JOB SHOP OPTIMIZATION THROUGH MULTIPLE INDEPENDENT PARTICLE SWARMS

By

BRIAN IVERS

Bachelor of Science in Electrical Engineering

Oklahoma State University

Stillwater, OK

2003

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2006

JOB SHOP OPTIMIZATION THROUGH MULTIPLE

INDEPENDENT PARTICLE SWARMS

Thesis Approved:

Dr. Gary G. Yen
Thesis Advisor

Dr. Rafael Fierro

Dr. Carl D. Latino

A. Gordon Emslie
Dean of the Graduate College

ACKNOWLEDGEMENTS

I thank my advisor, Dr. Gary Yen, for having confidence in me throughout this process and for giving me the great opportunity to work with him. He has taught me how to think critically which will prove invaluable during my engineering career. Also, thank you to my family who has believed in me and encouraged me through this endeavor, especially my Mom for her love and support and my Dad who has *always* been my advisor. A special thanks to my Grandpa Ivers who I know will be sad to see me leave OSU and who I hope approves of this research. And finally, to my girlfriend and high school sweetheart, Louisa Kinder, for staying with me while I've worked toward my master's for the past 2½ years 500 miles away from her. Thank you so much.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

For several decades now much research has been conducted into the field of evolutionary computation, or meta-heuristic search techniques. These techniques are preferred for many optimization problems where the search space either very complex or very large. One of the ways researchers can measure the effectiveness of their designed algorithms is to test them on known optimization problems, such as the Job Shop Problem (JSP). Not only does the JSP serve as a benchmark to judge various algorithms by, but also as a model for which the field of scheduling theory can be advanced. Scheduling theory is studied by researchers in many fields and a direct application to today's industry. In the real world, scheduling problems like the JSP can be thought of as production process or production scheduling problems.

1.1 Current Need

It is fairly obvious that optimal production process is important to today's commercial industry. The online encyclopedia Wikipedia states that "scheduling (production process) is an important tool for manufacturing and engineering". This importance stems from the desire to lower production costs and increase profits. It is often said that time is money. While this general statement can't be proven for every situation, it is definitely assumed

to be true when it comes to business. It is in the best interest of both the company and the individual to use time efficiently. This is why it is important to study scheduling problems like the job shop problem. The efficient use of time means lower costs and higher profits. In many ways, our way of life, made possible by modern computer technology and networking, demands efficient scheduling. For example, simply consider the many postal or parcel services operating throughout the world, or air traffic control centers directing thousands and thousands of planes to safe ground. Indeed, optimal scheduling is a worthy area of research. The JSP is simply a helpful model to help aid in the eventual development of scheduling software for industrial, personal or commercial use. One only needs to perform a simple Google search to discover the many scheduling software packages available for both personal and commercial use. If good optimization methods can be developed for the scheduling models then perhaps they can have a beneficial impact on the current production process scheduling software, and thereby lower costs for industry.

Our ability to produce optimal schedules as engineers, scientists, employees and employers has not necessarily progressed with the advance in technology and processing power over the years. The amazing processing power of modern computers has definitely helped researches evaluate many different methods of scheduling at fast speeds, but in terms of finding a technique that *consistently* produces optimal schedules for many different sizes of problems has not yet been fully developed. Even the modern computer is hardly a match for the enormous combinatorial complexity of a medium scale Job Shop Problem (JSP). The Job Shop Problem belongs to the class of NP-Hard

problems, and is commonly thought of as one of the "harder" problems in that class. NP-Hard problems have an exponentially growing search space as the problem increases in dimension. For example, the search space (possible combinations) for a Job Shop Problem that consists of 10 jobs to be processed on 10 machines is $(10!)^{10}$. Which is equal to approximately 3.9594×10^{65} or 395,940,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000. Even though not all of these solutions are feasible, with this information we can see how even modern processors which perform billions of calculations per second are hardly a match for this kind of combinatorial space. Therefore, methods and algorithms must be developed to provide good search directions within this space for our modern computers to perform their calculations. This is where the great field of computational intelligence holds much promise. Where exact methods of optimization search out the best solution exhaustively by using mathematical formulations, methods of computational intelligence rely on certain heuristic principles and ideas to explore and then converge to the best found solution. Using computational intelligence computers are programmed to evolve or grow a solution over a period of iterations, usually using populations of solutions to accomplish this task. This population, or set of solutions to a given problem, are assigned certain measures and fitness values and allowed to interact over "time" in a competitive environment. These heuristics that seemingly magically govern such behavior come from observation our physical world, such as the way neurons work in our brain, or how species evolve over time, or even how schools of fish, flocks of birds, and colonies of ants move around space. Deciphering these amazingly tuned biological processes from mother nature and encoding them into computer programs presents quite

a challenge in most cases, since our natural world is governed by many delicate interrelated equations. Yet, our rudimentary translation of these principles into computer programs can produce great results, but leave much room for more improvement. With every personal touch or new aspect of biological governance embedded into our programs and commanded to evolve, we take a step closer to programs that will one day be able to find solutions, consistently, to such large and important problems like the Job Shop Problem.

A lot of research pertaining to optimization of the JSP in recent years has focused on the many meta-heuristic methods, see Dimopoulos, and C., Zalzala, A., [4]. Most of the research conducted in scheduling optimization uses the Genetic Algorithm, an algorithm that is modeled after evolution and the "survival of the fittest" law. According to Jain and Meeran [11], this evolutionary computation algorithm has had success with the JSP, it hasn't proven itself yet to be a superior method when compared to other computational intelligent techniques. One field of computational intelligence which has shown promise in solving other optimization problems is Particle Swarm Optimization (PSO). The PSO Algorithm was originally modeled after the behavior of a flock of birds in flight. It is currently considered one of the fastest converging techniques of computational intelligence, and a well suited algorithm for multi-modal functions, Song and Gu [19]. As of yet, particle swarm behavior has not been solely applied to the traditional Job Shop Problem. This could because of the fact that the PSO Algorithm doesn't lend itself inherently to the optimization of permutations, which are often use to indirectly represent solutions to the JSP. Although, the PSO optimization technique *has* been applied to other

scheduling problems, such as the Flowshop problem (FSP) and the Flexible Job Shop Problem (FJSP), see [12], [21], [22], [25], which are related problems and will be discussed later. One of the strengths of the PSO algorithm is its speed. It is widely hailed as very fast converging algorithm. Given the success of this optimization strategy in other areas, and the ever present need for better heuristic methods for solving Job Shop Problems, it is only fitting to try and make this connection of the PSO Algorithm to the JSP, especially due to the fact that constructing a feasible schedule can be very computationally expensive. Therefore, any algorithm that can prove itself faster will be advantageous. And of course, any new strategy should be studied at least briefly if it could open the door for more interesting research, which could lead to better optimization methods in the long run. Because of the previously mentioned obstacle regarding the continuous nature of the PSO and the permutation nature of an indirectly represented JSP, a unique approach has been taken to make this optimization possible. This approach involves giving each machine defined in a particular JSP its own independent particle swarm.

1.3 Outline of Work

This thesis is organized as follows: The Job Shop Problem is introduced in Chapter Two, and prior work is reviewed in Chapter Three. In Chapter Four, particle swarm optimization is explained. Chapter Five introduces the proposed algorithm, the JSP/PSO, and Chapter Six consists of the results from the proposed algorithm. Finally, in Chapter Seven conclusions are presented.

CHAPTER TWO

THE JOB SHOP PROBLEM

The Job Shop Problem (JSP) is one of many types of scheduling problems that researchers from many fields are currently attempting to solve optimally using various meta-heuristic algorithms. The solution to these scheduling problems is simply the determination of the optimal assignment of a finite number of resources to a finite number of operations, while adhering to many pre-defined constraints, usually precedent constraints. Precedent constraints, or technological constraints, dictate the order of operations for each job, or the order of machines a job must visit. A solution to a Job Shop Problem is a schedule specifying when each machine is to start processing certain operations that does not violate any precedent constraints. The ultimate goal is to minimize the makespan of the problem, or the minimum time required for all jobs to finish processing. In the following sections the JSP is defined mathematically, and then descriptions of the different types of schedules that can be constructed are explained. Also, for distinction and clarification some of the other more famous problems in scheduling theory are briefly explained.

2.1 The Job Shop Problem Defined

A $(n \times m)$ Job Shop Problem is defined by a specific number of jobs, *n*, each consisting of an order of operations, *m*, which are equal to the number of machines or resources specified in the problem. So a job, J_i is a predefined order of operations $\mathbf{O}_i = (O_{i,1}, O_{i,2}, \ldots, O_{i,m})$. Each operation O_{ij} has a processing time, or job duration, τ_{ij}. For the traditional JSP the following rules apply:

- Each job must be processed by each machine in a certain order (precedent constraints)
- Each machine can only process one job at a time
- Each job can only be processed by one machine at a time
- Each job must be processed by each machine exactly once
- No preemption is allowed, or once a job has started processing it can not be interrupted.

The specific order of operations, or the order of machines that a job must visit, are the precedent constraints, or the technological constraints for that job. These precedent constraints give the JSP its difficulty. A traditional Job Shop Problem is simply defined by specifying the technological or precedent constraints and the processing times for each operation. An example of a (10×10) JSP, the famous MT10 [14] problem, is shown in Table 2.1.

Table 2.1: Scheduling Problem Example

	Machine Sequence (Processing Time)
Job 1	0 (29) 1 (78) 2 (9) 3 (36) 4 (49) 5 (11) 6 (62) 7 (56) 8 (44) 9 (21)
Job 2	0 (43) 2 (90) 4 (75) 9 (11) 3 (69) 1 (28) 6 (46) 5 (46) 7 (72) 8 (30)
Job 3	1 (91) 0 (85) 3 (39) 2 (74) 8 (90) 5 (10) 7 (12) 6 (89) 9 (45) 4 (33)
Job 4	1 (81) 2 (95) 0 (71) 4 (99) 6 (9) 8 (52) 7 (85) 3 (98) 9 (22) 5 (43)
Job 5	2 (14) 0 (6) 1 (22) 5 (61) 3 (26) 4 (69) 8 (21) 7 (49) 9 (72) 6 (53)
Job 6	2 (84) 1 (2) 5 (52) 3 (95) 8 (48) 9 (72) 0 (47) 6 (65) 4 (6) 7 (25)
Job 7	1 (46) 0 (37) 3 (61) 2 (13) 6 (32) 5 (21) 9 (32) 8 (89) 7 (30) 4 (55)
Job 8	2 (31) 0 (86) 1 (46) 5 (74) 4 (32) 6 (88) 8 (19) 9 (48) 7 (36) 3 (79)
Job 9	0 (76) 1 (69) 3 (76) 5 (51) 2 (85) 9 (11) 6 (40) 7 (89) 4 (26) 8 (74)
Job 10	1 (85) 0 (13) 2 (61) 6 (7) 8 (64) 9 (76) 5 (47) 3 (52) 4 (90) 7 (45)

Here each row is a job sequence J_i with processing time τ_{ij} in parentheses. There are ten rows, which mean there are ten jobs, and there are ten columns which mean there are ten machines. Each row is a permutation of numbers representing the sequence of machines that job must visit. For example, Job 3 must go to Machine 1 first for 91 time units, then it must be processed on Machine 0 for 85 time units, then go to Machine 3 for 39 time units and so on…

2.2 Related Scheduling Problems

The traditional Job Shop Problem has many "cousins", or other scheduling problems with the same goal, to produce an optimal schedule of a number of jobs through a number of machines. The nature of the specified constraints, the number of resources available, and the number of jobs are what differentiate one "shop" scheduling problem from the other. In the following sections some of the other scheduling problems are briefly explained to distinguish the traditional JSP, to add perspective on the scope of scheduling problems in general, and to provide context to their references later in this paper.

2.2.1 The Flexible Job Shop Scheduling Problem Another scheduling problem that proves itself to be computationally hard is the Flexible Job Shop Scheduling problem (FJSP). In the FJSP an operation of a given job can be processed by multiple machines. So each operation O_{ij} has a pre-defined list of machines that are capable of performing that operation. A FJSP with "total flexibility" means that any operation can be processed by any machine. However, the processing times by each machine for a given operation are all different, which lends to the need for optimization. Most FJSP problems are optimized in two steps, first the optimal assignment of each operation to a machine is made, then the optimal schedule is determined. The FJSP has an even bigger computational space than the JSP, and in fact the JSP can be considered a general case of the FJSP.

2.2.2 The Flow Shop Scheduling Problem The Flow Shop Scheduling Problem (FSP) is another *n* job by *m* machine scheduling problem. The FSP differs from the JSP in that each job *j* has the same order of operations, or precedent constraints. The goal is to determine the optimal order of operations that all jobs will share that will minimize the makespan.

2.2.3 The Single Machine Weighted Tardiness Problem In the Single Machine Weighted Tardiness Problem (SMWTP) there is only one single machine and a list of operations to be processed on that machine. At first thought, this does not seem to pose an extraordinary scheduling problem, because with only one machine the makespan, or total completion time, of any permutation of jobs will be the same. However, the problem's

difficulty comes into play with the addition of due dates and weights for each job, parameters not defined for the other scheduling problems in a non-dynamic environment. The goal becomes to find the optimal order of operations not to minimize the makespan, but to minimize the total weighted tardiness. In the SMWTP the tardiness of each job is multiplied by its weight, and the summation of this weighted tardiness is the final cost value of the schedule. Suddenly, with the addition of weighted tardiness and due dates, this problem becomes a scheduling problem of great complexity like its other cousin scheduling problems, especially for problems with 50 or more operations each with different weights.

2.3 Types of Schedules

As stated before, a solution to a job shop problem is in essence a schedule of operations, or a set of starting times for each operation, which does not violate any of the imposed upon constraints. It turns out that schedules can be classified into different categories depending upon how the sequencing of operations affects the makespan of the problem. This will be explained in the following sections.

2.3.1 Semi-active Schedules Semi-active schedules are schedules in which the next operation in a technological sequence is scheduled at the earliest allowable time. In semi-active schedules no operation can be started earlier without changing the operating sequence of any machine. Changing the operating sequence of a machine will not necessarily violate precedent constraints. For example, if Jobs 1 & 2 are both waiting to be processed by Machine 4 then it doesn't necessarily violate any precedent constraints to

change the order of Jobs 1 & 2 on Machine 4. The fact that they will be scheduled at the earliest allowable time makes it a semi-active schedule. To understand this concept, it is easier to show visually what a semi-active schedule is NOT. Consider the following simple JSP problem that will be used throughout this thesis shown in Table 2.2:

Table 2.2: Simple JSP Example

Job	Machine Sequence (Processing Time)		
Job 1	1 (3)	2 (5)	3 (2)
Job 2	1 (5)	3 (1)	2 (4)
Job 3	2 (4)	1 (2)	3 (1)

Figure 2.1 is a Gantt chart of one possible non-semi-active schedule for this simple (3×3) JSP.

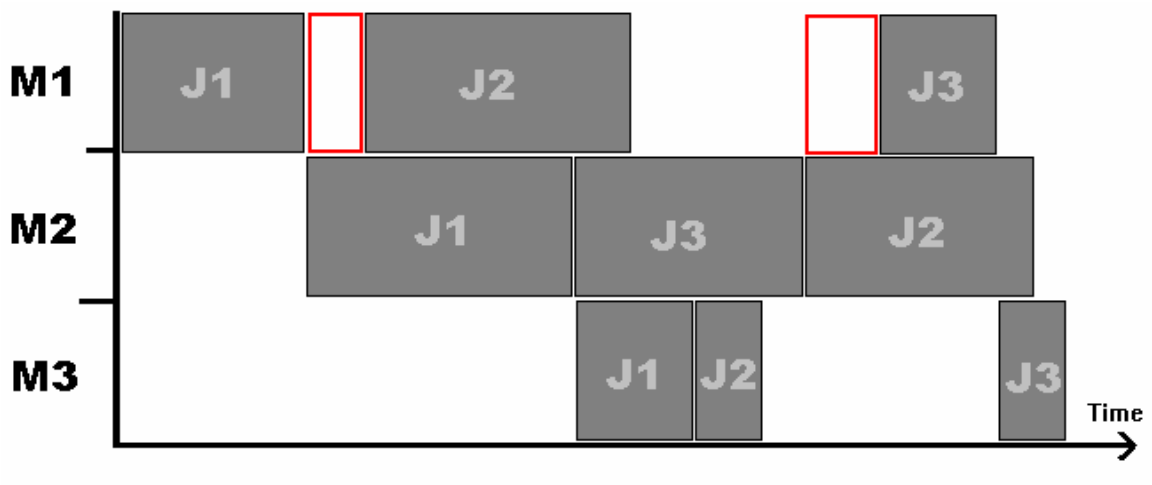


Figure 2.1: Non-semi-active Schedule Example

Notice how the second and third operations on Machine 1 are not scheduled at the earliest allowable time. The red boxes mark this unnecessary delay. Obviously, schedules that are not semi-active are not optimal. To make this schedule semi-active the second and third operations on Machine 1 can simply be moved left to the earliest allowable time.

Figure 2.2 shows this new semi-active schedule. Notice that no operation can be started earlier without altering the operating sequence of any machine.

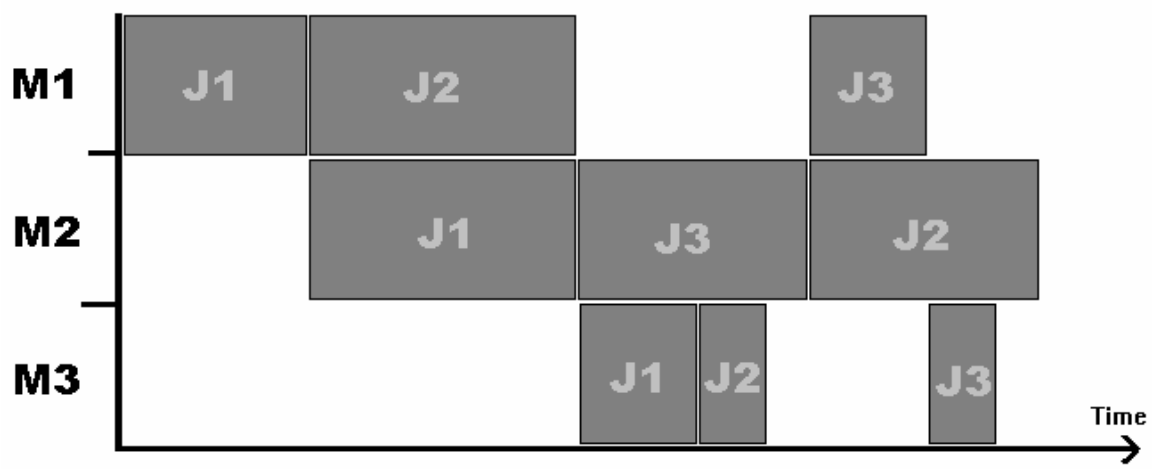


Figure 2.2: Semi-active Schedule Example

2.3.2 Active Schedules These are schedules in which no operation can be started earlier without violating a *precedent constraint*, or increasing the total processing time of any machine. The schedule above in Figure 2.2 is semi-active, which means no operation can be started earlier without altering the *operating sequence* of any machine. However, if we were to alter the operating sequence of any machine, an active schedule could be produced. Now, it might be apparent that the semi-active schedule above is not optimal. Semi-active schedules in general are not optimal. Optimal schedules lie in the space of active schedules. It is possible to alter the operating sequence of the machines to produce a schedule with a smaller makespan and preserve the precedent constraints of the problem. To turn a semi-active schedule into an active one, permissible left shifts are made. Permissible left shifting simply means switching the operating sequence of two adjacent operations as long as it doesn't violate any precedent constraint or cause a delay in any of the machine sequences. When all permissible left shifts are made to a semi-

active schedule an active schedule is thereby obtained. This method is useful in turning an already existing semi-active schedule into an active schedule. The following figures demonstrate this left shifting procedure.

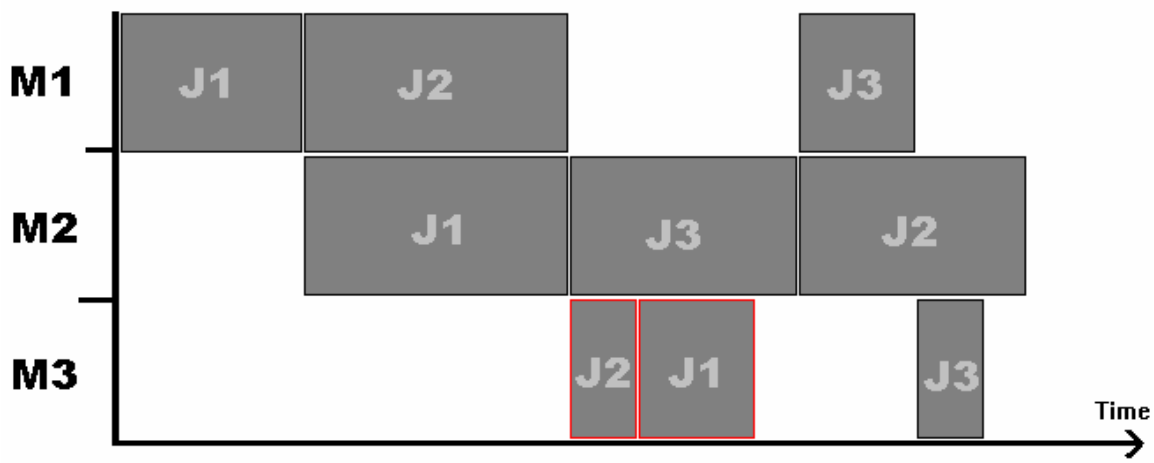


Figure 2.3: Left Shifting (a)

In Figure 2.3 the first two operations in Machine 3's operating sequence are switched, or left shifted. This move does not violate any of the precedent constraints of the problem and does not delay any machines sequence. This left shift may seem pointless, because the overall makespan of the problem did not decrease. However, it allows us to do the next left shift shown below in Figure 2.4.

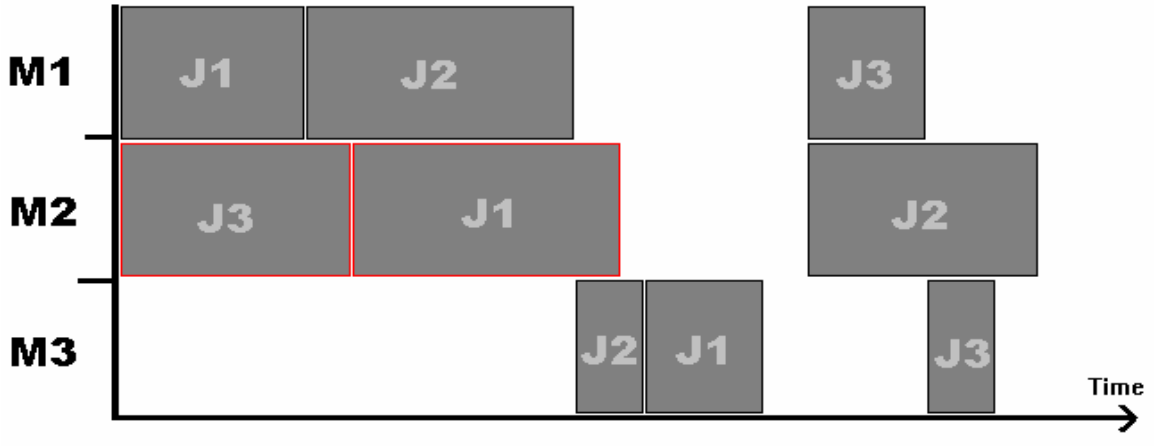


Figure 2.4: Left Shifting (b)

This time the first two operations in Machine 2's sequence were shifted and scheduled at the earliest allowable time. Figure 2.5 below shows the final *active* schedule.

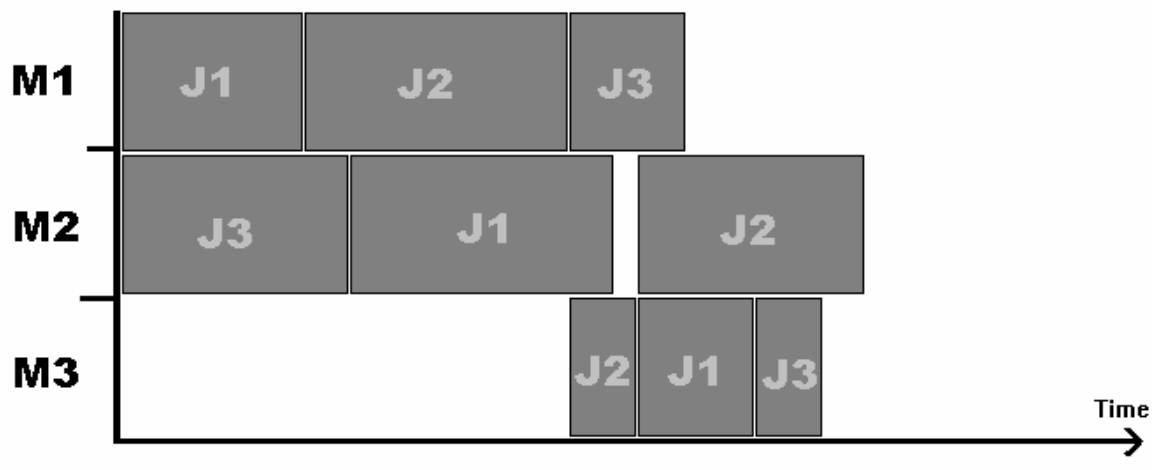


Figure 2.5: Active Schedule after Left Shifting

Note the drastic improvement of the makespan obtained by permissible left shifting. This is an active schedule because there are no more permissible left shifts without violating a precedent constraint. For a simple (3×3) JSP one might mistakenly think that there is only one active schedule and it is optimal. However, this is not the case. Many more active schedules could be generated for this problem, but obviously not all of them can be optimal. The schedule above may or may not be optimal. Keep in mind the enormous combinatorial space of the JSP as mentioned in the Introduction. For this (3×3) problem, there are $(3!)^3 = 196$ possible schedules. Granted, many of these schedules are infeasible, but depending upon how an optimization algorithm works those infeasible schedules are still in the search space, which adds complexity.

