.W. Barnes, "Reactive Search for Flexible Job Shop Scheduling", *Technical Report ORP98-04*, The University of Texas, Austin, October 1998.

[20] M. Mastrolilli, and L. Gambardella, 2000, "Effective Neighbourhood Functions for the Flexible Job Shop Problem", *Journal of Scheduling*, Vol. 3, Issue 1, 3-20, 2000.

[21] J. Gao, M. Gen, and L. Sun "A Hybrid Genetic and Variable Neighborhood Descent Algorithm for Flexible Job Shop Scheduling Problems", *Computers & Operations Research,* 2892 – 2907, 2008.

[22] K. Mesghouni, S. Hammadi and P. Borne, "Evolution Programs for Job-Shop Scheduling", *IEEE International Conference on Systems, Man, and Cybernetics (SMC '97)*, 1997.

[23] H. Chen, J. Ihlow and C. Lehman, "A Genetic Algorithm for Flexible Job-Shop Scheduling", *Proceeding of the 1999 IEEE International Conference on Robotics & Automation*, 1999.

[24] H. Zhang, and M. Gen "Multistage-based genetic algorithm for flexible job-shop scheduling problem", *Journal of Complexity International*, Vol. 11, pp. 223, 2005.

[25] J. Tay and D. Wibowo, "An Effective Chromosome Representation for Evolving Flexible Job Shop Schedules", *Proceedings of AAAI Genetic and Evolutionary Computation*, 2004.

[26] J. Gao, M. Gen, and L. Sun, "A hybrid of genetic algorithm and bottleneck shifting for flexible job shop scheduling problem", *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, 2006.

[27] S.C. Lin, E.D. Goodman, and W.F. Punch, "A Genetic Algorithm Approach to Dynamic Job Shop Scheduling Problems", *Proceeding of Seventh International Conference on Genetic Algorithms*, 1997.

[28] T. Yamada and R. Nakano, "Genetic Algorithms for Job-Shop Scheduling Problems", *Proceedings of Modern Heuristic for Decision Support*, 1997.

[29] L.N. de Castro and J. Timmis, "Artificial Immune Systems: A Novel Paradigm to Pattern Recognition", *Artificial Neural Networks in Pattern Recognition*, 2002.

[30] Z.X. Ong, J.C. Tay, and C.K. Kwoh, "Applying the Clonal Selection Principle to Find Flexible Job-Shop Schedules", *Computers and Industrial Engineering*, Vol. 54, No. 3, 442-455, 2005.

[31] N. Liouane, I. Saad, S. Hammadi, and P. Borne, "Ant Systems and Local Search Optimization for Flexible Job Shop Scheduling Production", *International Journal of Computers Communications & Control*, Vol. 2, No. 2, 174-184, 2007.

[32] J. Tay, N. Ho, "Evolving Dispatching Rules Using Genetic Programming for Solving Multi-Objective Flexible Job-Shop Problems", *Computers & Industrial Engineering*, Vol. 54, Issue 3, 453-473, 2008.

[33] G.E. Vieira, J.W. Herrmann, and Ed. Lin, "Rescheduling Manufacturing Systems: A Framework of Strategies, Policies, and Methods", *Journal of Scheduling*, Vol. 6, No. 1, 39-62, 2003.

[34] V. Subramaniam, and A. Singh, "mAOR: A Heuristic-based Reactive Repair Mechanism for Job Shop Schedules", *International Journal of Advance Manufacturing Technology*, 2003.

[35] M Gholami, M Zandieh, "Integrating Simulation and Genetic Algorithm to Schedule a Dynamic Flexible Job Shop", *Journal of Intelligent Manufacturing*, Vol. 20, No. 4, 481-498, 2009.

[36] J. Gao, M. Gen, and L. Sun, "Scheduling Jobs and Maintenances in Flexible Job Shop with a Hybrid Genetic Algorithm", *Journal Of Intelligent Manufacturing*, Vol. 17, No. 4, 493-507, 2006.

[37] H. Ishibuchi, T. Yoshida, and T. Murata, "Balance between Genetic Search and Local Search in Memetic Algorithms for Multiobjective Permutation Flowshop Scheduling", *Evolutionary Computation, IEEE Transactions on*, Vol.7, No.2, 204-223, 2003

[38] I. Kacem, S. Hammadi, and P. Borne, "Pareto-Optimality Approach for Flexible Job-Shop Scheduling Problems: Hybridization of Evolutionary Algorithms and Fuzzy Logic", *Mathematics and Computers in Simulation*, Vol. 60, No. 3-5, 245-276, 2002.

[39] J. Schaffer, "Multiple Objective Optimization with Vector Evaluated Genetic Algorithms", *Proceedings of the 1st International Conference on Genetic Algorithms*, 93-100, 1985.

[40] E. Zitzler and L. Thiele, "Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 4, 257-271, 1999.

[41] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the Strength Pareto Evolutionary Algorithm", *Swiss Federal Institute of Technology*, TIK-Report, 2001.

[42] V. Subramaniam et al, "Machine Selection Rules in a Dynamic Job Shop", *The International Journal of Advanced Manufacturing*, 2000.

[43] C. Gagné, and M. Parizeau, "Genericity in Evolutionary Computation Software Tools: Principles and Case-Study", *International Journal on Artificial Intelligence Tools*, Vol. 15, No. 2, p. 173, 2006.

[44] C. Bierwirth, D. Mattfield and Herbert Kopfer, "On Permutation Representations for Scheduling Problems", *Lecture Notes in Computer Science*, Vol. 1141, 310-318, 1996

[45] R.J. Abumaizar, and J.A. Svestka, "Rescheduling Job Shops under Random Disruptions", *International Journal of Production Research*, 1997.