2.3.3 Non-Delay Schedules These are schedules in which no machine is kept idle while it could be processing an operation. The active schedule in Figure 2.5 is also happens to be a non-delay schedule as well. The instant an operation becomes available for a machine then it is scheduled without unnecessary delay. It might be tempting to think of optimal schedules as non-delay schedules, however this is not always true. Often holding a machine idle for a period of time so that more jobs become available for processing (meaning they get done processing on other machines), and then scheduling one of these newly available jobs can lend to optimal schedules. This is best illustrated with an example. In the following figure, Figure 2.6, another possible Gantt chart is shown of the JSP in Table 2.2 (the JSP of the previous example) with a single change. The processing time of the first operation in Job Two has been increased to make this point. Clearly, this an active *non-delay* schedule, since no machine is kept idle when it could be processing a job.

Figure 2.6: Non-delay schedule

This schedule is a non-delay schedule, however it is not optimal. Notice when Job 1 is done processing on Machine 2 it is immediately scheduled on Machine 3 (because it is

the first job to become available for that machine). Also notice how Job 2 on Machine 1 finishes not too long after the previously mentioned event. Now, notice how the next operation for Job Two also happens to be for Machine 3. The point is both of these operations require Machine 3 within a relatively small amount of time. With non-delay schedule building, Job 1 is immediately processed first. It turns out it might be better to delay processing on Machine 3 and process Job 2 instead of Job 1 first. This idea is shown with the schedule below in Figure 2.7.

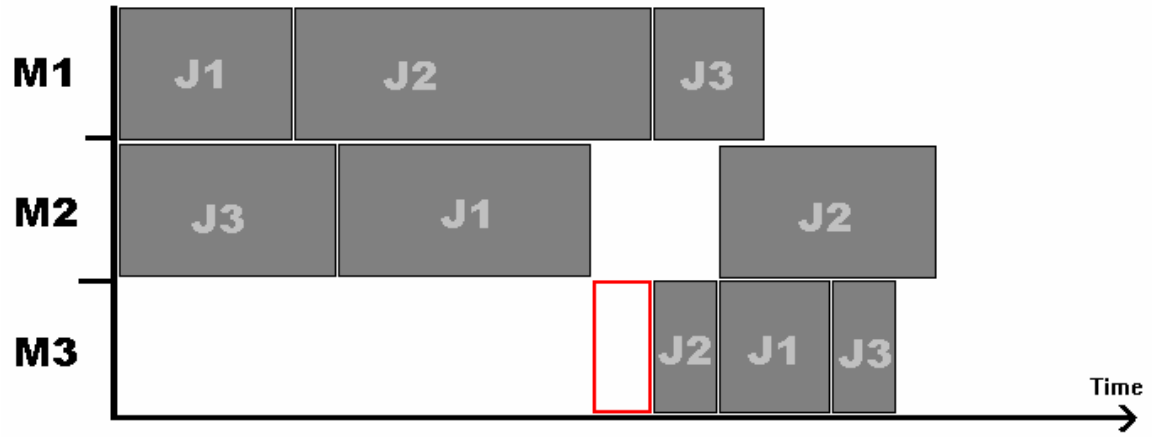


Figure 2.7: Active Schedule Example

The red box indicates the idle delay time for Machine 3. This delay and the resulting processing of Job 2 before Job 1 on Machine 3 produced a more optimal solution. This is why we can safely limit our search to the set of active schedules, but not necessarily the set of non-delay schedules. This schedule is both active and semi-active, and should not be confused with a non-semi-active schedule. A schedule that is not semi-active holds machines idle for no benefit. It is important to understand the relationship of these different types of schedules. The figure below should help the reader of this thesis.

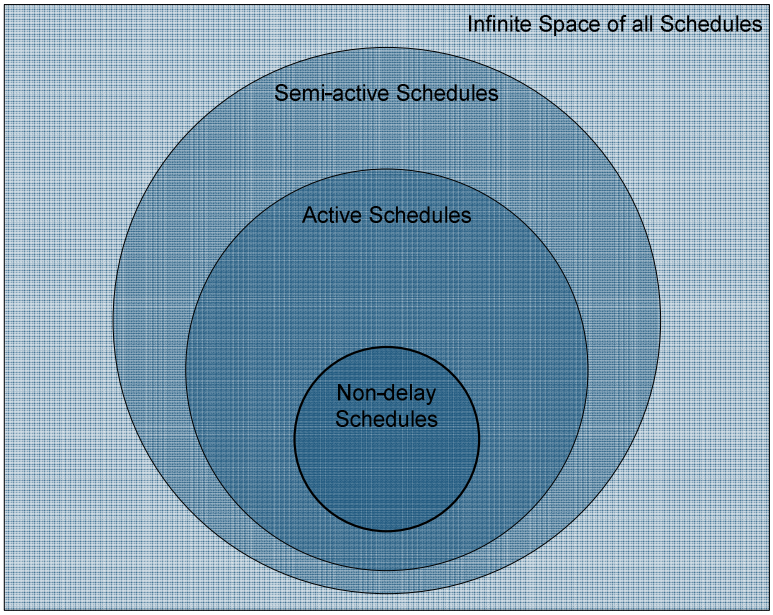


Figure 2.8: Relationship of Schedules (a)

By looking at Figure 2.8 we can see that active schedules are a subset of semi-active schedules and non-delay schedules are a subset of both active schedules and semi-active schedules. We know that optimal schedules are active schedules, but *not necessarily non-delay* schedules, therefore it is important to not limit your search to strictly non-delay schedules. However, it should be obvious that optimal schedules will most likely be schedules where the amount of delay time for any given machine is kept to a minimum. In simple terms, most of the time it is desirable for operations on machines to start right after another one finishes, however every once in a while it is desirable to schedule an operation for a machine that will not be available immediately for processing when the machine first becomes available. Therefore, it might be beneficial to limit searching to a set of what are called *Parameterized Active Schedules*.

2.3.4 Parameterized Active Schedules Parameterized Active Schedules are non-delay schedules where the delay is *no more than* a specified parameter. By specifying a maximum amount of delay time for machines, researchers can limit the search space to a subset of active schedules and a superset of non-delay schedules. The larger the parameterized delay, the more active schedules are included in this set. Parameterized active schedules are explained by Goncalves, Mendes and Resende [8]. Figure 2.9, which is an adaptation from a figure in [8], illustrates this relationship below.

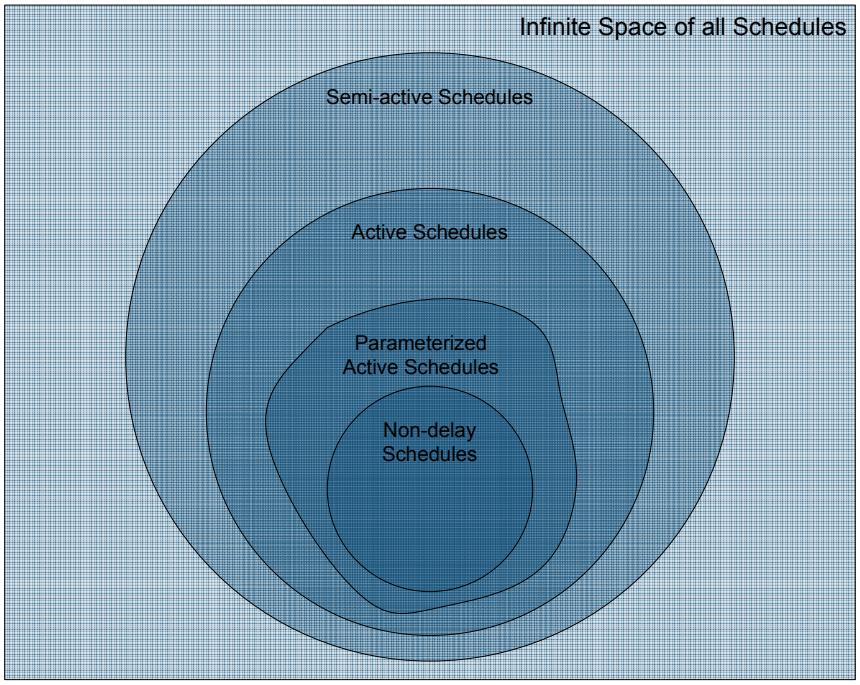


Figure 2.9: Relationship of Schedules (b)

Limiting a search to a set of parameterized active schedules is helpful in that the search space is decreased, but risky because an optimal schedule could lie outside a set defined by a parameter of delay/idle time. Obviously, when the parameter is zero, non-delay schedules are produced, and if the parameter is very large semi-active and non-semi-active schedules are produced. Parameterized active schedules played an important part

in this research, which will be explained in detail in Chapters 5 and 6. Now that the different types of schedules have been discussed, how schedules are represented and built can be explained.

2.4 Representation of Schedules

In the examples of previous sections, Gantt Charts have been used to show various solutions to the Job Shop Problem. Gantt Charts allow a visual interpretation of a schedule, which is helpful when analyzing the makespan of a Job Shop Problem, or the classification of a schedule. Usually a Gantt chart is built from a *representation* of the schedule in the form of numbers or in the form of permutations. This representation can be of a direct fashion or indirect fashion, both have its advantages and disadvantages.

2.4.1 Direct Representation Direct representation of a schedule is precisely that. Direct representation of a schedule is any form of representation that directly specifies when all the operations are to begin processing on the machines. Optimization occurs directly in the schedule space, perhaps by left-shifting. Another example would be creating $(n \times m)$ set of starting times for a $(n \times m)$ Job Shop Problem and optimizing those values directly, based on the makespan they produce. This type of relationship could be well suited for meta-heuristic optimization since a meta-heuristic algorithm to learn or evolve a set of starting times, simply based on an objective value, like the makespan. It is obvious to see when using direct representation, infeasible schedules can be produced, because the precedent constraints of the problem are not considered. Consider the same Job Shop Problem of used in the previous examples shown in Table 2.3.

Table 2.3: Small JSP

Job 1	1 (3)	2 (5)	3 (2)
Job 2	1 (5)	3 (1)	2 (4)
Job 3	2 (4)	1 (2)	3 (1)

If direct representation was used in the form of starting times for each operation, the solution might look like this after optimization, if t_{ij} is the starting time for operation O_{ij}.

$$t_{ij} = \begin{bmatrix} 8.0 & 0.0 & 5.0 \\ 3.0 & 1.0 & 9.0 \\ 5.0 & 1.0 & 0.0 \end{bmatrix}$$

Which will produce the following schedule of Figure 2.10:

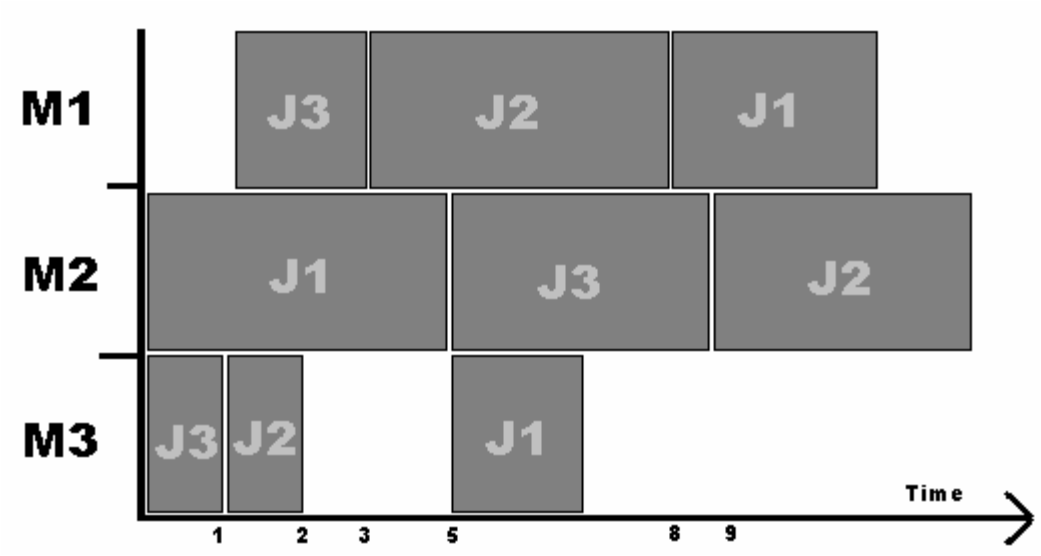


Figure 2.10: Infeasible Schedule

This is a very good schedule in terms of makespan, but it is infeasible, because it violates several precedent constraints. For example, Job 1's precedent constraints dictate it must go to Machine 1, then Machine 2, then Machine 3, but this is not true for the above example. These precedent constraints add difficulty to the problem, especially when

using a direct representation scheme, because additional measure must be taken in order to produce feasible schedules. Since many meta-heuristic optimization algorithms stochastically operate in the decision space of the problem many of their pure solutions will violate precedent constraints. Therefore, if direct representation is implemented, extra steps must be added for optimization to take place. Either infeasible schedules can simply be removed from the population, or repair procedures can be performed to transform infeasible schedules to feasible schedules (perhaps something like left-shifting). Extra steps can also be added during the schedule building process itself to ensure feasible schedules, however, technically the direct representation would then become *indirect* representation, and the extra steps would become a scheduling algorithm. The scheduling algorithm would then become necessary to interpret the schedule. Direct representation is advantageous because of its simplicity and speed, but not good because of its indiscriminate search of both feasible and infeasible schedules. Indirect representation has the opposite characteristics.

2.4.2 Indirect Representation and Scheduling Algorithms One common way to avoid violating precedent constraints is to represent a JSP schedule in an *indirect* fashion and use a scheduling algorithm to transform the indirect representation into a feasible schedule. The scheduling algorithm consults the precedent constraints of the problem and uses the indirect representation of the problem to make decisions that ensure the generation of a feasible schedule. This is advantageous because the optimization operators can operate without venturing into infeasible space. These scheduling algorithms can range from very simple to very complex, and can produce schedules

anywhere from semi-active to non-delay. The more restrictive the search of the schedule space, the more complicated the scheduling algorithm becomes to produce those schedules. The following two examples of a scheduling algorithm illustrate these facts.

2.4.2.1 Permutation with Repetition An example of an indirect representation is Permutation with Repetition originally proposed by Bierworth [1]. In this representation a permutation of job numbers $(n \times m)$ long is decoded into a feasible schedule by a scheduling algorithm. Each number represents a job, and that number is repeated *m* times. The kth repetition of job *j* represents the kth operation to be processed for job *j*. This scheduling algorithm builds semi-active schedules by default, which include the class of active and non-delay schedules. An example of the permutation with repetition and the scheduler is shown in Figure 2.11. The corresponding precedent constraints of the JSP are also shown in Table 2.4.

Table 2.4: Simple Job Shop Problem

Job 1	1 (3)	2 (5)	3 (2)
Job 2	1 (5)	3 (1)	2 (4)
Job 3	2 (4)	1 (2)	3 (1)

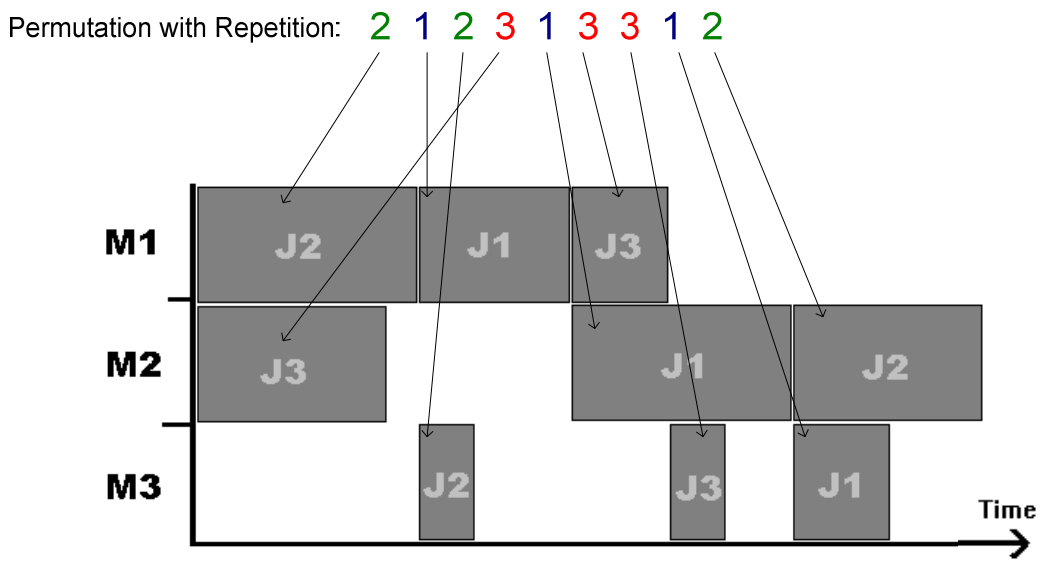


Figure 2.11: Permutation with Repetition

It is easy to understand by using an indirect representation with a corresponding scheduling algorithm it is possible to ignore precedent constraints when searching in the decision space (the $(n \times m)$ permutation of numbers for the example above). This is because a scheduling algorithm will consult the precedent constraints before scheduling. Permutation with Repetition is a simple and effective scheduling algorithm.

2.4.2.2 The GT Scheduling Algorithm As mentioned before, the type of schedule produced as a result of a scheduling algorithm very much depends upon the complexity of the scheduling algorithm. A scheduling algorithm that simply schedules operations as early as possible is liable to produce semi-active schedules, which include non-optimal schedules. However, an algorithm that "looked ahead into the future" might produce active schedules. Since all optimal schedules are active schedules it would be beneficial to limit our search to the set of active schedules. The GT Algorithm does just this. It is

the most famous scheduling algorithm by far, and was developed by Giffler and Thompson [12] in 1960 and has been used ever since. The algorithm is presented below:

1. Let D be a set of all earliest schedulable operations in all job sequences not yet scheduled.

2. Let operation, O_{jr}, be the operation with the earliest completion time, EC, in set D. $O_{jr} = \min \{O - D \mid EC(O) \}$. Where j is the job and r is the machine.

3. Develop a conflict set for machine M_r consisting of all operations that will require machine r before operation O_{jr} will be completed. ES = Earliest Starting Time. $ES(O_{kr}) < EC(O_{jr})$.

4. Select an operation out of the conflict set, and schedule it on M_r.

Basically the algorithm builds a schedule one operation at a time. First it selects a machine by determining which unscheduled operation has the earliest completion time, EC, and what machine it requires to be processed on, *r*. It then looks ahead for operations that can start to process on *r* before time EC, and places these operations in a conflict set. It is called a conflict set because they all have an equal "right" to be processed on machine *r* during the considered time EC. By default, the GT Algorithm searches active schedule space. However, one of the great aspects of this algorithm is that if the algorithm can be used to build only non-delay schedules by making EC equal to zero. This means conflict sets will only be made from operations that already are ready for machine *r*, not any that will become available. This should agree with what has been explained about non-delay schedules previously. In fact, we can search the space of parameterized active schedules, by making EC a value between 0 and EC. This idea is explored in more detail in Chapter 6.

Once the conflict set has been generated which may consist of one or more operations, an operation must be selected and scheduled. The GT Algorithm does not specify how to do this, an active schedule will be built regardless. However, an *optimal* schedule will require the right selections out of the conflict sets generated throughout the schedule building process. This is where priority lists and dispatching algorithms come into play with the GT Algorithm. These are explained in more detail in the following chapters. It turns out that the proposed JSP/PSO Algorithm uses the GT Algorithm in conjunction with priority list, which is explained in Chapters 5 and 6.

CHAPTER THREE

LITERATURE REVIEW

The job shop problem comes from the field of deterministic scheduling theory. Scheduling theory is not a new area of scientific research. Rather, it has been around for over 50 years. Since the advent of neural networks, evolutionary algorithm and other meta-heuristic techniques the problem has lured researchers from many different fields in a quest to find a better optimization technique. In this section some of the early defining works in scheduling theory are mentioned, and then a literature review is performed on the JSP and its history with some of the approximation methods used to solve it, particularly the meta-heuristic technique, Particle Swarm Optimization.

3.1 The Job Shop Problem History and Early Methods of Optimization

One of the most famous and defining works in scheduling theory was published in the early 1960's. Giffler and Thompson's "Algorithms for Solving Production Scheduling Problems" [7] introduced the most famous and widely used scheduling algorithm, called the GT algorithm. The GT algorithm insures the construction of an active schedule. Active schedules are explained in Chapter 2. Three years later "Industrial Scheduling" [14] by Muth and Thompson was published which introduced the first famous job shop scheduling benchmark problem, a (10×10) problem that took 20 years to solve exactly,

because of the high combinational complexity, which are the nature of such problems. Since then researchers have applied many deterministic algorithms to the JSP, and more recently the use of heuristics and evolutionary techniques. In "A State-of-the-art Review of Job-Shop" [11] Jain and Meeran divide the approaches to the solving this huge combinatorial problem into two main techniques, efficient and exact. A visual representation of these various methods and their relation is illustrated in Figure 3.1 below.

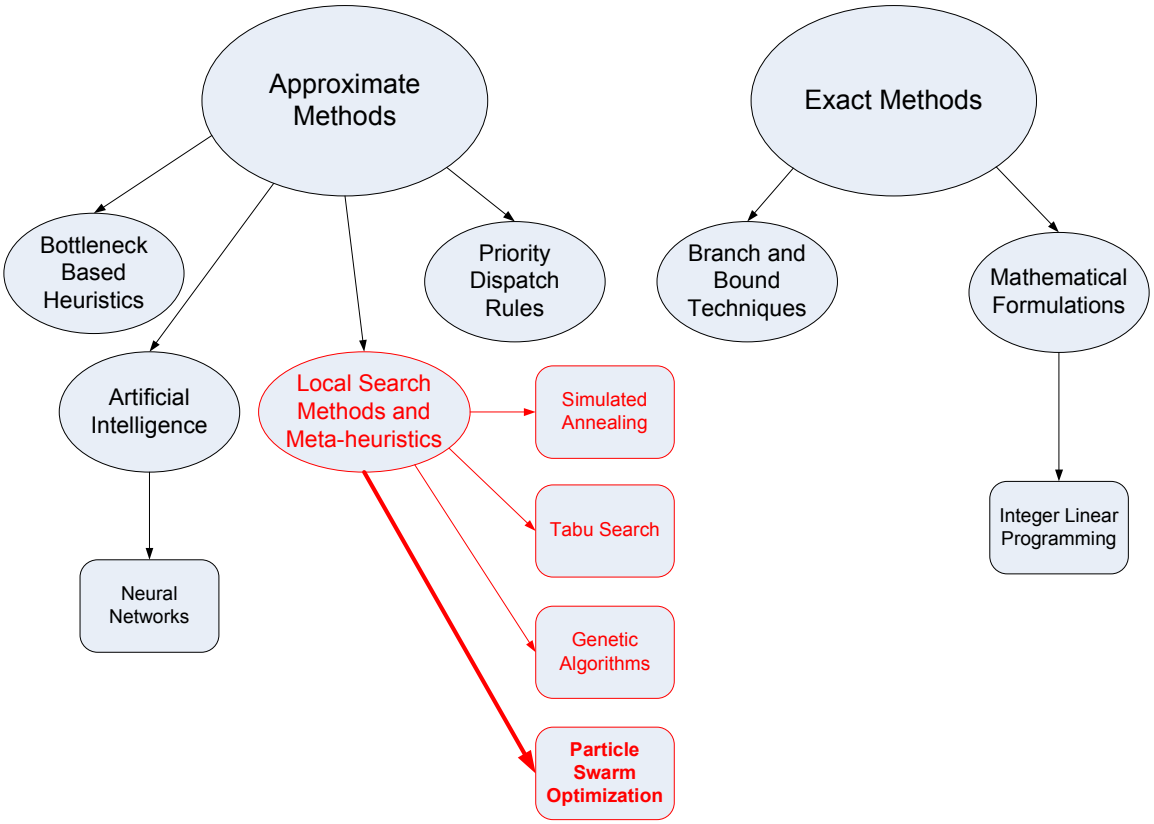


Figure 3.1: JSP Methods

Exact methods, obviously, use mathematical formulations to arrive exactly at the optimal solution. Since the problem space of the JSP problem is so large, these exact methods are not useful for anything but problems of small dimension, probably (10×10) or less.

This is why the MT10 problem, a (10×10) job shop problem introduced in 1963 took 20 years to solve exactly. The proposed research has focused on the application of a meta-heuristic technique called Particle Swarm Optimization (PSO). As will be shown, not much research has been conducted in the area of the JSP and the PSO, although the JSP does have a long history exploiting other meta-heuristic techniques shown in Figure 3.1, especially the Genetic Algorithm.

3.2 Approximate Methods

Approximate methods, called approximate because they do not use mathematical formulations, calculation of gradient for example, to arrive at an exact optimal solution. Approximate methods have the ability to explore the solution space quickly and arrive at a near optimal solution in a directed stochastic manner. Approximate methods hold the key to generating the best schedules of problems of order (10×10) or higher. Because of the exponentially growing search space of the JSP, to exhaustively and exactly determine the optimal solutions for JSPs with order higher than (10×10) can be too time consuming, even with modern computers. This was especially true back in the 50's and 60's when today's computing power was not available. To compensate engineers, scientists and production managers relied on heuristics to help them produce a near optimal schedule. These heuristics normally took the form of Dispatch Rules, and are explained in the following section.

3.2.1 Dispatching Rules Dispatching rules are another way to indirectly determine the schedule of a particular JSP. Dispatching rules are commonly used in conjunction with

the GT Algorithm which was developed around the same time period as the GT Algorithm. Recall that the GT algorithm must select operations from a conflict set, step 4 from the GT Algorithm (explained in Chapter 2). This is where dispatching rules come into play. Dispatching rules are heuristic guide lines which dictate which jobs should be selected out of a conflict set according to some measurable standard, such as "shortest processing time". Developing good dispatching rules is where a lot of heuristics come into play in the Job Shop Problem. Obviously, one can surmise that there might exist intuitive ways of assigning priority to operations so as to generate a schedule with near optimal makespan. Before the processing power of modern computers, a lot of focus was given to the development of Dispatching Rules. In fact there are hundreds of dispatching rules that have been tested through the years, starting in the 1950'and 60's when Jackson [9],[10], Smith [18], Rowe and Jackson [17] constructed the earliest work on dispatch rules. Recently, in 1996 Chang [2] composed a survey on many dispatching rules and compared them to each other. The most popular ones were summarized in a table by Jain and Meeran [11]. This table, Table 3.1, is shown below to give an idea of the kind of heuristics that are involved in this area of scheduling theory.

Table 3.1: Dispatching Rules

Name	Rule
SPt/Twkr	Smallest ratio of the processing time to total work time
Lrm	Longest remaining work excluding the operation under consideration
Mwkr	Most work remaining to be done
Mopr	Most number of operations remaining
Spt/Twk	Smallest ratio of processing time to total work
Fcfs	The operation that arrived the earliest is processed first
Lso	Longest subsequent operation
Spt	Shortest processing time
Fhalf	More than one half of the total number of operations remaining
Ning	Next operation is on the machine with the fewest number of operations waiting

3.2.2 Meta Heuristic Methods Meta heuristic techniques can be classified as Local Search techniques because of the way they search and solve problems. Meta-heuristic techniques all start at a certain initial solution, or group of solutions, and will iteratively arrive at a near optimal solution by making small or local changes in the problem space. Most research of meta-heuristic techniques and scheduling problems have focused on using the Genetic Algorithm, although Simulated Annealing, Tabu Search, Particle Swarm Optimization and Ant Colony Optimization have been applied as well.

3.2.2.1 Priority Lists Like dispatching rules, priority lists are another heuristic way of selecting an operation out of a conflict set. Each machine is assigned a processing priority of the jobs in the JSP. Operations are selected out of conflict sets by referencing the machines processing priority for the jobs in the conflict set. Each machine's priority list can then be optimized by determining the best permutation of job numbers or best permutation of the priority list. Therefore, this indirect representation method lends itself much more conducive to meta-heuristic optimization than using

dispatching rules. In 1985 Davis [3] was the first to apply an evolutionary algorithm in an effort to solve the Job Shop Problem. He used a genetic algorithm to evolve a priority list for each machine. An example of a priority list for a (5×5) problem is given in Table 3.2.

Table 3.2: Priority List Representation

	First Priority	Second Priority	Third Priority	Fourth Priority	Fifth Priority
Machine 1:	Job 2	Job 4	Job 1	Job 5	Job 3
Machine 2:	Job 1	Job 3	Job 5	Job 2	Job 4
Machine 3:	Job 4	Job 2	Job 5	Job 1	Job 3
Machine 4:	Job 3	Job 1	Job 2	Job 4	Job 5
Machine 5:	Job 5	Job 3	Job 4	Job 2	Job 1

This use of priority lists, by Davis, is what first opened the door to the application of meta-heuristic methods of solving the JSP. By evolving a priority list, Davis was able to ignore the precedent constraints of the problem, which makes using a genetic algorithm much easier. It should be noted that even though varying a machines priority list will indirectly affect the schedule of a specific JSP, it will not always produce a unique schedule. More than one priority list can easily generate the same schedule. This can make the optimization process more difficult.

Since Davis's work in [3], a large amount of research has been conducted into the application of the many meta-heuristic techniques to the JSP. Most of the research has been focused on the Genetic Algorithm (GA), although some consider the GA an ineffective way to solve the JSP. When compared to Simulated Annealing (SA), Tabu Search (TS), the GA seems to perform the poorest according to a comparative study done by Pirlot [16].

3.2.2.2 Particle Swarm Optimization As of yet, Particle Swarm Optimization has not been applied to the traditional Job Shop Problem, with one exception. In 2004 Weijun, et al. [24] applied a hybrid Simulated Annealing/PSO to the traditional JSP. In their study, Simulated Annealing (SA) was used to fine tune solutions found by the PSO algorithm. The results of this hybrid algorithm were very promising. Many of the most widely used Job Shop test bench problems were solved to optimal or best known solutions. This means that the implementation of a pure PSO algorithm might very well be an efficient and effective way of solving these types of huge combinatorial problems without using Simulated Annealing for fine tuning.

However, there is slightly more literature published on the application of the PSO algorithm to some of the other scheduling problems, such as the Permutation Flowshop Problem (PFSP) also called the Flow-shop Scheduling Problem (FSSP), or the Single Machine Weighted Tardiness Problem (SMTWT). A brief description of these problems can be found in Chapter 2. Tasgetiren, et al. [22] in 2004 applied PSO to the Single Machine Weighted Tardiness problem. They developed the Smallest Value Position Rule (SVP) in order to transform the continuous space of the PSO to the permutation space used to represent a solution of the SMTWT problem. Tasgetiren also published a paper on the PSO applied to the Permutation Flowshop Sequencing Problem, along with Liang, Sevkli, and Gencyilmaz [21]. The same SPV rule was used for the necessary space transformation from continuous to permutation space. This space transformation concept, which is explained in detail in Chapter 5, was used for the Traveling Salesman Problem by Pang, et al. [15] in 2004. Their method of space transformation was called

the GVP rule, or Greatest Value Priority. Using PSO and local search techniques they were able to solve medium scale (50 – 75 city) TSP Problems.

Particle Swarm Optimization has also been applied to the Flexible Job Shop Problem (FJSP) by Xia and Wu [25] recently in 2005. The FJSP is another scheduling problem related to the Job Shop Problem and is described briefly in Chapter 2 of this thesis. They used the PSO not to determine the schedule, but to assign each operation to a machine as is required for the FJSP. Instead of using a Greatest Value Priority Rule (GVP) or Smallest Value Position Rule (SVP), Xia and Wu simply limited the search space of the PSO to the value of the highest number in the permutation and simply rounded off any decimal values created from the PSO equations (see Chapter Four) to get an integer value. This integer value then represented a particular machine in the problem at hand. So the PSO in not necessarily new to the arena of scheduling problems, just the traditional Job Shop Problem.

Not all researchers have used this particular space transformation technique to apply the PSO in permutation problems. In 2003 Pang, et al. [23] developed a way to represent the difference in two permutations as a function of Swap Operators (SO). These Swap Operators perform a switch on two numbers in a permutation, and multiple swap operators in a given order form a Swap Sequence (SS). A Swap Sequence can, therefore, actually represent the difference between permutations. The authors then defined mathematical operations for the SS so that the traditional PSO velocity and position equations could be applied to an actual permutation. The velocity and position equations

for PSO are discussed in Chapter Four of this thesis. The authors' methods were applied to a simple 14 city TSP and were able to achieve the optimal solution. This achievement demonstrated the success of their method, however harder and larger TSP problems were not tested. Lian, Gu and Jiao [12] in 2005 developed a Similar Particle Swarm Optimization Agorithm (SPSOA) and applied it to the Flow-shop Problem (FSSP or PFSP). The Flow-shop problem is briefly described in Chapter 2. These researchers, like Pang, et al. [23] developed a way for the PSO algorithms to operate directly in the space of permutation problems. Their PSO algorithm is called "similar" because they used crossover and mutation techniques, easily performed on permutations, but originally developed for the Genetic Algorithms (GA). Their crossover and mutation operations were used to update the velocity and position of the particles in the PSO-like algorithm.

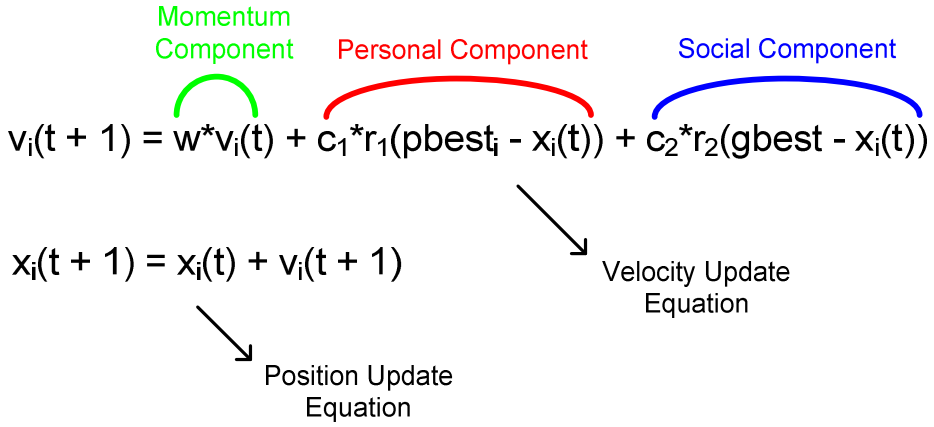
Many of the meta-heuristic techniques used to solve sequencing problems like the TSP, FSSP, SMWTP, FJSP and others, can also be adapted to solve the JSP since both solutions can be represented by a permutations of numbers. However, the JSP and FJSP are unique because their solution must be a permutation for every machine in the problem. In other words, there must be a set of permutations—one for each machine in the problem. Therefore, some extra steps might have to be implemented in the encoding and decoding process of the permutations. This issue is explained in more detail in Chapter 5. In the following chapter the general PSO Algorithm is discussed.

CHAPTER FOUR

PARTICLE SWARM OPTMIZATION

Particle Swam Optimization falls under the category of Swarm Intelligence. Swarm Intelligence is an optimization strategy that draws upon the knowledge of many agents interacting locally in an environment to find a global solution to a problem. There is no global control of these agents and they interact randomly and share information about their environment with one another. In the physical world, flocks of birds, schools of fish and ant colonies are examples of this behavior. In 1995 [5] Kennedy, a social psychologist, and Eberhart, an electrical engineer, first applied swarm intelligence behavior to the search space of optimization problems. They were inspired by a computer simulation of a flock of birds developed in 1990 by Heppner and Grenander. Their concept was to give each particle a social component and an individual component. Individual particles' behaviors would be influenced by their best positions (in the search space) found and by the best positions found by all particles in the swarm. Their hope was to design a search method that was able to find multiple optima not just the global. This way the particles can explore the search space and eventually converge to the global optimum. The agents or particles in this algorithm search the problem space by "moving through it" with a certain velocity. Each position a particle has in this space represents a

possible solution to the problem at hand. The particles in a traditional PSO algorithm are governed by the following equations for a single dimension in Figure 4.1:



- i = 1, 2…p , p = number of particles in swarm
- pbest is the best position found by that particle so far
- gbest is the best position found by any particle so far
- v = velocity of particle in a single dimension
- x = position of particle in a single dimension
- t = iteration number
- w is the inertial constant
- c_1, c_2 are acceleration constants
- r_1, r_2 are random numbers evenly distributed between [0,1]

Figure 4.1: Particle Swarm Optimization Equations

After the particles are first instantiated in the search space they move around with a certain direction and speed. As a particle "flies" through the search space the objective value is evaluated from each position. The particle has a personal or cognitive component and a social component that allow it to find better solutions to the objective problem. The previous personal best position is remembered by each particle, as well as the best position found by the whole swarm, which give it the personal and social components. The personal and global bests are determined by the quality of the

solutions those particles provided at those positions in the search space. It is evident by the equations above that both the personal best of a particle and the global best of all particles in the swarm influence the velocity of an individual particle. The farther a particle is away from their personal best or the global best, the larger the corresponding components of the velocity equations will be. Of course, these component values are subject to the random numbers r_1, r_2 and C_1, C_2. The inertial component labeled in green controls the impact of the particles previous velocity. The right selection of *w* will insure a particle that's velocity does not blow up over time. In fact, it is usually desirable to need a particle's velocity to slow down over time, and thereby become more precise and converge to the best found solution. It is also common to vary *w* with the iterations. The two acceleration constants, $c_1$and c_2, can also be varied with the iterations. *The relationships between these two constants at any given iteration will put either an emphasis on personal exploration, global, or both*. The random numbers r_1 and r_2 add the necessary stochastic quality that will ensure decent exploration of the search space. Since many optimization problems have equality or inequality constraints it is usually necessary to "block in" the particles within a certain region of the search space, by ensuring their magnitude in a given dimension never exceeds a specified value. The same cap, or minimum and maximum value restrictions, that might be placed on the position of a particle may also be placed on the velocity. The pseudo code for PSO is given below in Figure 4.2.

```
Begin
        -  Generate initial random positions and velocities for swarm
                - Initialize User Parameters: w, c1,c2, maxIterations
                - Store particles positions
                - Determine global best position and store

                while(terminate != true or iterations != maxIterations),
                                for all particles,
                                                Evaluate Fitness
                                                if fitness of current particle is <
personal best,
                                                                personalBest =
current particle's position
                                                end

                                                if fitness of current particle is <
global best,
                                                                globalBest =
current particle's position
                                                end
                                end

                                for all particles,
                                                update velocity
```

Figure 4.2: Particle Swarm Optimization Pseudo Code

The equations in Figure 4.1 are velocity and position update equations for a single dimension. It might be obvious that for a multiple dimension problem, a particle will have a velocity component in each dimension, because each particle must represent a solution to the problem. So for a 3-dimensional problem, the equations in Figure 4.1 will be used three times for a single particle, each time calculating the velocity in a single dimension. The PSO concept is best illustrated in a simple 3-dimensional problem that can be visually represented. Suppose the optimization problem that needed to be minimized is the simple function shown below:

$$f(x) = x_1^2 + x_2^2 + x_3^2 \qquad \text{where } x_1, x_2, x_3 \geq 0$$

Now, the solution is obviously $\mathbf{f} = [0.0\ 0.0\ 0.0]^T$, or the minimum will occur when x_1, x_2, and x_3 all equal zero. The problem space can be represented in a 3-dimensional cube. In order to optimize this problem using PSO, we must randomly generate particles in the search space. The size of our swarm will consist of only 3 particles in an effort to make the graphics that follow easier to understand. Normally, the swarm size is much larger, anywhere from the tens to the hundreds. The initial positions (representing three initial solutions) are shown in Figure 4.3 below. The constants C_1 and C_2 for this example will both be set to 2.

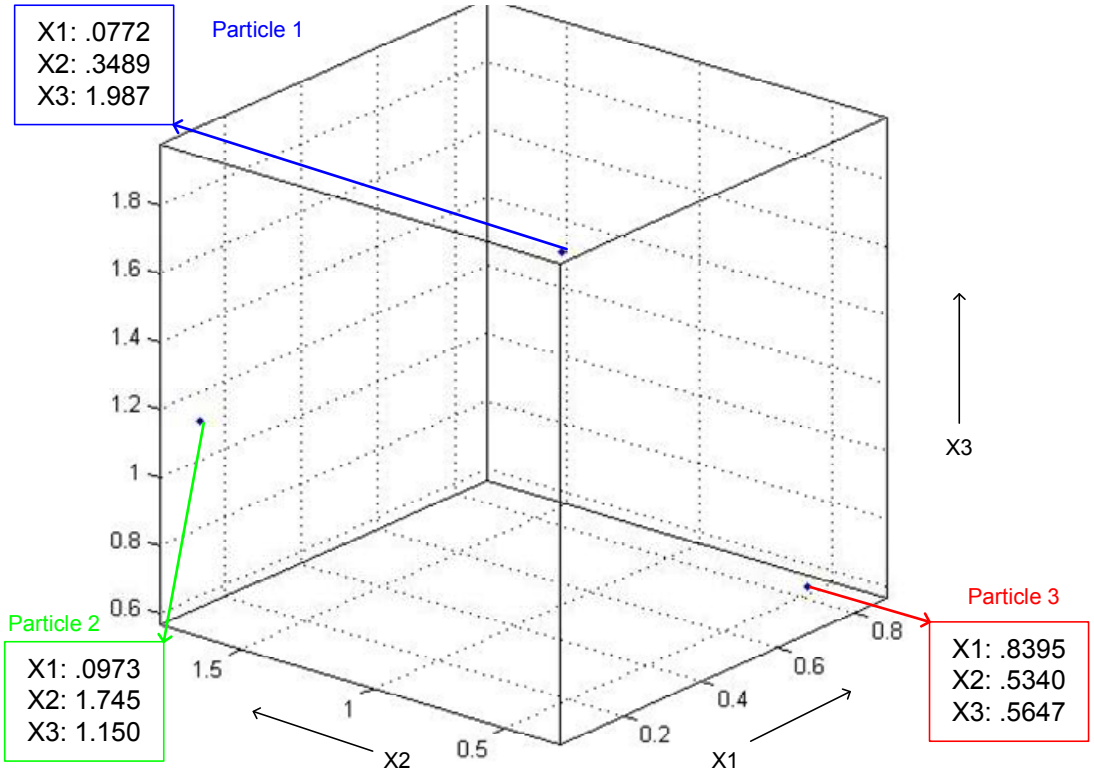


Figure 4.3: Particle's Initial Positions

When those particles' values are plugged back into our objective function, we get the following:

Particle 1: 4.0759
Particle 2: 4.3770
Particle 3: 1.3088

Neither of these solutions is very good, however the best solution is the solution represent by Particle 3. Knowing how PSO works, we would expect the other two particles to move in the direction of Particle 3, at least at first. This can be seen in the figure below. After 100 iterations, the particles of traveled very close to the optimal solution. Their paths are shown in the next two figures, Figure 4.4 and 4.5 from different distances.

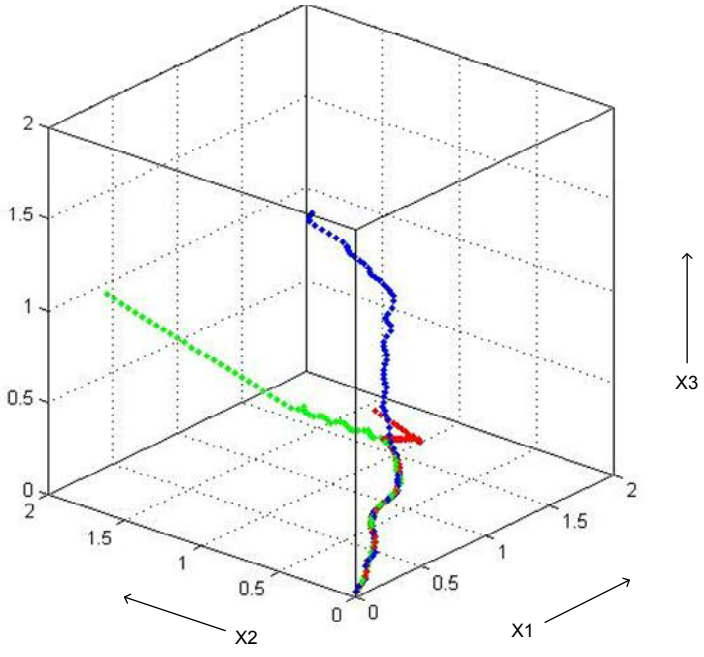


Figure 4.4: Particle's Paths to Optimum

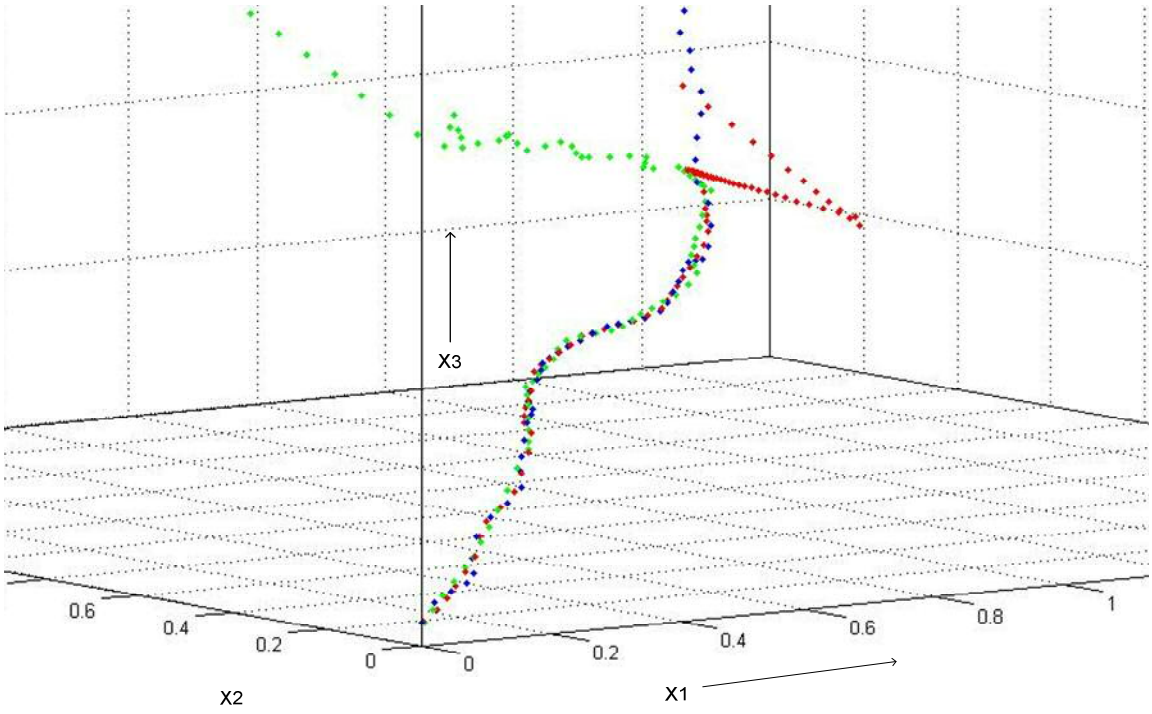


Figure 4.5: Close Up Path to Optimum

Notice how the 3 particles quickly converged to a fairly similar position and then moved as a group toward the optimal solution of $[0.0\ 0.0\ 0.0]^T$. By modifying the parameters in velocity and position update equations, a different type of behavior could be obtained, if desired.

The final best position of the three particles, or solutions, after 100 iterations is:

$$x_1 = .0187$$
$$x_2 = .0000$$
$$x_3 = .0440$$

This yields a function value of:

$$f(\mathbf{x}) = .0023$$

This simple 3 dimensional example demonstrates how PSO works for continuous functions, such as the space of the previous example. Unfortunately, as previously mentioned, the Job Shop Problem solution does not live in continuous space, but

combinatorial space, which can be represented by permutations. In the following chapter, Chapter 5, this issue is dealt with when PSO is applied to the JSP.

CHAPTER FIVE

PROPOSED JSP/PSO ALGORITHM

Since the conception of this optimization strategy, the PSO algorithm has been applied to many optimization problems, including permutation problems like the Traveling Salesman Problem. However, PSO in its purest form is not well suited for searching the permutation space of numbers. The particles in the PSO algorithm are designed to explore continuous space, which was seen in the example of the previous chapter. Therefore, extra care must be taken to adapt the PSO algorithm to permutation space. To the best of my knowledge no one has tried to apply Particle Swarm Optimization (PSO) to the traditional Job Shop Problem, save for [24] who applied a hybrid PSO/SA technique. The PSO has been applied to other scheduling problems, which are similar to the JSP. These other methods were discussed in Chapter 3 of this thesis. While there has been research conducted on the application of the PSO algorithm to the Traveling Salesman Problem, there has not been much research on the application of the PSO algorithm to the JSP problem. Therefore, this research is focused on exploiting the advantages of the PSO algorithm for the optimization of the JSP.

The proposed JSP/PSO Algorithm will use a set of permutations to *indirectly* represent a solution to a JSP. There are two obstacles that will prevent immediate use of the PSO to the JSP. One obstacle, which should be obvious, from studying the equations of the

traditional PSO algorithm in Figure 4.1, is that a permutation of numbers which can represent a solution to the JSP (in the context of a scheduling algorithm) can not be optimized directly with the PSO governing equations. *There has to be a way to transform the continuous space into permutation space*. The other obstacle is deciding how to transform this permutation(s) of numbers into a schedule, or in other words deciding what kind of scheduling algorithm will be implemented. These two distinct processes are shown in Figure 5.1 on the next page of the entire process for clarification. The space transformation process is shown in blue, and the decoding of the permutation into a schedule is shown in green. This chapter will discuss in detail how these two previously mentioned "obstacles" are addressed in the proposed JSP/PSO Algorithm.

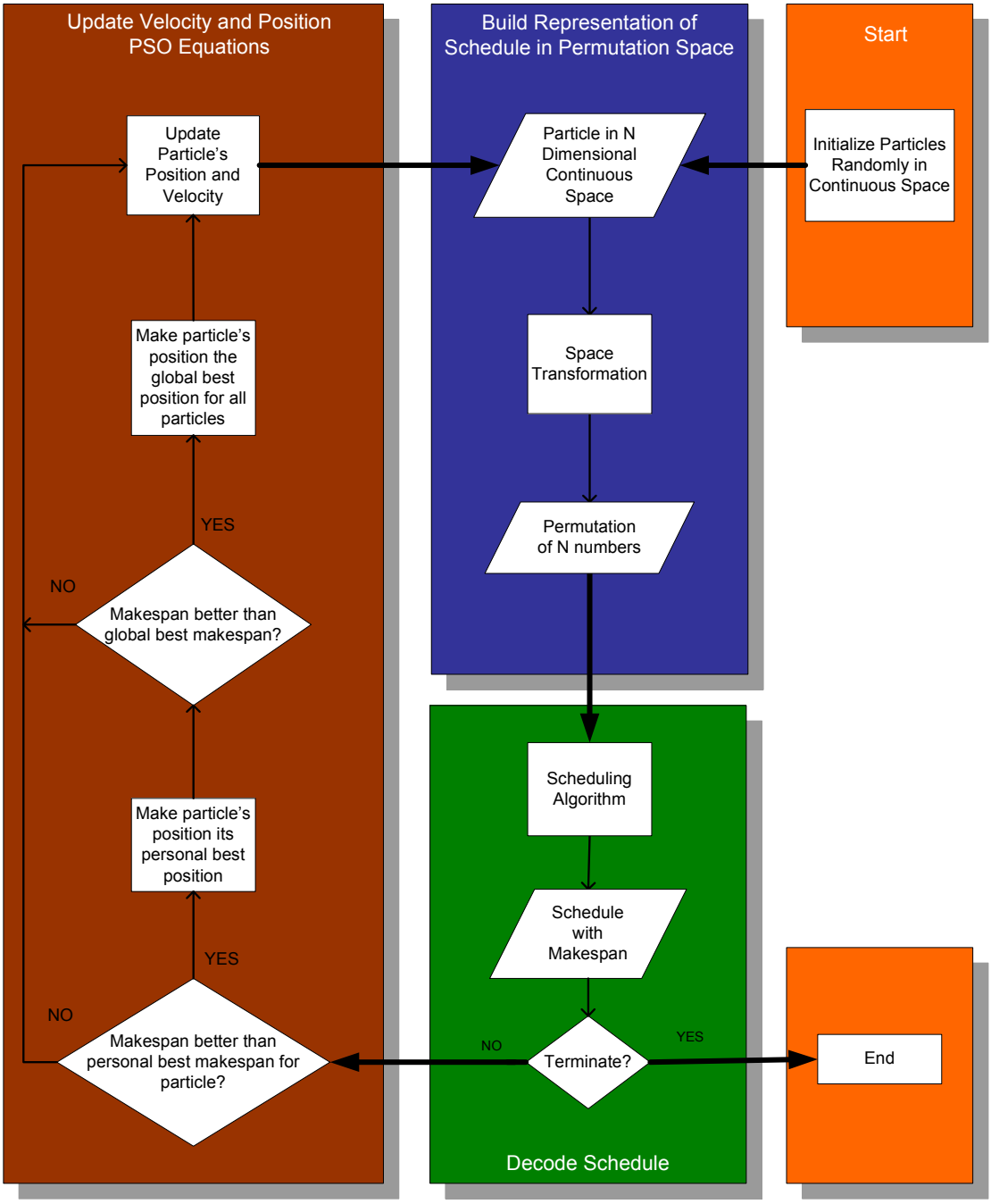


Figure 5.1: Block Diagram of PSO/JSP Algorithm

5.1 From Continuous Space to Permutation Space

In order to transform the continuous space of the particles into permutation space (shown in blue in Figure 5.1) the Greatest Value Priority Rule, or GVP [15] was implemented. The GVP rule is simply the assignment of each dimension or component of a particle in continuous space an integer index. The sequence of these assignments put together make up the permutation. So if there are *n* dimensions in continuous PSO space, the permutation will be *n* values long. To determine the permutation a particle represents, the dimension that has the greatest magnitude is given a 1 value. Then the dimension of the particle that has the next greatest magnitude is assigned a value of 2. This process is repeated for all dimensions of the problem. In the figure below, Figure 5.2, a particle in 3 dimensional space is transformed into a permutation using the procedure above.

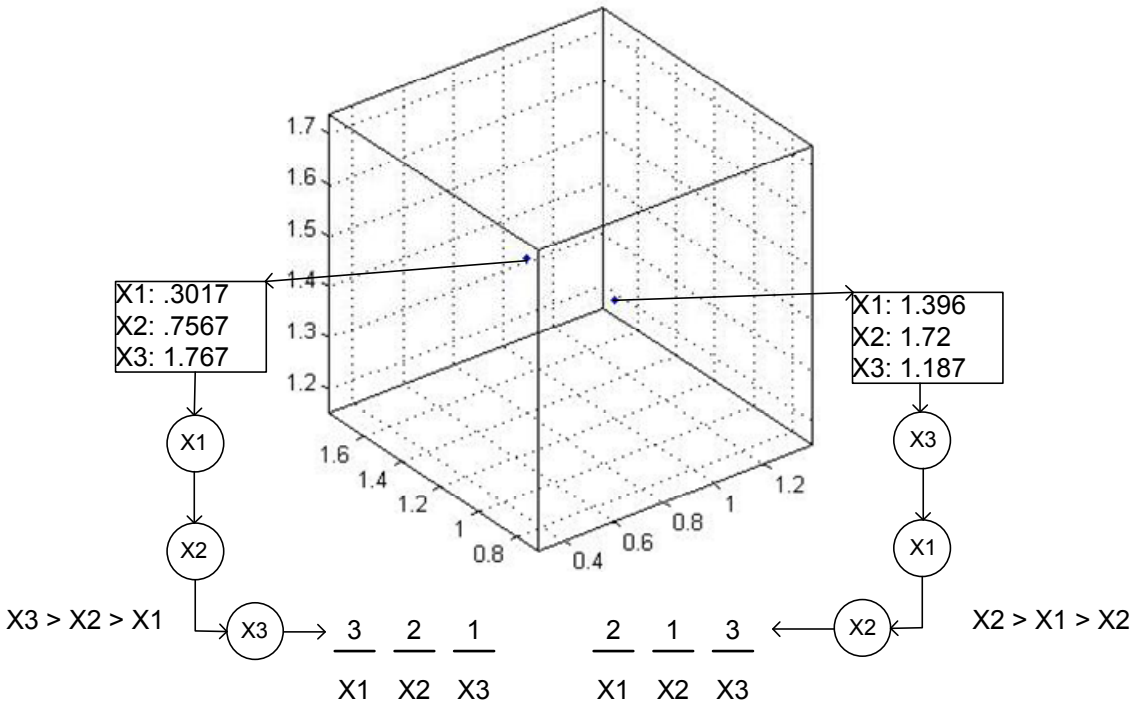


Figure 5.2: Space Transformation

This permutation of 3 numbers could easily represent processing priority for a machine in a JSP with 3 jobs. Obviously, for a JSP with *n* jobs, a particle with *n* dimensions in continuous space could be used to determine a *single* machine's schedule in a JSP. This is how the GVP rule works. The rest of the PSO can behave the same way as was discussed in the example of Chapter 4.

5.2 From a Permutation to a Schedule

Now, that the space transformation obstacle has been discussed, the next obstacle can be explained. This next obstacle has to do with how exactly the schedule will be represented by permutation of numbers, which now can be optimized by the PSO. This process is shown in green in Figure 5.1. This section will address this issue specifically. The advantages and disadvantages of two representation methods will be discussed, permutation with repetition and priority list representation. The JSP/PSO Algorithm was influenced by both of these representation methods.

5.2.1 Permutation with Repetition Representation Permutation with Repetition represents a job shop schedule by using the job number in a sequence that is $(n \times m)$ long. Each job number is repeated *m* times, or the number of machines there are in the JSP. Since each job has a certain technological sequence, or processing order, each instance of the job number in the sequence represents the next operation to be processed in that technological sequence. By using a scheduling algorithm that simply schedules operations specified by the permutation as early as possible, a semi-active schedule can

be generated. This process was shown and briefly explained in Chapter 2, but also shown below in Table 5.1 and Figure 5.3 for convenience.

Table 5.1: Simple Job Shop Problem

Job 1	1 (3)	2 (5)	3 (2)
Job 2	1 (5)	3 (1)	2 (4)
Job 3	2 (4)	1 (2)	3 (1)

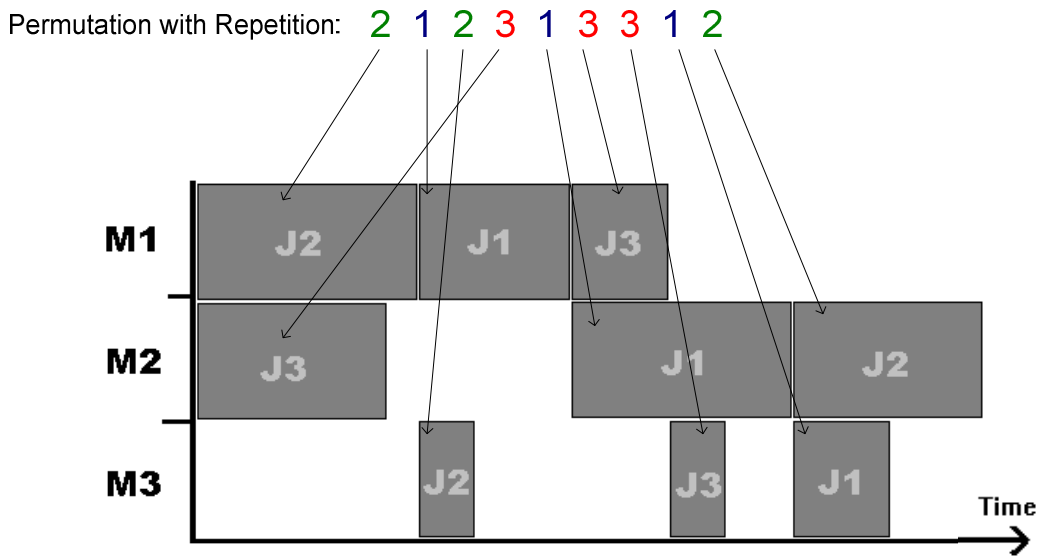


Figure 5.3: Permutation with Repetition Schedule

This is an easy way to represent and decode a schedule. However, there are a couple of drawbacks. One drawback is the fact that this procedure is not guaranteed to produce an active schedule. It is only guaranteed to produce a semi-active schedule, which contains the classes of active and non-delay schedules. It is obvious that the schedule in the example above is not an active schedule, rather semi-active. By simple left shifting the 3rd operation on Machine 2 with the one before it, a schedule with a shorter makespan is produced. The other drawback is that many of the same permutations will produce the

same schedule. For example, the last two job numbers could be switched in the above permutation and the same schedule will result. This is because both of those job numbers are representing operations that require different machines. The lack of one to one matching is not a surprise when considering that the space of permutations with repetition is much greater than the number of possible semi-active schedules, feasible or infeasible. In fact the permutation space of n numbers is $n!$. However, if the any of the numbers in the permutation repeat the search space is reduced by a factor of the repetition number factorialized. So, for a particle to represent a schedule in Permutation with Repetition form, the particle will have to have $n \times m$ dimensions, which will consist of n numbers that each repeat m times. Which in turn means that the search space for a particle becomes

$$\frac{(n \times m)!}{(m!)^n}.$$

For example, a (10×10) JSP has a combinatorial search space of

$$\frac{(100)!}{(10!)^{10}} \cong 2.357 \times 10^{92}.$$

This is huge, to say the least. This method seems like overkill especially when many of the permutations will also result in the same schedule, and many will result in semi active schedules as well. However, the benefits of using Permutation with Repetition might out weigh these drawbacks. The main benefit of using this representation is that the corresponding scheduling algorithm is very fast and simple, and therefore not computationally expensive. This makes sense, because to build semi-active schedules operations are simply scheduled at the earliest starting time, which is easily done when decoding a schedule represented by a permutation. Limiting your search space to active

schedules usually involves a scheduling algorithm that looks ahead in time, which adds computational time and complexity. The GT Algorithm is a good example of this, which can be used with priority lists.

5.2.2 Priority List Representation It is actually possible to reduce the size of the search space by using a priority list for each machine instead of an $n \times m$ set of numbers. In a priority list representation each machine has its own priority of jobs that it *prefers* to process of length n. This gives a combinatorial space of $(n!)^m$. For a (10×10) JSP, this is equal to $(10!)^{10} \cong 3.9594 \times 10^{65}$. This is 5.9531×10^{26} smaller than the search space of Permutation with Repetition. This reduction in search space should be helpful when the appropriate scheduling algorithm is used to decode each machines priority list into a schedule. The scheduling algorithm usually used with a priority list is the GT Algorithm. This algorithm is discussed in Chapter 2. By using the GT Algorithm, I am guaranteed to build active schedules. So not only is the search space reduced, but I can build active schedules as well. An example of a priority list is given in Table 3.2 in Chapter 3. However, to build active schedules, decisions must be made while the schedule is being generated, which require looking ahead in time. Looking ahead in time is surprisingly computationally expensive, because it requires many programming loops. For example, the GT Algorithm looks ahead a certain amount of time from the current period to see which operations will become available for processing on a given machine. A decision must be made as to which operation to process from those that need that machine. This process must happen for each operation in a JSP, and can take a significant amount of

time. This could make using a priority list not worth the benefit of the reduced search space property of the GT Algorithm.

5.3 The JSP/PSO Idea

The research objectives of this thesis were two fold. The first was to apply a well known successful meta-heuristic optimization method, the PSO Algorithm, to the Job Shop Problem. The second stemmed from the first, which was to use it to solve the JSP indirectly with a priority list because of the reduced search space as discussed previously. This objective stemmed from the first because the traditional way to use PSO to solve problems is to make each particle a solution to the problem at hand. The easiest way to make each particle a solution to a Job Shop Problem would be to have each particle have $n \times m$ dimensions. Then by using the space transformation procedure discussed earlier, the particle could then be turned into a permutation with repetition schedule representation. This way each particle represents a possible valid schedule. However, even for a small (10×5) JSP, a particle would have to have 50 dimensions, and the search space becomes $\frac{(50)!}{(5!)^{10}}$ which is about 4.912×10^{43}. And as shown before, the same problem's search space represented by a priority list is $(10!)^5$, or 6.2924×10^{32}. So, the second research objective became using the priority list representation instead of permutation with repetition.

Unfortunately however, because of the way the PSO algorithm works, i.e. information is extracted solely through its position in n dimensional space. So, in order for each particle

to represent a complete solution, every operation in a JSP must get its own dimension in a particle. This means that even if a priority list representation is used, there still must be a dimension for every number in the priority list. This does not ultimately reduce the search space. This dilemma is shown graphically below in Figure 5.4.

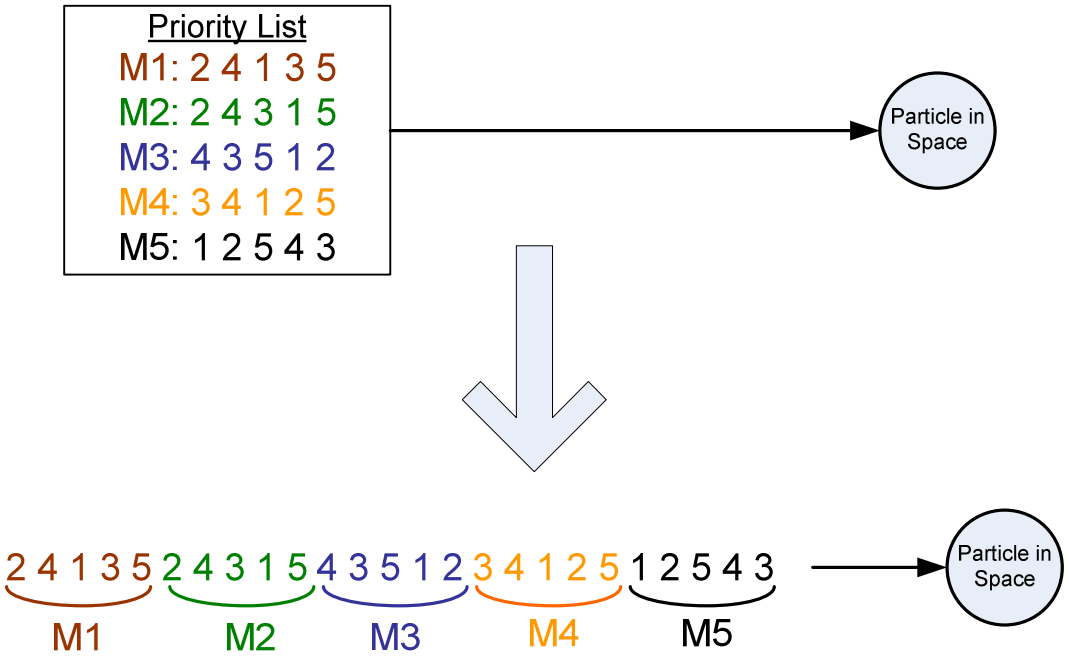


Figure 5.4: Priority List to a Particle (a)

Several creative ways could be developed to make this transformation, but in the end a permutation of length $n \times m$ will be required. The figure above shows how having a partitioned permutation of length $n \times m$ that divides up into a priority list does not reduce the search space to $(n!)^m$, as was the goal. Therefore, some other course must be taken.

To accomplish this task, the JSP/PSO encodes only part of a solution into a single particle, and several particles will combined together to make a solution. Since the goal

is to solve the JSP by optimizing a priority list, the priority list will be divided by the number of machines in the JSP, and a single particle will represent one machine's priority list. So for a (10×5) Job Shop Problem (10 jobs and 5 machines) there will be 5 particles that combine to make a solution, each one representing a processing priority of length *n*. See Figure 5.5 for an illustration.

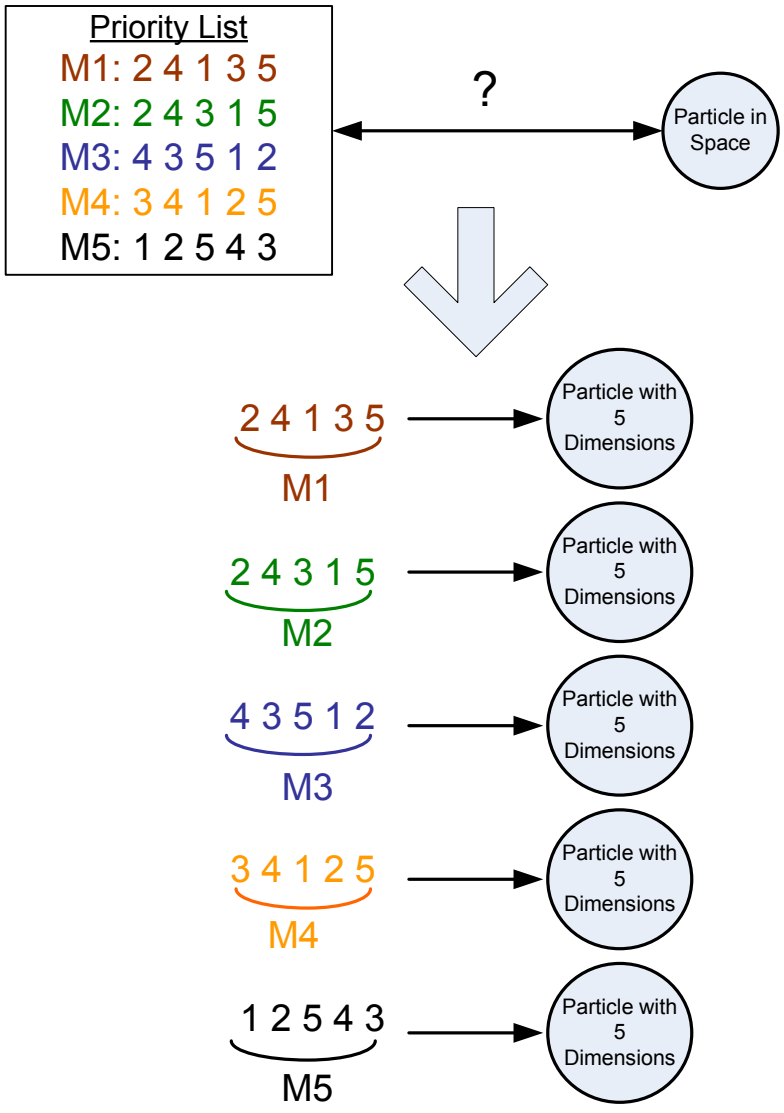


Figure 5.5: Priority List to Particle (b).

Technically, the arrows should be pointing from the particle to the permutation since the permutation will come from the particles position in space, but this figure is simply attempting to illustrate the space division concept. A better more detailed illustration of the whole concept is presented in Figure 5.6.

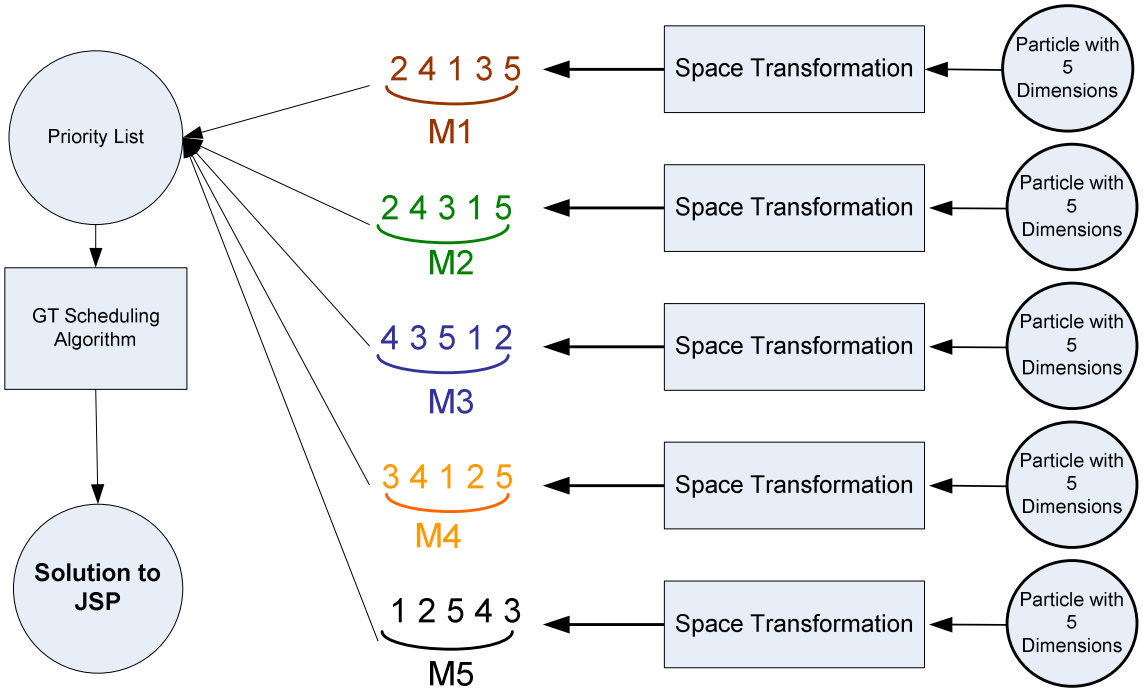


Figure 5.6: JSP/PSO Block Diagram

Using this technique, the search space has effectively become $n!$ for each particle, and taken together as a whole the search space is $(n!)^m$. Now, some questions arise at this point as to how the Particle Swarm Optimization will work in this fashion, because each particle only represents a partial solution to the problem. It is clear there must be several particles in a "swarm" so that a global best position can be shared with other particles. However, the global best position of the particles representing the operating priority for

Machine 1 is not going to be the best position for the particles representing the operating priority for Machine 2. Therefore, in the JSP/PSO there are *m* number of swarms, one for each machine. This is simply illustrated in Figure 5.7.

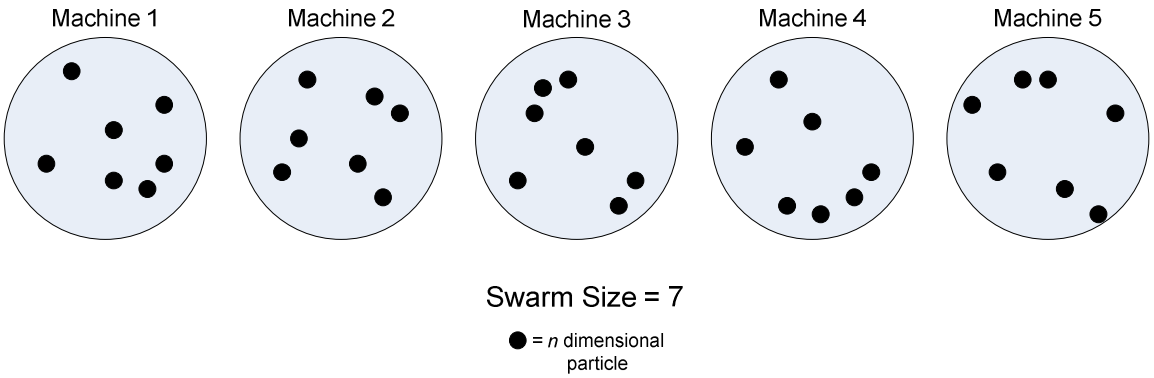


Figure 5.7: JSP/PSO Swarm for an *n x 5* JSP

Each machine in the JSP has its own swarm and thereby it's own global best for the particles to work toward. Which makes sense, because each machines priority list will naturally be different. Notice, that the swarm size is 7 even though there are actually 35 particles. Here's where things can get tricky, the personal best for each particle and the global best for each swarm are determined by the makespan of the solution to the JSP that they represent a partial solution to, *1/m* to be exact. As stated earlier, one particle from each swarm together represents a solution to the JSP problem. *Which particles are combined with which particles to form a complete solution is a critical variable in this process*. In the JSP/PSO Algorithm one particle in each swarm is "linked" to one other particle in every other swarm and they remain "linked" together through the course of the optimization. The linking is done at initialization and it is done randomly. Of course, there is no actual real link (they don't share information), basically this means that the

same particles from every swarm will always come together to form a solution. In other words, "linked" particles share the same "fitness", so to speak, every iteration. This is shown graphically below in Figure 5.8.

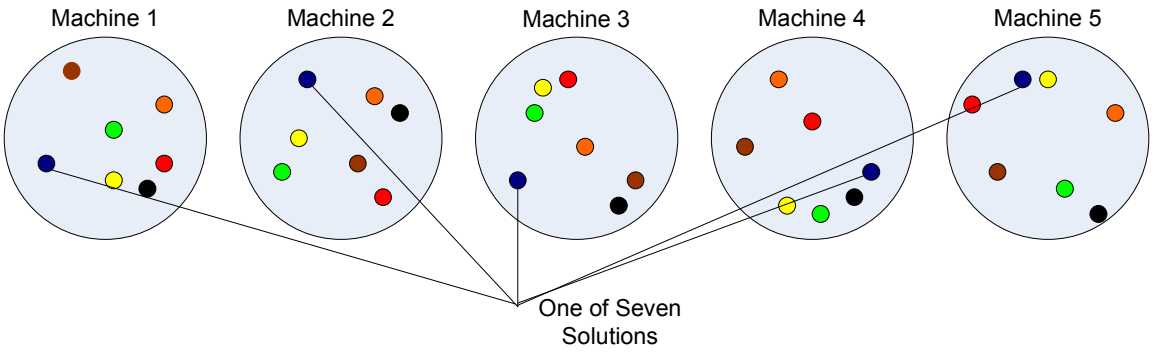


Figure 5.8: JSP/PSO Linking

This is an important concept to understand. Essentially, these particles move with no knowledge of the other swarms, but *their* "fitness" or *their* best position in space is definitely affected by the position of the other particles "linked" to it in the other swarms.

At first, this may not seem like it should work as an optimization method, and this space division concept may not work well for other meta-heuristic methods, but because the PSO requires that the particles store knowledge of their personal best position and the global best position in space the PSO can be used in this situation. When a better makespan is found by a certain set of linked particles, *m* number of separate best positions are recorded, one for each of the *m* different swarms. If this makespan happens to be the best one obtained so far, then each swarm notes that respective position obtained by the particle in their swarm as the global best. *Just because those particles have no knowledge of the other swarms doesn't mean they won't work together indirectly to*

explore areas that brought them the best fitness in the past. This idea is attempted to be shown graphically in Figure 5.9 below. The arrows signify a pull in that direction.

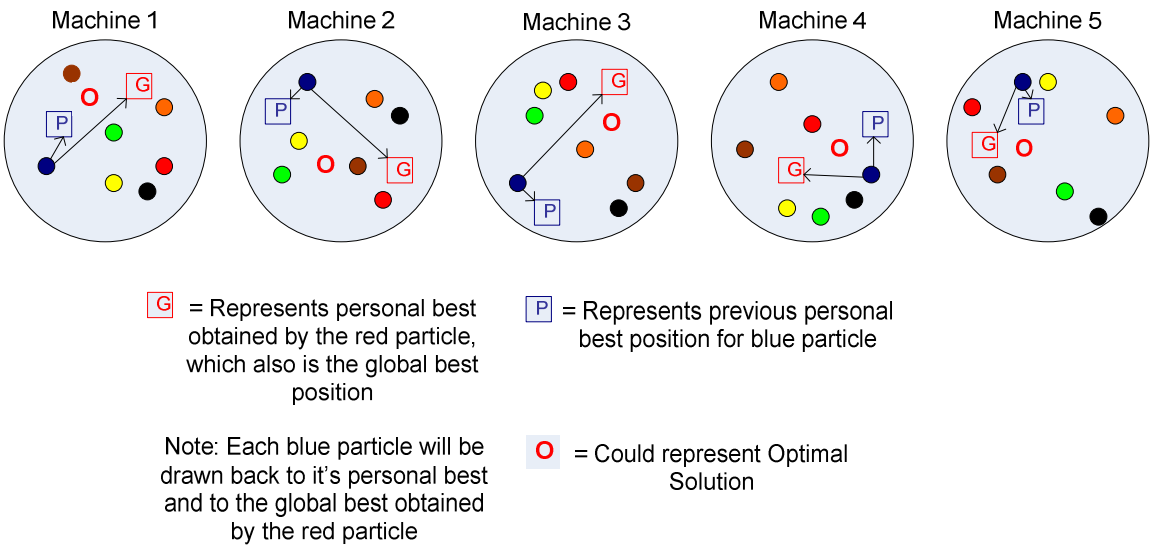


Figure 5.9: JSP/PSO Personal and Global Best Position Example

It might be easy to understand now why particles should for the most part remain linked to the same particles in other swarms throughout the optimization process. It should be intuitively apparent that "re-linking" particles whose personal best positions were found as a part of another set of linked particles would not be helpful. However, under the right conditions this could be an interesting and possibly beneficial mutation operator.

5.4 Why PSO?

As stated earlier, I believe that PSO is particularly well suited to be the meta-heuristic method to optimize the JSP in the fashion described in the previous section, specifically the division of the search space by the number of machines in a particular JSP. The PSO algorithm makes this possible by the knowledge each particle contains. The ability to

simply give the particles a search direction by the parameters of the PSO to either explore new territory or to exploit previously known territory makes the optimization method even more controllable. This makes the particles to work together in an indirect fashion. Also, the fact that the particles aren't assigned a specific fitness, but simply contain knowledge of where good positions are in their search space makes this way of solving the Job Shop Problem realistic. Since particles in other swarms will be drawn to the areas in their space that corresponded with the good positions of the particles in the other swarms. It would be interesting to see how other proven meta-heuristic optimization methods would perform in place of the PSO for this optimization strategy.

Another reason, I believe that the PSO is worth exploring as a possible optimization method in the JSP is because of the unique ability it has *in combination with the space transformation technique* to search permutation space. The space transformation from continuous space to permutation space, as explained earlier in this chapter, has a unique ability to search permutation space. For example, let's say for a (6×6) JSP (6 Jobs, 6 Machines) a particle representing $1/6^{th}$ of a priority list has the following dimensional and permutation values shown in Figure 5.10 below.

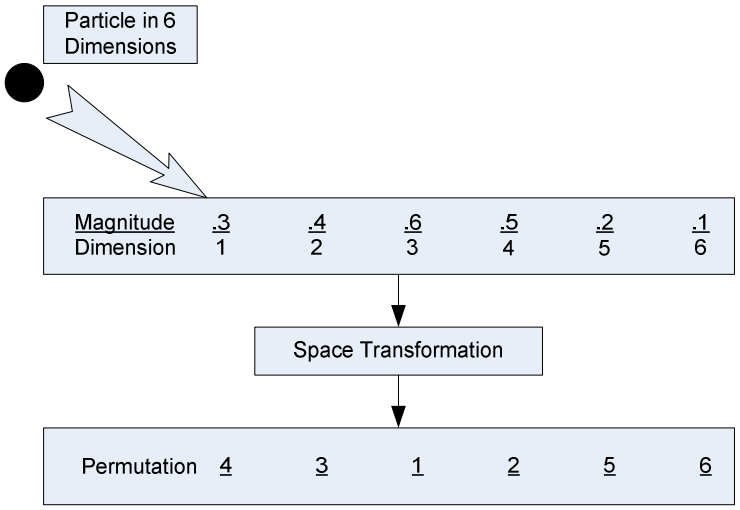


Figure 5.10: Six Dimensional Particle in Continuous and Permutation Space (a)

Now, for simplicity lets assume that the particle is traveling significantly in only one direction or one dimension, say dimension 3. If the velocity and position update PSO equations change this dimensional value significantly from .6 to .15, then a drastic change has taken place in the permutation space, while the particle in the continuous space remains relatively the same. This is shown in Figure 5.11 below.

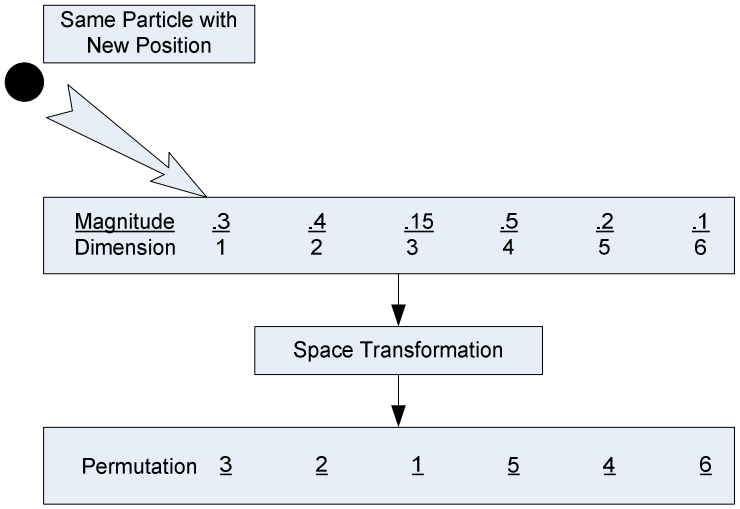
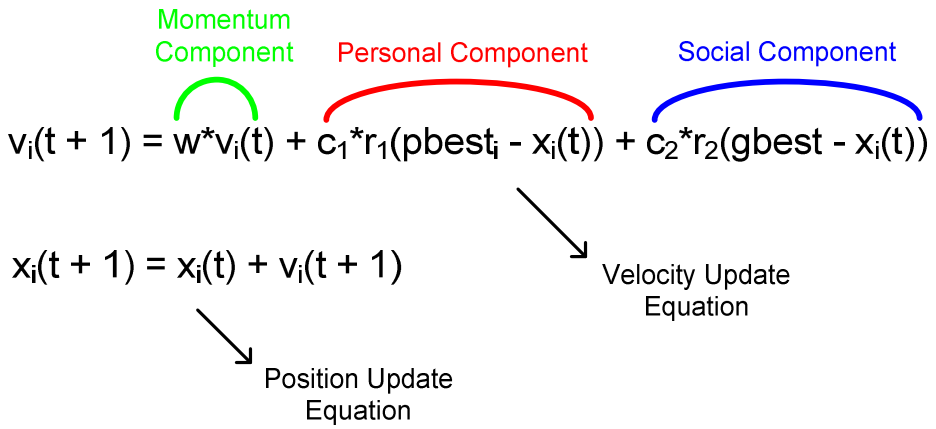


Figure 5.11: Six Dimensional Particle in Continuous and Permutation Space (b)

The permutation has changed fairly drastically by moving the particle in only one direction. This should give an example of the kind of ability that PSO along with the GVP Space Transformation Rule has to search permutation space. The ability of a particle to move significantly in any dimension can be govern by the values of the user defined coefficients of the PSO equations presented in the previous chapter, but shown below for convenience. Also, by imposing maximum velocity constraints, we can effectively limit the change in any particles position from iteration to iteration.



- i = 1, 2…p , p = number of particles in swarm
- pbest is the best position found by that particle so far
- gbest is the best position found by any particle so far
- v = velocity of particle in a single dimension
- x = position of particle in a single dimension
- t = iteration number
- w is the inertial constant
- c_1, c_2 are acceleration constants
- r_1, r_2 are random numbers evenly distributed between [0,1]

Figure 5.12: PSO Velocity and Position Update Equations

The equations shown above in Figure 5.12 are for one dimension only, the current dimension *i*. It is easily realized that the way the programmer governs these equations with the constants can drastically affect the results of the optimization. In fact, the

outcome of the optimization, or the priority list, is very sensitive to the way these equations behave. The specific parameter values used in the optimization routines are discussed in the next chapter as well as the results of the JSP/PSO Algorithm on test bench Job Shop Problems.

CHAPTER SIX

RESULTS

In this chapter the relevant details of the computer program used for the proposed JSP/PSO Algorithm are discussed, and a brief analysis on how they affect the performance of the algorithm is given. Also, the performance of the JSP/PSO Algorithm on many standard JSP test bench problems are presented for scrutiny. Finally, these results are compared to results obtained from other JSP optimization methods in recent literature.

6.1 JSP/PSO Program Specifics

The JSP/PSO was written and tested in MATLAB programming language. There are two parts to this algorithm, the scheduling part and the optimization part, or the PSO part. Much of both the scheduling part and the optimization (PSO) part have been generally discussed. However, in the following two sections more details are disclosed as to exactly how this program operated in these two areas.

6.1.1 PSO Program Specifics Obviously, the PSO user defined parameters control exactly how the particles behave, which can have an effect on how the optimization

works in the long run. The user defined constants and parameters can have a drastic effect on the outcome of the problem being optimized. Therefore, a lot of time was spent fine tuning the PSO part of the program to achieve the optimal values for this type of PSO application. In this section the details of how the PSO was programmed to behave are presented.

First and foremost, a swarm size of 20 is used for all test bench problems. Some JSP problems are easy enough that only 10 particles could be used in a swarm, but most problems aren't that simple. Swarm sizes of more than 25 did not seem to improve the results appreciably, if at all, to justify the increased program running time. The particles were confined to an arbitrary space of -.5 to .5 in all dimensions. If a particle travels, or attempts to travel out of the allowed space, the particle can not simply be assigned the maximum value or minimum value whichever the case may be. This is because of the numerous dimensional values that might end up with the same value of .5 or -.5, which will then make the permutation from space transformation procedure meaningless. This was dealt with by only moving a particle half way to the boundary if it tried to step outside the PSO search region. This avoided the problem of particles lying on the boundary in multiple dimensions and adequately allowed the particles to move freely, but also stay inside the search region. The reason this space is arbitrary is because it matters not what the specific value of a particle is in a given dimension, only its relationship to its other dimensional components. This is explained in Chapter Five in more detail. A maximum and minimum velocity was also placed upon the particles of .5 and -.5 respectively. This can also be thought of as a maximum step size for a particle in a given

dimension for one iteration. This velocity restriction is important, because it can control how fast a particle can move and thereby affect how quickly any dimensional component can become less than or greater than the other dimensional components of the particle, which in turn determines the permutation that particle represents. If a particle attempted move faster than this maximum or slower than the minimum velocity, it was simply assigned the appropriate maximum or minimum velocity value.

A mutation operator was also utilized in this program. In meta-heuristic methods, mutation operators are used to help escape local optima, and ensure global coverage of the search space. Typically, mutation operators randomly manipulate (in a specified manor) what can be called the genotype of the problem, in this case a particle. Mutation usually occurs at a very small rate, because using them too much would destroy any good information contained in the population or swarm of individuals that might have already been acquired through the optimization process. In this program, a particle has a 5% chance of being selected for mutation. The mutational operator randomly selects two dimensions of a particle (remember each dimension represents a job place in the priority list). The value of the first dimensional component is removed from the particle and reinserted at the place of the second randomly selected dimension. An illustration of this simple process is provided in Figure 6.1 below.

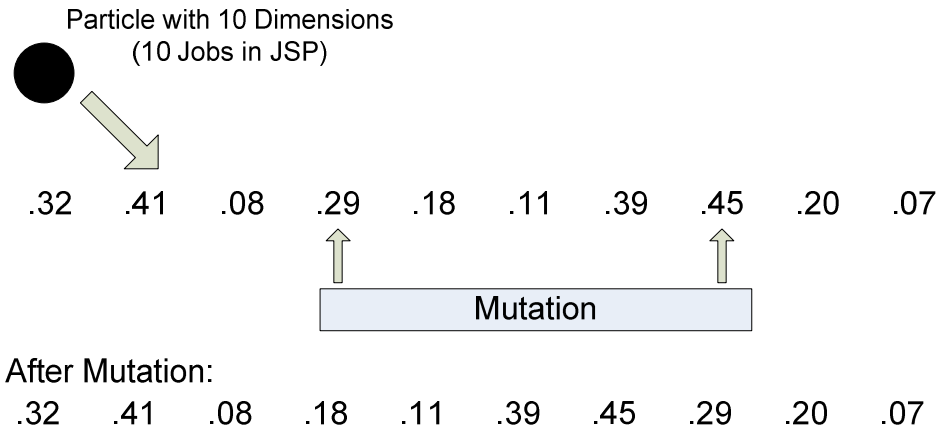


Figure 6.1: Mutation Example

This mutation operator should help ensure that a good global exploration of the permutation space is achieved by the PSO algorithm through the course of its optimization.

Of course, the mutation operator is not the only way particles are able to explore the search space in the PSO algorithm, the concept of the PSO algorithm are that particles "swarm" through the search space, but remember their best position and the global best position. As explained in Chapter 4, how "free" these particles are able to move away from global best and personal best positions depends upon the C_1, C_2 and W constants that are found in the velocity update PSO equations. The PSO equations are presented again below for convenience in Figure 6.2.

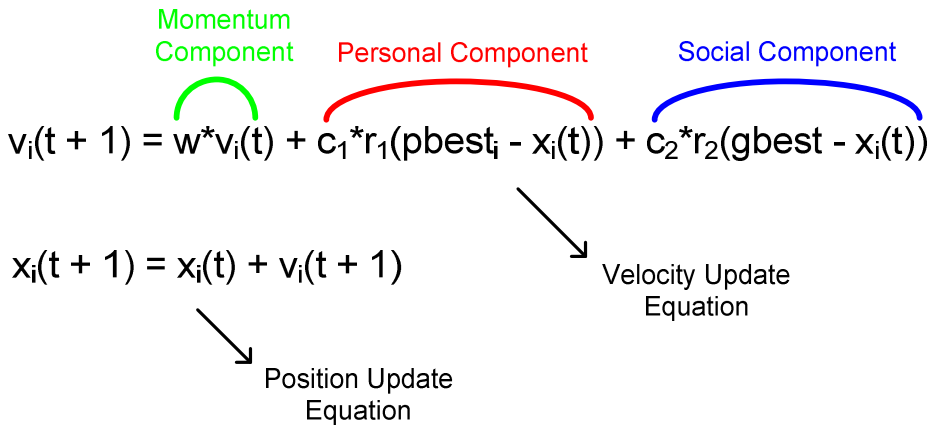


Figure 6.2: PSO Velocity and Position Update Equations

For many PSO applications C_1 and C_2 are simply set to a constant value throughout the optimization process. The number 2 is common to use for both C_1 and C_2, this means that the particle will have an equal pull to the global best found solution and its personal best found solution. Some researchers have taken this a step further and programmed C_1 and C_2 to change with respect to the number of iterations. The equations presented below are one possible way of doing this.

$$C_1 = (C_{1Final} - C_{1Initial}) \times \left(\frac{iterationMax - iteration}{iterationMax} \right) + C_{1Final}$$

$$C_2 = (C_{2Final} - C_{2Initial}) \times \left(\frac{iterationMax - iteration}{iterationMax} \right) + C_{2Final}$$

$$W = (W_{Final} - W_{Initial}) \times \left(\frac{iterationMax - iteration}{iterationMax} \right) + W_{Final}$$

By making $C_{1Initial}$ a larger number than C_{1Final} and $C_{2Initial}$ a smaller number than C_{2Final}, particles will be allowed to explore a larger territory at the beginning of the program and then converge near the global solution toward the end of the program. Obviously, the same can be done with the velocity constant W. This constant controls the impact of the previous velocity of the particle. Dynamic constants were used as explained above, but

they were "cycled" over and over through the course of the optimization. Instead of the constants taking the entire number of total iterations to go from an initial value to a final value, they make this transition several times through the course of the iterations. To do this, the common modulus operator was used. The main reason this was done, among others, was to ensure that particles could escape a local optimum through the course of the program simulation. The desire to have the particles converge to the best found solution at some point is desirable, because the true optimal solution may very well lie somewhere near a best found solution. However, schedules that are very close in makespan can be extremely far apart in regard to the sequencing of operations, meaning to go from a schedule with a makespan of x to a makespan of $x-1$ the particles may have change their dimensional values significantly. However, the particles won't be able to do this very well if the constants of the PSO equations have already de-emphasized exploration by decreasing the C_2 parameter and increasing the C_1 parameter. It is surely conceivable that cycling these constants will help in this regard, because cycling allows the particles to explore the search space more freely at certain times during the optimization. This cycling method probably is not necessary in many applications of the PSO, but it was found through the course of this research that since the JSP is such a huge combinatorial problem that anything to aid the particles in escaping local optima is helpful.

However, there is one problem with this idea. Even if the constants are cycled to allow the particles to explore again, if they've all been drawn to the same area by a previous large C_2 constant at the end of the previous cycle, this hardly does any good. This is why

in the JSP/PSO the C_2 global starts out small and does not increase much through its cycle. This way particles are only *slightly* pulled to the global best during each "cycle", and then allowed to explore again when the C_1 constant gets reset to a larger value. Perhaps, this is better seen with examples of values that were used. The initial and final values used for the C_1, C_2 and W constants are shown below.

$$C_{1Initial} = 1 \qquad C_{1Final} = .25$$

$$C_{2Initial} = .01 \qquad C_{2Final} = .25$$

$$W_{Initial} = .9 \qquad W_{Final} = .8$$

Notice the large $C_{1Initial}$ value and small $C_{2Initial}$ value, which means a lot of emphasis is placed on personal exploration at the beginning of each cycle. Also notice, how the global best constant, C_2, does not ever have a greater value than the personal best constant, C_1. They only equal each other for one iteration each cycle. This means that for one cycle of the constants the particles will not be "drawn in" too much to the global best search space per cycle, but will eventually get there through several cycles. Essentially, if there are enough cycles the particles will eventually be "reigned into" the global solution area, after going through periods of exploration and convergence in other areas of the search space. The particles search behavior are fairly sensitive to the W constant, because it is less than 1 so it has the effect of slowing down the particle over the course of several iterations. It turns out that the W constant did not need to change too much through the course of the optimization, because it always helped the optimization process to let the particles have a decent amount of momentum.

This method of cycling the constants C_1, C_2 and W did improve the results that were obtained through the simulations. Through the course of all testing, a cycle factor of 200 iterations and maximum iteration limit of 2,000 was used. So the constants were reset a maximum of 10 times during one run of the program. To get an idea of how this method affected particles behavior in space, graphs are presented below to better convey the concepts that have been discussed so far. Figure 6.3 shows a single particles' velocity in a single dimension through iterations, and Figure 6.4 is a close up view of this velocity.

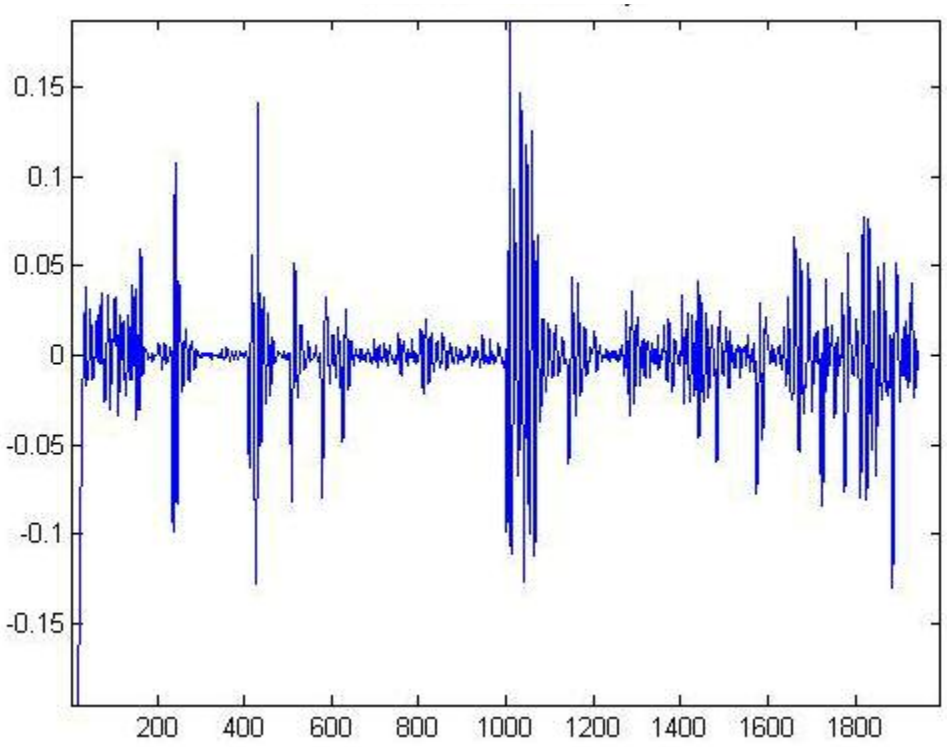


Figure 6.3: Plot of One Particle's Velocity

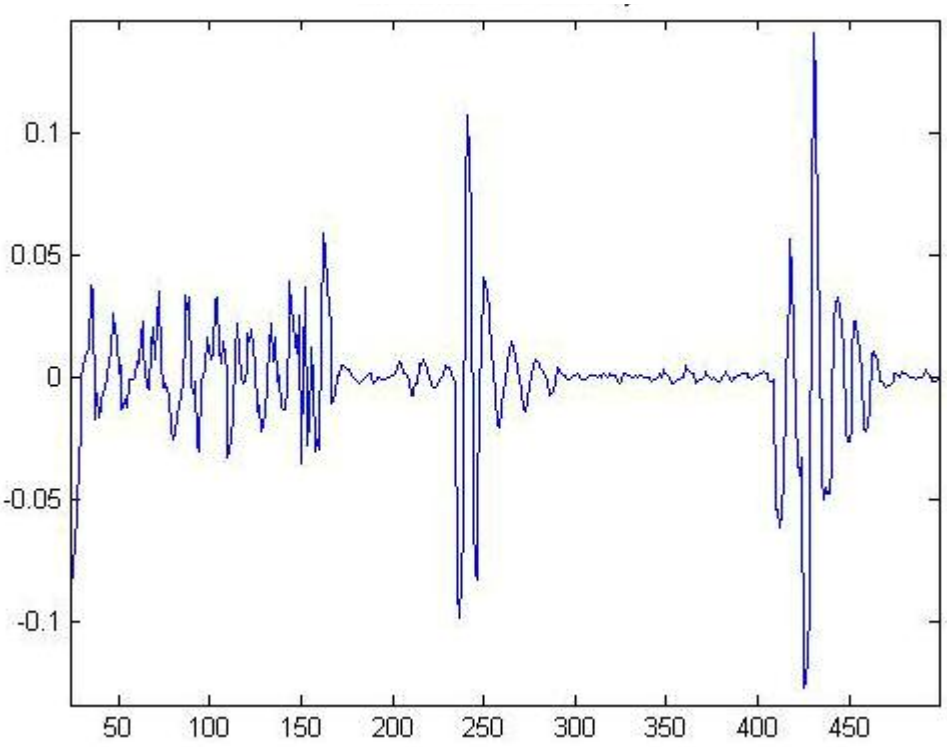


Figure 6.4: Close Up of One Particle's Velocity

Notice how this particle's velocity for the most part seems to spike around the iterations that are multiples of 200. This is because the cycle previously discussed is set to 200 iterations. Another way to illustrate how these "particles" are behaving is to plot multiple particles in the same swarm. In Figure 6.5 below, *one* dimension of *three* different particles are plotted versus the number of iterations. These particles are part of the same swarm, and are thereby optimizing the priority list for the same machine. Also a graph of the best found makespan, or objective function value, is presented in Figure 6.6 for comparison of the objective function value to what was happening in the search space at certain times in the optimization process.

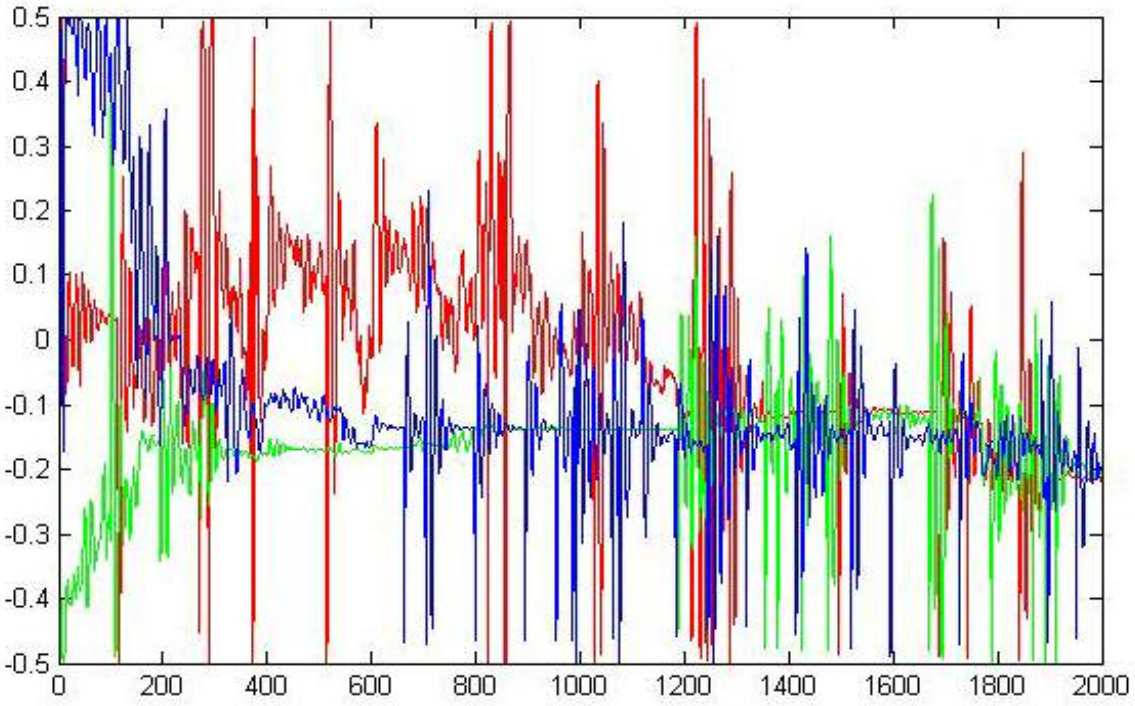


Figure 6.5: Three Different Particle's Movement in the Same Dimension

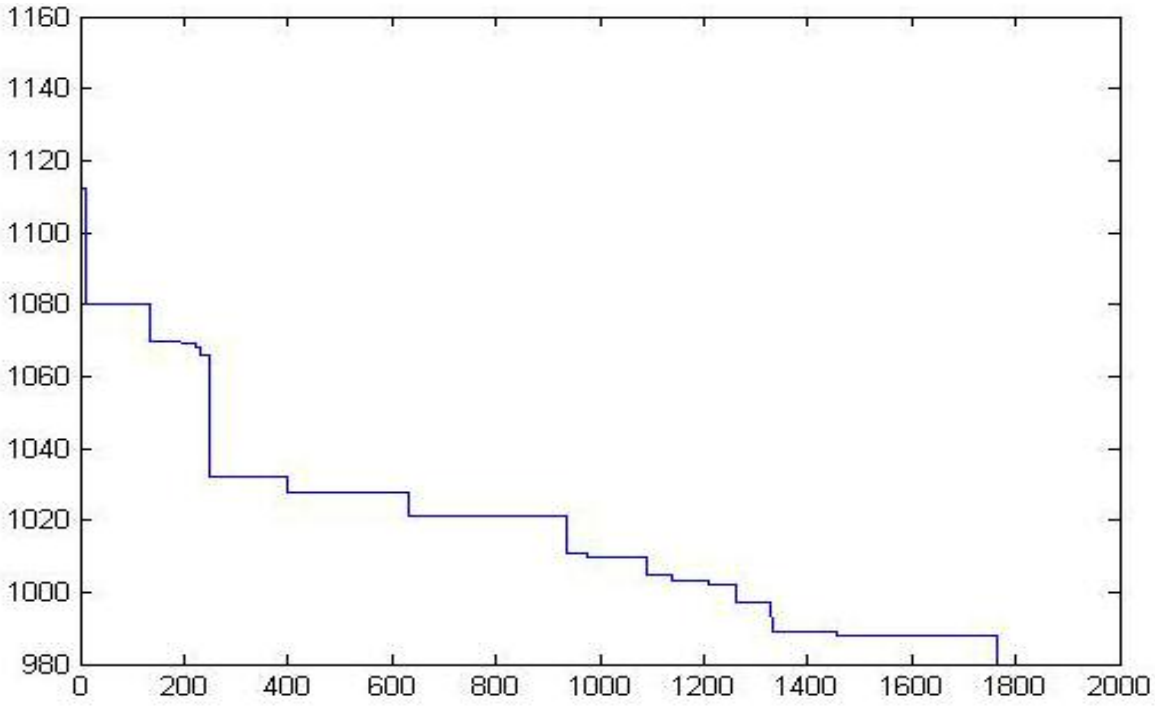


Figure 6.6: Objective Value (Makespan) versus Iterations

These plots might better help the reader understand what the particles are actually doing during optimization. Remember that Figure 6.5 represents one dimension in space, which corresponds to a certain job in the priority list of a JSP after space transformation has been completed. What is interesting about this graph is how all three particles start off in very different positions in space and are brought closer together throughout the course of the optimization, hopefully the final value they converge too is the optimal value for that dimension. Notice, however, that on every multiple of 200 iterations the three particles are brought slightly closer to what is the current best global position, then they are sent out to explore again. The particle plotted in red illustrates this concept fairly well. Every two hundred iterations the red particle is brought only slightly closer to what the global best position is, until around 1,800 iterations it has been brought in all the way. It appears the best found value for this dimension was around -.2. Of course, this specific value doesn't mean much, until we know what the values of all the other dimensions were at the end. The next figure, Figure 6.7, will help with this.

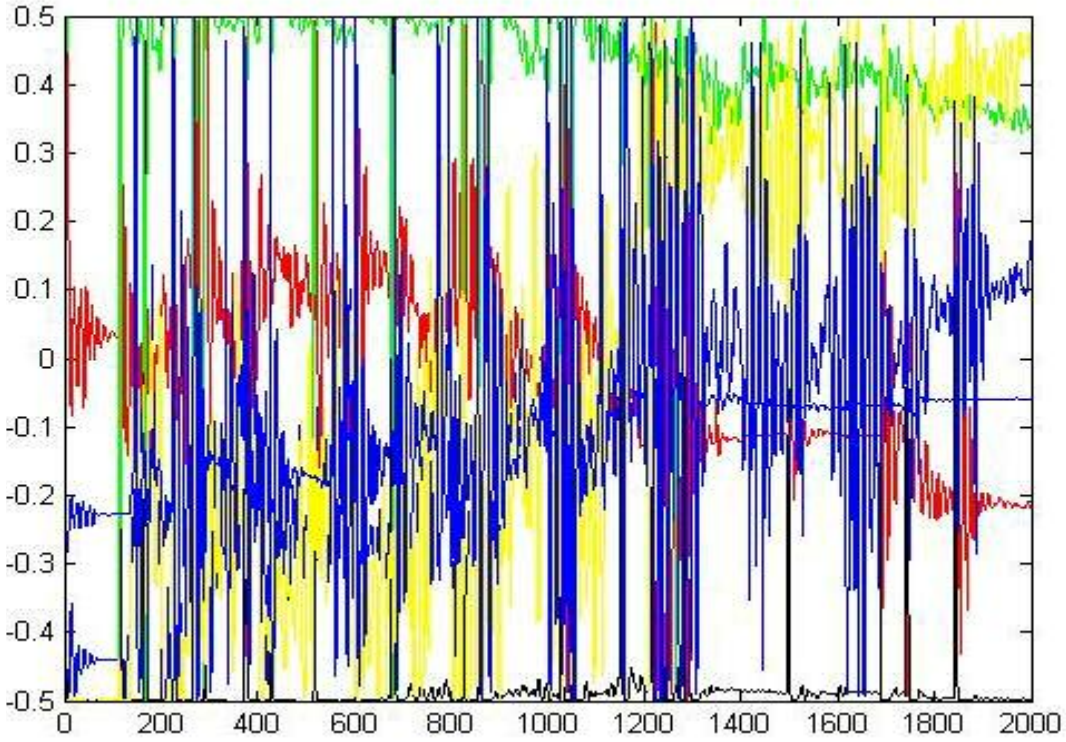


Figure 6.7: One Particle's Movement in Six of its Dimensions

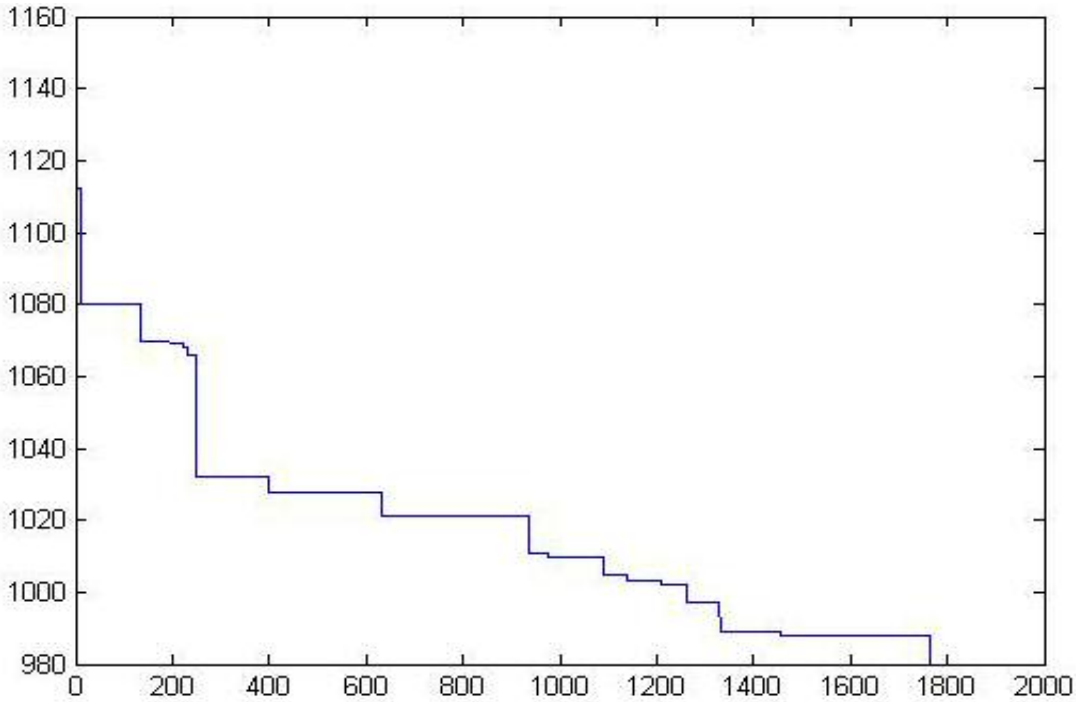


Figure 6.8: Objective Value (Makespan) versus Iterations

Figure 6.7 shows us what was happening to a single particle through the course of the optimization, six different dimensions versus iterations. Figure 6.8 allows us to see what was happening to the corresponding objective function value during these times. Remember from the space transformation procedure that the dimension with the greatest value will be given the permutation of 1, and so forth. According to Figure 6.7 the dimension represented by the yellow color would be assigned the permutation 1, and the dimension represented by the color green would be given the permutation 2, and so on. Looking at the graph, one can get the idea that the particle is simple "sorting" its dimensions out. To get a better idea of how fast these particles are moving the graph below, Figure 6.9, has been provided, which is a zoomed in version of Figure 6.7.

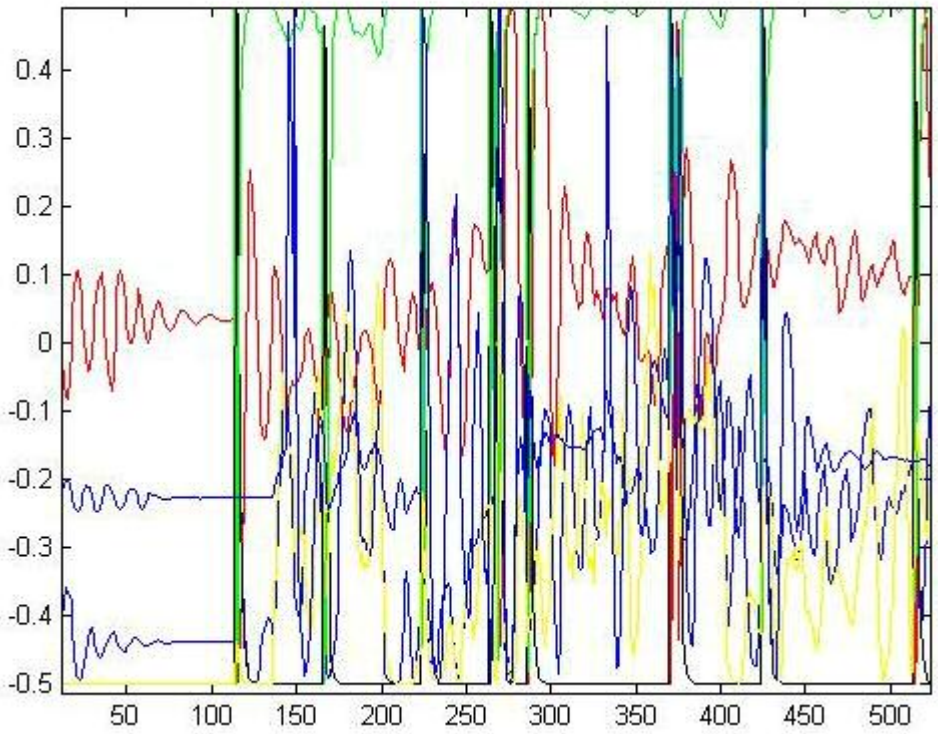


Figure 6.9: Close up of One Particle's Movement in Six of its Dimensions

As mentioned before, there are two parts to this optimization process, the meta-heuristic search method and the building of the corresponding schedule. The graphs and explanations in this section have attempted to show how the search part is conducted, or more specifically, how the particles have been programmed to behave. The result of this part of the program is a priority list for each machine. Once a priority list has been built the scheduling algorithm then builds a schedule. As will be explained in the next section this scheduling algorithm has direct influence on the ability of our optimization technique to find the optimal solution. The exact scheduling building specifics for my implementation of the proposed JSP/PSO are presented in the following section.

6.1.2 Schedule Building Specifics The scheduling algorithm commonly used along with a priority list is the GT scheduling algorithm, and both are used in the JSP/PSO Algorithm. The GT scheduling algorithm was discussed previously in Chapter 2. Basically, the GT scheduling algorithm builds active schedules by looking slightly forward in time. Active schedules are schedules in which no operation can be started earlier without increasing the processing time of any machine. The GT Algorithm is presented again below for convenience.

1. Let D be a set of all earliest schedulable operations in all job sequences not yet scheduled.

2. Let operation, O_{jr}, be the operation with the earliest completion time in set D. $O_{jr} = \min \{O - D \mid EC(O) \}$. Where j is the job and r is the machine.

3. Develop a conflict set for machine M_r consisting of all operations that will require machine r before operation O_{jr} will be completed.
   $ES(O_{kr}) < EC(O_{jr})$.

4. Select an operation out of the conflict set, and schedule it on M_r.

Remember, all optimal schedules are active schedules. Therefore, it would seem beneficial to search only within the set of active schedules. However, there are two problems with this idea. One, it is computationally expensive to look ahead in the schedule building process, and two, the set of active schedules is fairly large. Most of the schedules in this active space, are very "non optimal" schedules, but active nonetheless. Recall from Chapter 2 the set of non-delay schedules, which are a subset of active schedules. These are schedules in which no machine is held open for any period of time when it *could* be processing a job. For a more detailed explanation and examples, refer to Chapter 2. Basically, non-delay schedule building corresponds to *not* looking ahead in time, and not considering an operation for a machine which will become available in the near future. One of the many things learned through the course of this research is that not only searching non-delay schedules much faster computational wise, but many times the optimal solutions can be found! Moreover, the optimal solutions that do not lie in the set of non-delay schedules often lie just outside that space in the set of a 10 unit delay schedule, or a 20 time until delay schedule, or some relatively small time unit delay. Which means, instead of not looking ahead in time at all (non-delay), the scheduling algorithm will consider operations that looks ahead in time only a specified parameter of time units. As explained in Chapter 2, this set is called parameterized active schedules. Of course, a 10 unit time delay doesn't have any context to judge its significance by unless you know some information about the JSP at hand, such as the average time for an operation. It's unfortunate, but one parameter of time considered small for one JSP might be really large for another. This makes finding a robust parameterized active Job Shop Scheduling algorithm even more difficult.

It should be stated for completeness just exactly how the GT algorithm looks ahead in time, and how this can be adjusted for either a non-delay schedule or some parameterized active schedule. This was briefly discussed in Chapter 2, and is explained again here for convenience and context. The ability to build either a non-delay schedule, an active schedule, or a parameterized active schedule lies in the 3rd step of the GT Algorithm. The first step in the GT Algorithm is to select the operation with the earliest completion time from all unscheduled operations, operation O_{jr}. Next, a conflict set for the machine that is called for by O_{jr} is constructed, machine r. This conflict set is filled with operations that will need that same machine, r, before O_{jr} is completed, hence a conflict set. This is where the GT Algorithm looks ahead in time, specifically τ_{jr} units ahead, which is the processing time of O_{jr}.

In order to create a parameterized active schedule, the algorithm should only look ahead a certain parameter of time instead of all of τ_{jr}. To build a non-delay schedule, make this parameter zero. Then conflict sets will only consist of operations that are already available for processing by machine r. Since τ_{jr} will be different for every operation considered the impact of a static value like 5, 10 or 20 will be varied. Remember that the GT algorithm only schedules one operation at a time, so for a JSP with 100 total operations, then 100 conflict sets must be generated, even if these conflict sets contain only one operation. It is safe to assume that many optimal schedules could be built by using non-delay schedule building ($\tau_{jr} = 0$) *except* for a few times in which a delay parameter should be used (τ_{jr} = Parameter). So another downside of specifying a static delay parameter throughout the entire schedule building process is the increased search

space that is only necessary for a few operations! It might be easy to understand that when the size of the parameter is increased, the size of the conflict set is usually increased, thereby increasing the references to the priority list. This not only increases computational time, but it requires a more "precise" priority list. It should be noted, if not already understood that since a priority list is used in conjunction with a scheduling algorithm the order of the priority list will not necessarily end up being the order of the operations of the resulting schedule, in fact most likely not. It is unclear what kind of impact this mismatch has on the optimization outcome, if any, but would be an interesting topic of research. The reason this mismatch could be an issue is that one different decision during the course of the schedule building procedure could change all the conflict sets from that point on, seemingly making it advantageous to have a "precise" priority list. However, the complications presented here, specifically relating to parameterized active schedule building is just beyond the scope of this research, it is just noted here to give the reader a better understanding of parameterized schedules and their relationship to priority lists. Since there is no way to look at a Job Shop Problem and determine whether the solution lies in the set of non-delay schedules or in some specific set of "x" delay schedules, the safest schedule building is pure active schedule building. However, as mentioned before this has significant draw backs. Through the course of the simulations the enormous computational advantage was apparent of searching parameterized active schedule space and non-delay schedule space. To help illustrate this idea, and to better convey the strengths and weaknesses of the JSP/PSO algorithm, the results of the test bench problems in the following section are presented according to the delay parameter that was used. The reader can then draw their own conclusions about

usefulness of limiting the schedule building to all active schedules, parameterized schedules or non-delay schedules, and also the possible danger of such a practice.

6.2 Simulation Results

All the information from my program has been explained in the previous sections, and the proposed JSP/PSO Algorithm was explained in the previous chapter, so the reader should have all the necessary information to understand how the following results were obtained. Before the results are presented, the questions at stake are summarized below. They are presented below in order of significance.

1. Search space division by each machine's priority list.
2. Application of the PSO to the JSP in general.
3. The significance of limiting schedule searching to non-delay or just "outside" the non-delay domain.
4. Cycling of the PSO constants, C_1, C_2 and w.

To test the proposed algorithm, two well known benchmark function suites were used, the MT suite [14], which consist of 3 problems, and the LA suite [20], which consists of 40 problems. Three sets of the results from the optimization of these problems are presented in this section according to the delay parameter used during the optimization process. Each set consists of 15 runs, or 15 trials, of the JSP/PSO on the 43 bench functions. It is not unusual to have trials consist of 50-100 runs, but 15 were used here because of the significant amount of time it took to run 1 trial of a large JSP, on the order of hours. Every trial either terminated by obtaining the optimal solution, obtaining a

stagnant non-optimal solution for more than 600 iterations, or reaching the maximum iteration value of 2,000.

The first set are the results presented from building strictly non-delay schedules ($\tau_{jr} = 0$) are presented and discussed. Then the results are presented and discussed when using a time delay parameter of 10 ($\tau_{jr} = 10$), followed by the results and discussion of pure active schedule building. Finally, these results are compared to results obtained from another recent work in the JSP realm, Liu, Zhong, and Jiao [13].

6.2.1 Time Delay Parameter of 0 (Non-Delay Schedules)

Table 6.1: Time Delay of 0 Results Table

Name	Dimension (*n x m*)	Makespan					Avg. Function Evals
		Optimal	Found out of 15 Trials	Best	Avg	StDv	
MT06	6x6	55	0	57	57	0	10,563
MT10	10x10	930	0	956	985.6	15.43	16,975
MT20	20x5	1165	0	1180	1192.1	10.69	16,100
LA01	10x5	666	15	666	666	0	441.33
LA02	10x5	655	0	668	676.2	8.46	15,175
LA03	10x5	597	0	606	621.47	7.67	15,580
LA04	10x5	590	0	611	612	2.07	11,921
LA05	10x5	593	15	593	593	0	30.667
LA06	15x5	926	15	926	926	0	298.67
LA07	15x5	890	15	890	890	0	1,493.3
LA08	15x5	863	15	863	863	0	997.33
LA09	15x5	951	15	951	951	0	448
LA10	15x5	958	15	958	958	0	169.33
LA11	20x5	1222	15	1222	1222	0	650.67
LA12	20x5	1039	15	1039	1039	0	729.33
LA13	20x5	1150	15	1150	1150	0	334.67
LA14	20x5	1292	15	1292	1292	0	60
LA15	20x5	1207	13	1207	1212.3	13.90	3,924
LA16	10x10	945	0	988	1003.2	8.28	12,929
LA17	10x10	784	0	792	807.27	16.89	12,508
LA18	10x10	848	0	860	873.13	11.82	13,293
LA19	10x10	842	0	875	877	3.46	12,068
LA20	10x10	902	0	938	940.67	0.90	12,184
LA21	15x10	1046	0	1082	1119.4	19.77	19,079
LA22	15x10	927	0	977	998.4	14.90	19,184
LA23	15x10	1032	5	1032	1048.1	14.62	13,681
LA24	15x10	935	0	975	1004.7	11.07	16,741
LA25	15x10	977	0	1013	1053.9	18.72	18,744
LA26	20x10	1218	0	1237	1270.6	17.83	22,309
LA27	20x10	1235	0	1290	1317.7	18.17	23,911
LA28	20x10	1216	0	1251	1299.6	16.86	23,311
LA29	20x10	1152	0	1247	1281.4	24.40	24,828
LA30	20x10	1355	1	1355	1394.6	25.80	23,439
LA31	30x10	1784	13	1784	1787.9	14.16	6,990.7
LA32	30x10	1850	7	1850	1865.5	19.73	11,747
LA33	30x10	1719	4	1719	1735.5	18.58	21,365
LA34	30x10	1721	0	1748	1773.5	20.61	23,897
LA35	30x10	1888	7	1888	1906.5	32.65	17,920
LA36	15x15	1268	0	1332	1362.3	20.23	17,451
LA37	15x15	1397	0	1468	1490	14.52	18,320
LA38	15x15	1196	0	1280	1311.3	19.83	16,347
LA39	15x15	1233	0	1267	1320.2	18.45	14,864
LA40	15x15	1222	0	1286	1302.3	13.81	17,451

6.2.1.1 Non-Delay Analysis The results from the JSP/PSO using non-delay schedule building are fairly polarized. With the exception of a few problems, the optimal solution was either found exactly all 15 times, or found none of the times. One might attribute this to the fact that the solutions to the problems that were never found might not lie in the space of non-delay schedules, or perhaps the size of the problems were too large for the given amount of iterations allowed. The solutions to many of the problems with only 5 machines were found optimally, even if the number of jobs in the problem were 15 or 20. This might lead one conclude that real complexity of the problem increases more with the number of machines in the problem instead of the number of jobs, and I believe there is some truth to this. However, as will be shown in the next sections, the real difficulty of finding an optimal solution to a JSP has more to do with where in schedule space the solution lies. It gets much harder and takes longer to find a solution that lives outside of non-delay space and into parameterized active space, or even past parameter active space into total active schedule space.

6.2.2 Time Delay Parameter of 10

Table 6.2: Time Delay of 10 Results Table

Name	Dimension (*n x m*)	Makespan					Avg. Function Evals
		Optimal	Found out of 15 Trials	Best	Avg	StDv	
MT06	6x6	55	2	55	57.533	1.50	12,235
MT10	10x10	930	0	953	979.47	17.69	16,140
MT20	20x5	1165	0	1178	1189.3	9.34	19,007
LA01	10x5	666	15	666	666	0	554.67
LA02	10x5	655	0	665	674.67	10.39	15,139
LA03	10x5	597	2	597	613.33	10.68	14,045
LA04	10x5	590	0	611	611.67	1.75	12,589
LA05	10x5	593	15	593	593	0	85.333
LA06	15x5	926	15	926	926	0	377.33
LA07	15x5	890	14	890	890.6	2.32	3,197.3
LA08	15x5	863	15	863	863	0	1,909.3
LA09	15x5	951	15	951	951	0	968
LA10	15x5	958	15	958	958	0	381.33
LA11	20x5	1222	15	1222	1222	0	905.33
LA12	20x5	1039	15	1039	1039	0	1,144
LA13	20x5	1150	15	1150	1150	0	1,033.3
LA14	20x5	1292	15	1292	1292	0	228
LA15	20x5	1207	13	1207	1209.4	7.57	6,678.7
LA16	10x10	945	0	975	986.73	8.86	16,488
LA17	10x10	784	0	792	806.47	13.47	14,911
LA18	10x10	848	3	848	869.4	16.21	13,876
LA19	10x10	842	0	856	867.2	9.42	15,147
LA20	10x10	902	0	907	920.07	11.72	13,804
LA21	15x10	1046	0	1077	1114.8	15.38	20,628
LA22	15x10	927	0	948	979.8	17.06	21,964
LA23	15x10	1032	1	1032	1042.8	11.23	18,037
LA24	15x10	935	0	982	1000.9	15.42	19,951
LA25	15x10	977	0	1013	1041.1	16.76	19,039
LA26	20x10	1218	0	1243	1274.7	24.28	26,845
LA27	20x10	1235	0	1294	1334.5	25.30	26,099
LA28	20x10	1216	0	1279	1318.3	23.24	23,505
LA29	20x10	1152	0	1231	1279.3	32.09	25,819
LA30	20x10	1355	0	1373	1399.7	24.72	21,691
LA31	30x10	1784	2	1784	1796.4	12.84	20,148
LA32	30x10	1850	1	1850	1894.3	27.76	21,464
LA33	30x10	1719	0	1726	1763.6	22.23	20,176
LA34	30x10	1721	0	1747	1793.9	23.47	24,088
LA35	30x10	1888	1	1888	1919.4	28.51	23,853
LA36	15x15	1268	0	1335	1364.5	18.91	20,265
LA37	15x15	1397	0	1478	1493.1	10.27	19,469
LA38	15x15	1196	0	1259	1293.4	27.35	20,736
LA39	15x15	1233	0	1264	1298.1	23.23	17,416
LA40	15x15	1222	0	1269	1307.9	22.96	18,195

6.2.2.1 Parameterized Active Schedule Analysis Now the space of parameterized active schedules is considered. As stated before, the parameter used for this set of problems was 10. Most of the problems in the two test suites don't have single operation times that exceed 80, or fall below 20. Therefore, a parameter of 10 seemed like a good guess to search just outside the set of non-delay schedules, as was the goal. If most of the test problems had average operating times of say 20, then a parameter of 1-3 would have been more appropriate in defining this set. So, for this set of trials the GT Scheduling Algorithm considered operations for machines that aren't available at that exact moment in time, but *almost* available. Two aspects of the results are of interest to us here. One is the fact that the optimal schedule was found twice for the MT06 scheduling algorithm, where it was never found before when searching in the non-delay set. The same observation can be made about the LA03 and the LA18 problems. I would say this is evidence that those solutions don't live in non-delay schedule space, but just outside of it. However, the other aspect to note is the decreased number of times the optimal solution was found in many of the other test problems, like LA07, LA15, LA23, LA30, LA31, LA32, LA33, and LA35. Obviously, this is because the search space was increased too much for the JSP/PSO Algorithm to still find the optimal solution in the same amount of iterations. It is safe to assume that this trend will continue with a larger impact into the much larger space of active schedules. The following table contains the results from active schedule searching.

6.2.3 All Active Schedules

Table 6.3: All Active Schedules Results Table

Name	Dimension (*n x m*)	Makespan					Avg. Function Evals
		Optimal	Found out of 15 Trials	Best	Avg	StDv	
MT06	6x6	55	2	55	57.733	1.83	14,995
MT10	10x10	930	0	997	1040	21.71	23,143
MT20	20x5	1165	0	1224	1285.6	29.91	22,888
LA01	10x5	666	1	666	688.93	16.60	17,481
LA02	10x5	655	0	672	712.73	26.81	17,608
LA03	10x5	597	0	621	638.47	13.64	17,581
LA04	10x5	590	0	610	628.6	13.96	16,341
LA05	10x5	593	13	593	593.6	1.68	3,989.3
LA06	15x5	926	11	926	928.27	6.41	12,167
LA07	15x5	890	7	890	900.53	14.10	16,688
LA08	15x5	863	0	866	888.93	18.14	20,395
LA09	15x5	951	1	951	968.33	9.89	22,451
LA10	15x5	958	12	958	959.67	4.36	8,474.7
LA11	20x5	1222	7	1222	1230.9	10.90	18,987
LA12	20x5	1039	1	1039	1055.1	14.12	22,357
LA13	20x5	1150	4	1150	1170.9	17.34	23,376
LA14	20x5	1292	15	1292	1292	0	2,196
LA15	20x5	1207	0	1212	1265.3	27.87	26,484
LA16	10x10	945	0	982	1027.5	18.92	17,015
LA17	10x10	784	0	793	837.53	20.63	17,323
LA18	10x10	848	0	896	938.33	20.98	18,909
LA19	10x10	842	0	887	922.13	20.45	20,417
LA20	10x10	902	0	951	990.13	17.48	16,945
LA21	15x10	1046	0	1187	1244.1	39.65	24,400
LA22	15x10	927	0	1099	1135.5	30.61	22,785
LA23	15x10	1032	0	1142	1191.6	28.33	24,320
LA24	15x10	935	0	1100	1131.1	25.26	26,144
LA25	15x10	977	0	1112	1171.3	40.56	20,903
LA26	20x10	1218	0	1448	1495.1	21.85	25,519
LA27	20x10	1235	0	1446	1532.7	40.08	24,115
LA28	20x10	1216	0	1485	1512.5	21.91	25,561
LA29	20x10	1152	0	1383	1444.3	31.00	23,961
LA30	20x10	1355	0	1540	1582.3	23.35	22,572
LA31	30x10	1784	0	1966	2008.8	24.51	24,109
LA32	30x10	1850	0	2084	2137.5	28.63	25,240
LA33	30x10	1719	0	1900	1967.5	40.60	22,471
LA34	30x10	1721	0	1999	2019.6	17.18	20,843
LA35	30x10	1888	0	2068	2106.7	35.74	18,723
LA36	15x15	1268	0	1464	1517.3	38.21	25,645
LA37	15x15	1397	0	1585	1638.4	31.19	23,501
LA38	15x15	1196	0	1398	1454.8	23.59	23,409
LA39	15x15	1233	0	1447	1488.9	32.46	25,089
LA40	15x15	1222	0	1404	1444.9	22.08	27,939

6.2.3.1 Active Schedule Analysis The results from active schedule building are not too promising. Going from using a time delay parameter of 10, to the space of all active schedules, reduced the number of times optimal solutions were found for all test problems, with the one exception of the LA14 problem, and the MT06 problem where the solution found only twice. The time it took to run 15 trials of each problem increased significantly, because of the increased size of the conflict set for each operation scheduled, which thereby increases the computational cost. This increased time is on the order of tens of hours.

6.2.4 Gantt Chart and Priority List Analysis In reality, we are interested only in the makespan of a particular solution to the JSP, which is just a number. However, it is interesting and helpful at times to look at the schedule in the form of a GANTT chart. It is possible to "kind of" judge the complexity of a problem by looking at the complexity of its solution. In this section some priority lists and corresponding Gantt charts of some of the solutions the JSP/PSO produced to give the reader some visual feedback.

The following figure is a Gantt chart of the LA01 Problem. This *(10 x 5)* problem was solved optimally by non-delay scheduling. As can be seen by looking at the Gantt chart, the problem seems fairly easy to solve as compared to some other Gantt charts which will be shown later. Notice how most of the time all machines are processing an operation, this is a result of how the problem happens to be defined, the precedent constraints and processing times happen to allow for very efficient use of time. Presented in Table 6.4 along with the Gant chart in Figure 6.10 is the actual LA01 problem definition. Also

presented in Table 6.5 is the optimal priority list that was optimized by the PSO, which obviously indirectly represents an optimal schedule. As mentioned before the optimal priority list does not really come close to matching the order operations on the Gantt chart.

Table 6.4: LA01 Job Shop Problem

	Machine Sequence (Time)
Job 1	2 (21) 1 (53) 5 (95) 4 (55) 3 (34)
Job 2	1 (21) 4 (52) 5 (16) 3 (26) 2 (71)
Job 3	4 (39) 5 (98) 2 (42) 3 (31) 1 (12)
Job 4	2 (77) 1 (55) 4 (79) 2 (66) 3 (77)
Job 5	1 (83) 4 (34) 3 (64) 2 (19) 5 (37)
Job 6	2 (54) 3 (43) 5 (79) 1 (92) 3 (62)
Job 7	4 (69) 5 (77) 2 (87) 3 (87) 1 (93)
Job 8	3 (38) 1 (60) 2 (41) 4 (24) 5 (83)
Job 9	4 (17) 2 (49) 5 (25) 1 (44) 3 (98)
Job 10	5 (77) 4 (79) 3 (43) 2 (75) 1 (96)

Table 6.5: LA01 Optimal Priority List

Machine 1:	2	1	9	5	10	6	4	3	8	7
Machine 2:	1	6	3	4	2	7	9	5	10	8
Machine 3:	10	2	5	4	1	8	6	7	9	3
Machine 4:	7	3	1	2	5	8	9	6	4	10
Machine 5:	9	6	4	7	3	2	1	5	8	10

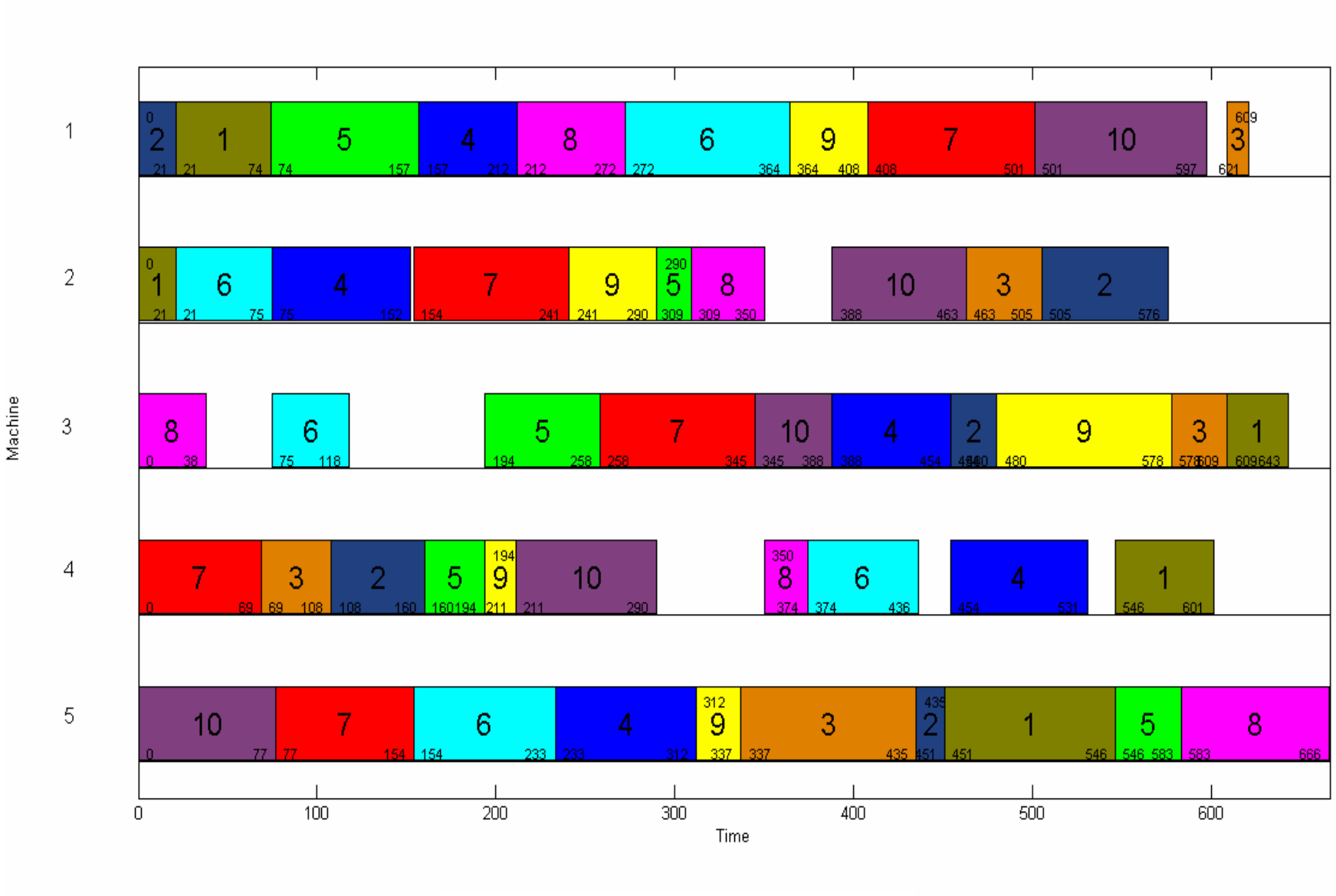


Figure 6.10: Gantt chart of LA01 Solution

One of the greatest examples of the complexity of the Job Shop Problem is how drastic a difference between two schedules can be with approximately the same makespan. There is nothing about the JSP would lend one to expect schedules that are close in makespan are also "close" in combinatorial space. Which means that to go from a makespan of say 668 to 666 may require significant changes in the order of the operations of all the machines. Needless to say, this makes optimization very difficult. If this is not more evidence to support the cycling of the PSO constants that was explained earlier, then it is definitely evidence to support the idea that particles should not necessarily be programmed to all converge to the best found solution toward the end of optimization. Figure 6.11 is a Gantt chart of a solution obtained by the JSP/PSO to the LA01 problem of the previous example, however, it is not the optimal solution. The solution has a makespan of 668, two time units longer than the optimal. Notice the overall drastic difference in the two schedules, and contemplate how difficult it would be for particles searching from the near schedule below to find the optimal schedule of Figure 6.10. This could be why the meta-heuristic optimization method of Simulated Annealing has had success in solving the Job Shop Problem, which is an optimization construct that accepts "up hill" moves with a certain probability. The priority list for this non-optimal schedule is also shown below for comparison to the optimal priority list of the previous figure.

Table 6.6: Priority List for Near Optimal LA01.

Machine 1:	9	2	8	6	7	10	4	5	1	3
Machine 2:	10	3	6	5	8	2	4	7	1	9
Machine 3:	4	3	6	10	5	7	9	1	8	2
Machine 4:	5	7	9	6	10	8	1	3	4	2
Machine 5:	2	6	9	7	1	4	3	5	8	10

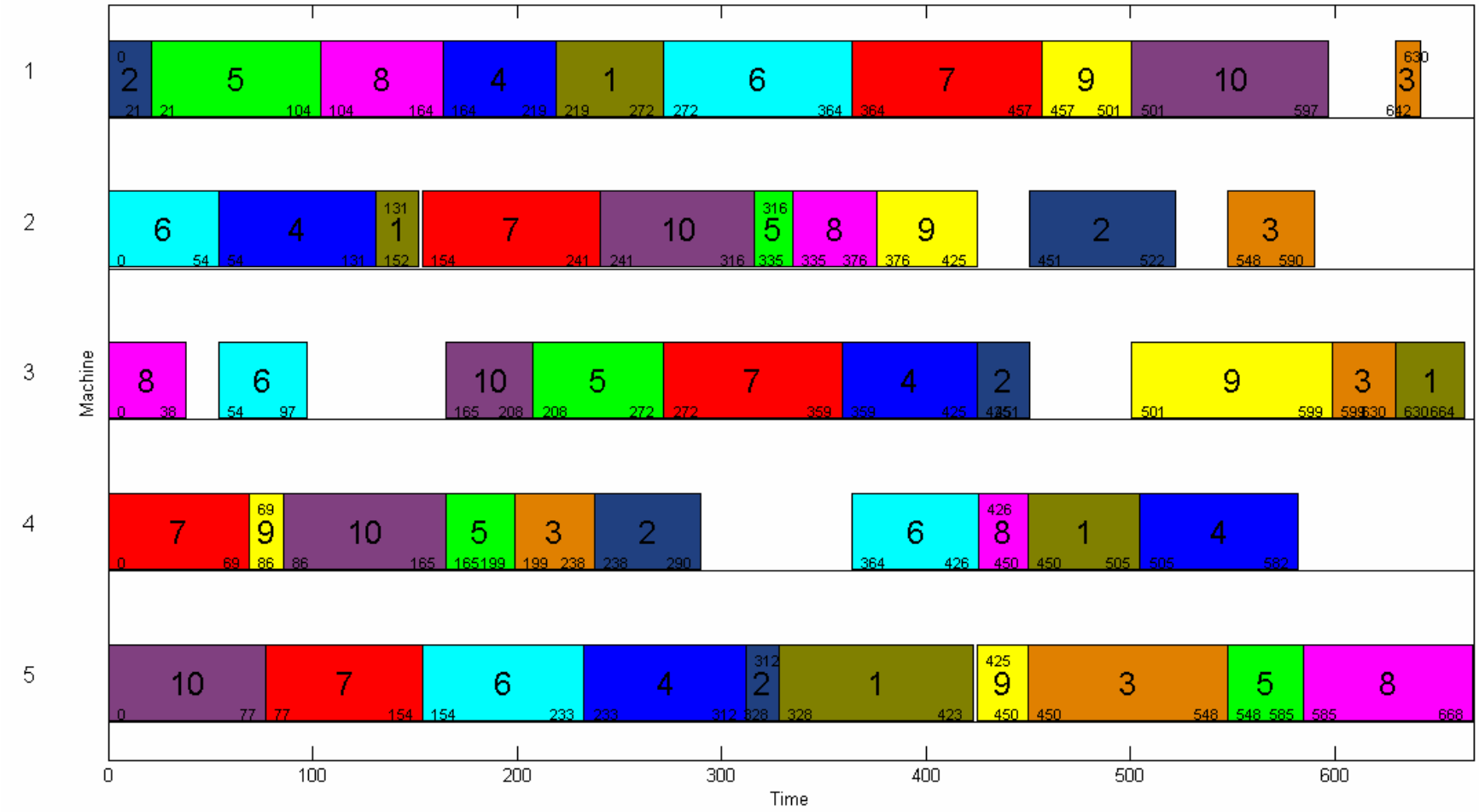


Figure 6.11: Gantt Chart for Near Optimal Solution of LA01

Another benefit of looking at the actual schedule of a solution is ability to see whether or not non-delay scheduling was needed. For example, the LA03 problem was one of the harder (10×5) problems to solve, however it was solved optimally a couple of times by using parameterized active scheduling, see Table 6.2, and it wasn't solved optimally at all using non-delay scheduling, see Table 6.1. By looking at the Gantt chart of the optimal schedule of LA03, we can see where this delay parameter was necessary. In Figure 6.12 below this delay is circled in red. Notice after Job 9 gets done processing on Machine 1 at time 286 that it does not start processing an operation until time 295, *even though* both Jobs 1, 8 and 10 are ready for processing by Machine 1. Instead it waits 9 time units for Job 5 to get done processing on Machine 5. Obviously, the delay parameter of 10 was just enough for finding of this optimal solution. Not only does the Gant chart in Figure 6.12 allow us to see where active scheduling is necessary, but it also illustrates an even better point. This delay of Machine 1 was the *only* time when it was necessary to not process waiting operations. So for 50 operations, only one needed parameterized active scheduling. So the schedule is largely non-delay. However, a constant delay parameter of 10 time units was used to schedule every operation! In other words, the search space was increased unnecessarily for every other operation. This fact is manifested in the results of the JSP/PSO Algorithm where the solution was only found 2 times out of 15 for parameterized active schedule searching, shown in Table 6.2, and 0 out of 15 times for the search all active schedules, shown in Table 6.3. I believe that this predicament could easily be the subject of more research.

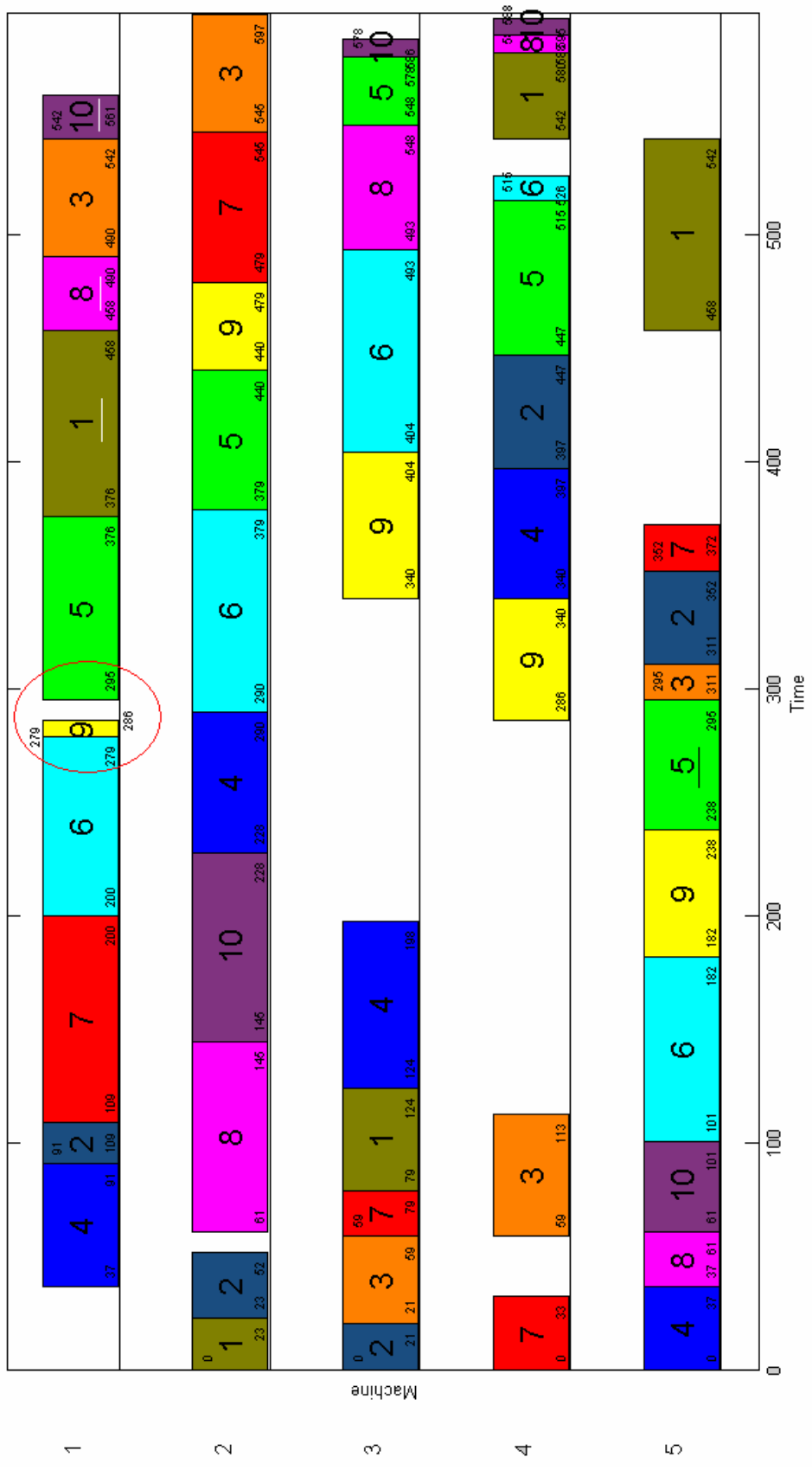


Figure 6.12: Gantt Chart for LA03 Solution

Lastly, to give the reader yet more context into the complexity of Job Shop Problems, two more Gantt charts are presented below. The first one, Figure 6.13, is a non-optimal schedule of the MT10 problem. This is a very hard (10×10) JSP that unfortunately was never solved by the JSP/PSO. It isn't hard to see why the problem is difficult to solve optimally. Notice how all the operations slant from the top left to the bottom right, this indicates how most of jobs all require the same machines at the same time through out the course of the schedule. This is seen more easily in the problems precedent constraints shown below.

Table 6.7: Precedent Constraints for MT10

	Machine Sequence									
Job1:	1	2	3	4	5	6	7	8	9	10
Job2:	1	3	5	10	4	2	7	6	8	9
Job3:	2	1	4	3	9	6	8	7	10	5
Job4:	2	3	1	5	7	9	8	4	10	6
Job5:	3	1	2	6	4	5	9	8	10	7
Job6:	3	2	6	4	9	10	1	7	5	8
Job7:	2	1	4	3	7	6	10	9	8	5
Job8:	3	1	2	6	5	7	9	10	8	4
Job9:	1	2	4	6	3	10	7	8	5	9
Job10:	2	1	3	7	9	10	6	4	5	8

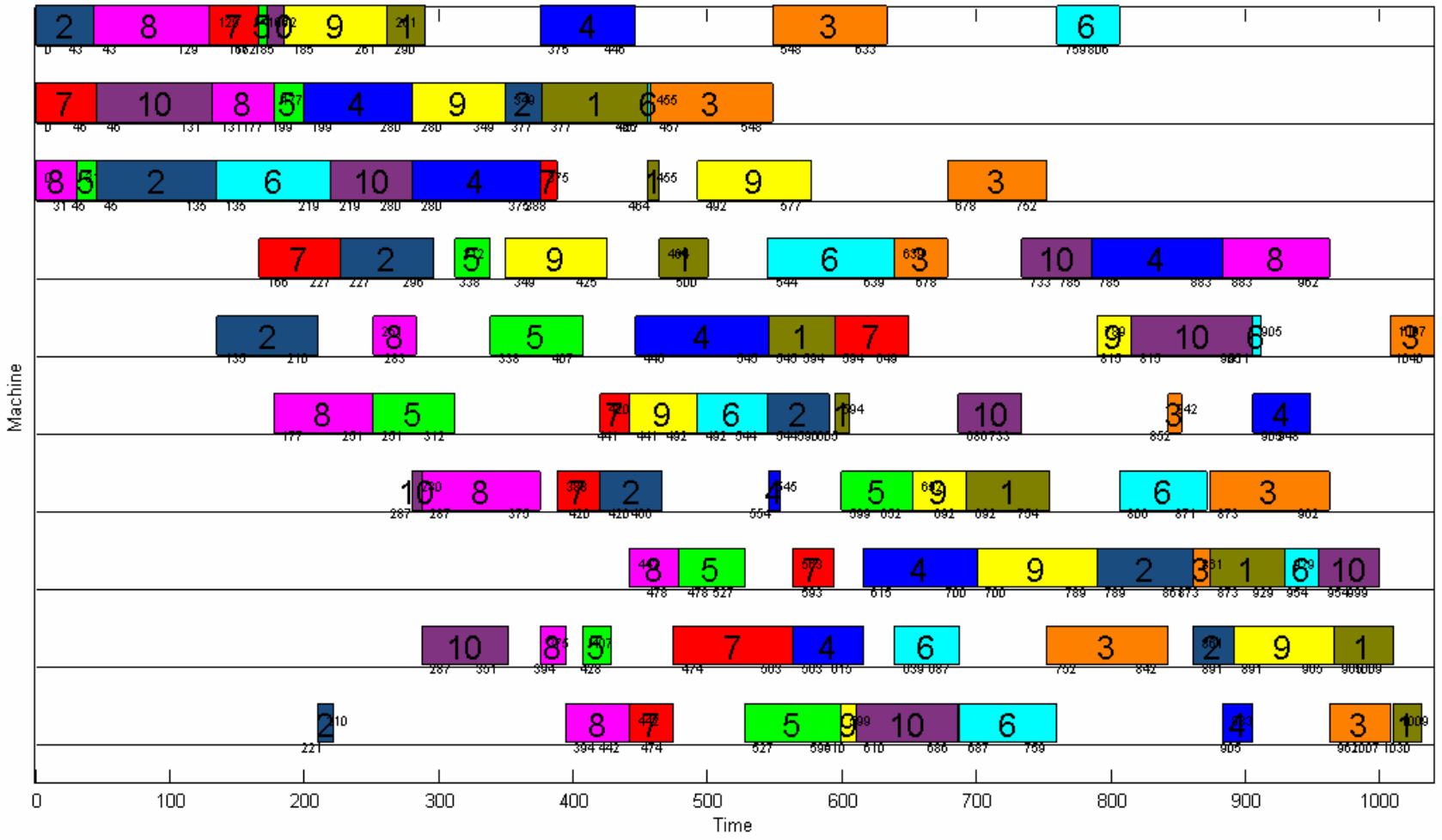


Figure 6.13: Gantt Chart for Non-optimal MT10 Schedule

Also, notice how in contrast to some of the previously presented schedules, there is a lot of idol time for all of the machines. The operations don't flow consecutively on any of the machines without down time. Simply put, this is just a very tough problem to solve because of the way it's defined. It might be tempting to think that the larger the problem is the harder it is to solve it. While there is some truth to this, it is not entirely true. It is obvious that the JSP/PSO had trouble solving the MT10 problem, so it might be tempting to think that it would have an even more difficult time solving a larger JSP. The Gantt chart in Figure 6.14 is the optimal schedule of the LA31 problem, a (30×10) JSP. This solution is a non-delay solution and was found 13 times out of 15 by non-delay schedule searching and 2 times out of 15 by parameterized schedule searching. Looking at Figure 6.14 it is easy to see the shear combinatorial oblivion that the solution lies in and it is also easy to see that unlike the MT10 solution there is hardly any idol time for any of the machines. Perhaps, this fact, more than problem size is an indication of how hard a particular JPS is to solve.

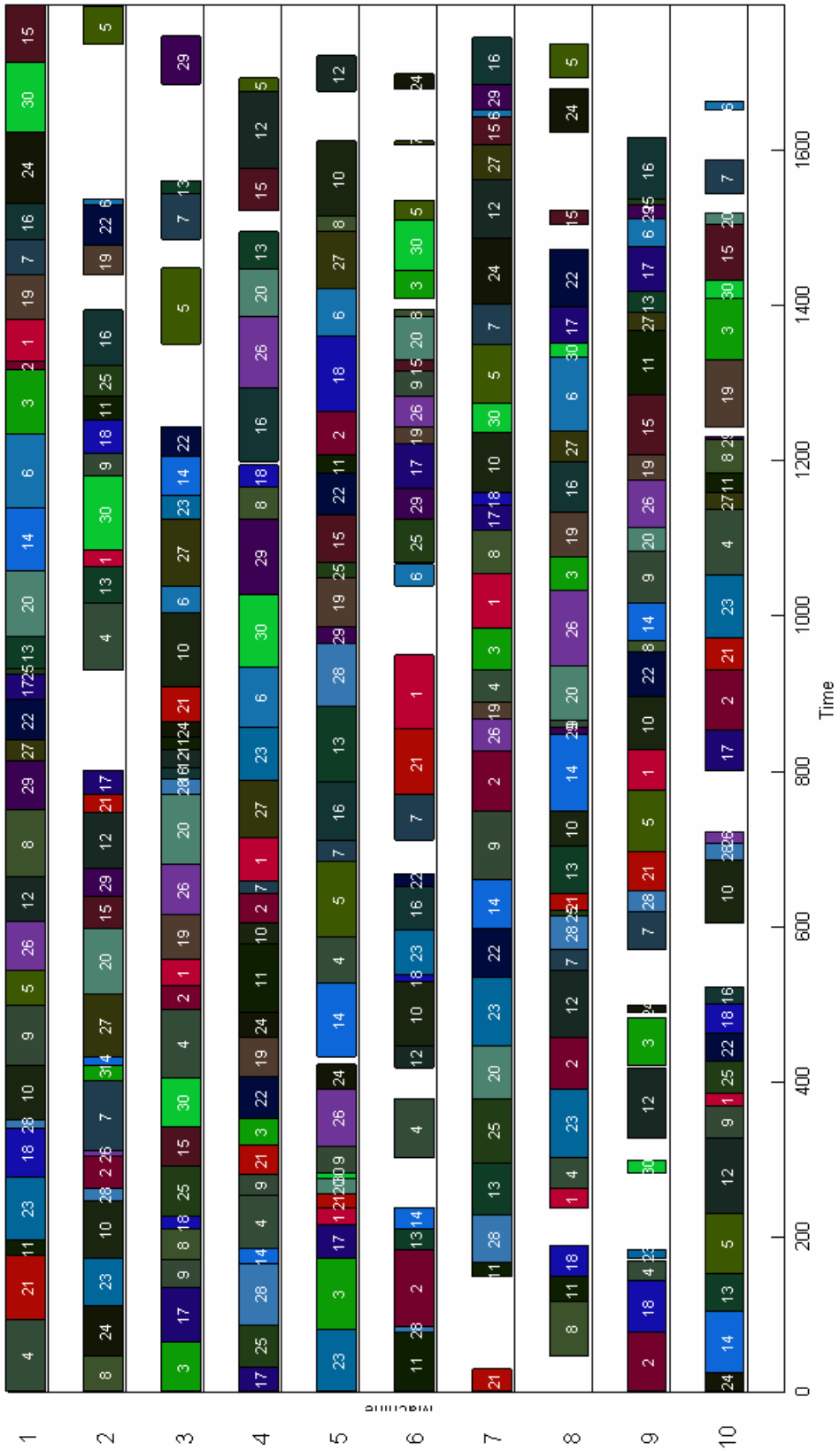


Figure 6.14: Gantt Chart of LA31 Solution

6.3 Comparative Analysis Now that the performance of the JSP/PSO has been presented and analyzed, the significance of this algorithm will be discussed by comparing its results to results of another recent JSP algorithm. As of this writing the most recent and complete publication of the results from many of the same JSP test functions came from Liu, Zhong, and Jiao [13], published in February of 2006. They used a multi-agent system, which is another emerging meta-heuristic optimization method, to solve the MT, LA and ORB test bench suites. The results they obtained are quite good. They have been taken from [13] and presented in the following table for comparison purposes.

Table 6.8: Comparative Results

Name	Dimension (*n x m*)	Makespan				Function Evals
		Optimal	Best	Avg	StDv	
MT06	6x6	55	55	55	0	450
MT10	10x10	930	930	944.45	7.81	2,270,375
MT20	20x5	1165	1165	1178.89	4.80	3,106,852
LA01	10x5	666	666	666	0	1,631
LA02	10x5	655	655	655.39	1.95	296,570
LA03	10x5	597	597	598.86	2.79	409,782
LA04	10x5	590	590	591.41	1.50	635,362
LA05	10x5	593	593	593	0	262
LA06	15x5	926	926	926	0	349
LA07	15x5	890	890	890	0	1,540
LA08	15x5	863	863	863	0	2,519
LA09	15x5	951	951	951	0	802
LA10	15x5	958	958	958	0	316
LA11	20x5	1222	1222	1222	0	495
LA12	20x5	1039	1039	1039	0	865
LA13	20x5	1150	1150	1150	0	1,061
LA14	20x5	1292	1292	1292	0	321
LA15	20x5	1207	1207	1207	0	5,578
LA16	10x10	945	945	945.79	.41	1,408,499
LA17	10x10	784	784	784	0	225,340
LA18	10x10	848	848	848.22	.97	719,109
LA19	10x10	842	842	853.79	5.16	2,174,830
LA20	10x10	902	902	908.11	1.92	2,042,142
LA21	15x10	1046	1046	1068.11	11.09	4,153,326
LA22	15x10	927	927	940.88	5.27	4,110,514
LA23	15x10	1032	1032	1032	0	68,246
LA24	15x10	935	937	958.79	11.66	4,188,613
LA25	15x10	977	977	993.50	8.39	3,738,814
LA26	20x10	1218	1218	1219.15	3.54	1,939,943
LA27	20x10	1235	1236	1263.83	8.38	6,812,466
LA28	20x10	1216	1216	1225.55	7.13	6,351,364
LA29	20x10	1152	1167	1201.88	15.60	5,972,694
LA30	20x10	1355	1355	1355	0	727,852
LA31	30x10	1784	1784	1784	0	37,799
LA32	30x10	1850	1850	1850	0	79,068
LA33	30x10	1719	1719	1719	0	26,238
LA34	30x10	1721	1721	1721	0	228,581
LA35	30x10	1888	1888	1888	0	105,840
LA36	15x15	1268	1274	1295.49	8.30	4,992,731
LA37	15x15	1397	1397	1429.24	14.32	5,517,937
LA38	15x15	1196	1204	1242.42	16.36	5,956,429
LA39	15x15	1233	1239	1258.61	11.85	6,011,274
LA40	15x15	1222	1224	1247.06	11.07	5,770,028

The most amazing fact about these results is that they were obtained using permutation with repetition, a semi-active schedule building representation! This means their search space was larger than the search space used by the JSP/PSO, but obtained much better results. This is probably the result of a couple of differences between the two programs. The main difference being the fact building schedules with permutation with repetition is much faster than using a priority list and GT Algorithm. Therefore, Liu, Zhong, and Jiao were able to perform many more iterations and function evaluations than I was using the JSP/PSO. This can be seen when comparing the number of average function evaluations in the above table with the number of average function evaluations from the JSP/PSO tables, particularly the table representing the non-delay set. For example, the average number of function evaluations it took for the JSP/PSO to solve the LA01 problem optimally 15 times is 441.33, where it took [13] an average of 1,631 function evaluations. A more drastic difference can be seen in the MT20 problem. The JSP/PSO never solved this problem optimally in any of the three sets, however, in the non-delay set its *Average Makespan* value came close to matching the same *Average Makespan* value of [13], but its average function evaluations was only 19,007 compared to 3,106,852 of [13]. This is a common occurrence through all three sets of results, that is, the fact that [13] had to perform several million function evaluations to arrive at the optimal solution. Unfortunately, it is very computationally expensive to perform a single function evaluation using the GT Algorithm and a priority list. So it would take too long to perform a million of them. This leads one to wonder how the JSP/PSO could fair if it was faster and was allowed to run a ½ million function evaluations to optimize a problem. I believe that if the JSP/PSO were a little faster then the results could have been

better. However, all is not lost here, because solutions were found and that indicates that there the space division concept worked. It actually works well for small Job Shop Problems whose solution lies in the set of non-delay schedules.

CHAPTER SEVEN

CONCLUSION

The proposed JSP/PSO Algorithm is capable of solving Job Shop Problems. This is significant because of the way this optimization progressed, through the division of the search space by the machines in the problem. As of this writing, I am not aware of any other researcher taking this approach in the Job Shop Realm. This space division technique shows that independent swarms or populations working toward a common goal may not have to have knowledge of such a "group effort". This may have been already known in the meta-heuristic optimization community, but to my knowledge never applied to the Job Shop Problem. Also, what I believe to be beneficial to the research community is my research on the significance of searching parameterized active delay schedules, and perhaps the complications that go along with indirect scheduling. Of course this search space division has application to huge combinatorial problems such as the JSP, however, what I consider a more important conclusion is the possibility of greater success of an algorithm developed that uses the same search space division by machines, but *does* facilitate information sharing between separate swarms or populations. It seems to me that if the proposed JSP/PSO Algorithm can work to solve small to medium Job Shop Problems, then a better algorithm that specifies a way of sharing information between swarms could do much better. I would think that a Neural Network would have great application here. I believe a neural network might be able to be trained to select specific

particles from each of the swarms that should come together to form a schedule, or priority list, instead of the static linking that occurs in the JSP/PSO. In the following sections the specific conclusions of lessons learned and things to do differently are discussed, as this just important as things that worked well.

7.1 Lessons Learned

7.1.1 Significance of Delay Parameter Probably the most unexpected discovery during my research was the criticalness of the delay parameter used by the GT Scheduling Algorithm. Recall that this parameter effectively limited the amount of active schedules in the search space. Early in my research, I thought that most optimal schedules would lie far away from non-delay schedules, but I have concluded that I was wrong. It turns out that many optimal schedules are non-delay schedules and those that aren't non-delay are "close" to it. It also turns out that the optimal non-delay schedules are the easiest to find, as seen the results of my simulations. This delay parameter is critical and it isn't easy to determine before hand. When my searches were limited to a parameterized active schedules not only did my program run faster, I also achieved better results. This is one of many issues I have with the use of active schedule building. To build semi-active schedules, this delay parameter isn't an issue. And even though the search space is increased significantly the solutions that result from strictly semi-active schedules are usually so bad that the search quickly narrows to exclude them. I believe an interesting area of research would be to develop a dynamic delay parameter that could easily adapt itself to all different sizes of Job Shop Problems by extracting problem specific information like average processing time for an operation.

7.1.2 GT Algorithm in General Another critical discovery I made through the course of my research and simulation was the enormous computational time of the GT Algorithm. As mentioned before the GT Algorithm looks ahead in time and uses priority lists to make decisions, this required many programming loops. For problems of (10×10) or bigger this was a big problem. Simulations took a large amounts of time, on the order of tens of minutes. However, as stated previously, using the JSP/PSO normally didn't require the large number of function evaluations as other methods did such as permutation with repetition. The JSP/PSO required less function evaluations, but they took longer to perform. Thankfully, this complies with the "no free lunch" theorem. Overall, I now believe that using the GT Algorithm to search only active schedules is not worth the computational expense. At the beginning of my research using the algorithm seemed like an obvious choice, but then I discovered the ease of searching simply in semi-active schedule space, like permutation with repetition. This ease stems from the actual simple code of such an algorithm and the incredible speed of running it compared to the GT algorithm. There seem to be many success stories of algorithms that search semi-active space, such as the multi-agent system in [13].

7.1.3 Priority List Representation Related to the drawback of the GT Algorithm is the drawback of priority lists in general. I believe that attempting to optimize something such as a priority list which indirectly affects the objective function is probably not the best way to solve a JSP, or at least not the best way to use a meta-heuristic technique. Just like how many different "permutation with repetition" instances will result in the same schedule, many different priority lists will also result in the same schedule, making

hard for meta-heuristic optimization. Both permutation with repetition and priority lists are examples of indirect scheduling, however at least permutation with repetition schedules are much faster to build. Since many computational intelligence optimization techniques rely on "survival of the fittest" principles, like selection pressure, having changes in the genotype (the particle in this case) not directly affect the phenotype (the schedule) does not work as well as it possibly could. In the case of the JSP there is a whole other algorithm (the scheduling algorithm) that stands in the way of the particles "output" and the resulting objective function. In many cases, two operations on a certain machine in a resulting schedule will be in the opposite order how they appear on the priority list. Some researchers have tried to fix this problem by using a technique known as "forcing". Fenton, P. and Walsh P. briefly review this concept, its purpose and effectiveness in [6]. Basically, forcing is the manipulation of the genotype to more accurately match the phenotype, or in this case the direct manipulation of the particles to more accurately match the schedule that they produced. However, this technique is not considered to help all that much in the JSP realm. It seems likely to me that using direct scheduling, and thereby having to deal with infeasible solutions might be a better trade off than using indirect scheduling and having a genotype/phenotype mismatch. For future research in this area, this would be my number one change. Using direct scheduling would also eliminate the use of the GT Algorithm, which would save on computation time. Also, it is very possible that a significantly less computationally expensive scheduling algorithm other than the GT Algorithm could be developed to be used in conjunction with the standard priority list. This would be an easy modification that might produce better results, but certainly improve computational expense.

7.1.4 Cycling PSO Constants The cycling of constants used during the Particle Swarm Optimization process worked well, seemingly better than having them not cycled, and definitely better than keeping them static. Although, there were no results presented as evidence of such a claim, it seems that I should disclose my practice of doing so and whether or not I believe it helped. One reason I believe this concept worked better because of the huge combinatorial space that is being searched by the particles, and regular non-linearity of it. Using the cycling C_1, C_1 and W constants allowed me to have more control over my program, especially if I wanted to terminate the program early, or keep it running more iterations. I don't consider this a big finding, but work mentioning for possible future study the PSO realm.

7.2 Final Thoughts

While no earth shattering discoveries were made, the results of my research do show promise and possibility for future work. The proposed JSP/PSO Algorithm did solve Job Shop Problems, and in some instances very quickly. To wrap up and be clear my specific conclusions are presented in the bulleted list below.

- The Proposed Algorithm works fairly well and shows promise.
- The GT Algorithm is computationally too expensive to justify its use.
- It is computationally worth searching the set of non-delay and parameterized active schedules.
- Cycling PSO constants works (granted, there was no proof of this, but feel like I should state it nonetheless).

I have found, probably like many students and professors, hard to wrap up my research and conclude with the results that I have. This is mainly due to the learning and better grasp of the subject matter that seems to happen constantly while running simulations and doing research. My situation is no exception to this. It is obvious that no incredible discoveries were made, but many discoveries of what "doesn't work so well" were made. Through the course of my research I grew to have a much deeper understanding of and respect for the Job Shop Problem. At first thought, it doesn't seem like much more than a big combinatorial problem, but after probing deeper, it is easy to get lost in it, especially the different classifications of schedules and the different ways to search them. What's interesting about this problem is its history and stubbornness to remain difficult to solve even with modern computing. I believe that there is a future area of research for the JSP and the PSO Algorithm, this is only the beginning of the combination of the two. No doubt the PSO Algorithm is strong and could be powerful against the JSP, and no doubt that there might be better ways to conduct the space transformation needed to use PSO in the JSP realm. Even more promising is the possible connection between the different swarms for the different machines in the JSP/PSO. It seems natural to think that if the optimization can work without some overall coordination, then it could work better with it.

It looks like the production line managers and routing coordinators will have to wait a little longer for a miraculous program that will dictate optimal schedules to them without trouble, the JSP/PSO is good, but it's not that good *yet*. It's just a particle in the swarm of meta-heuristic literature. Just like each particle in the JSP/PSO algorithm works

independently of the others and represents only part of the solution, but comes together with others to construct a solution, individuals in the research arena each work independently to complete the puzzle at hand one piece at a time. Hopefully, this work is a piece of that puzzle, one particle among a swarm of ideas.

REFERENCES

[1] Bierwirth, C. A Generalized Permutation Approach to Job Shop Scheduling with Genetic Algorithms. OR Spektrum, **17**:87-92. 1995.

[2] Chang, Y. L, Sueyoshi, T. and Sullivan, R. S. Ranking Dispatching Rules by Data Envelopment Analysis in a Job-Shop Environment, *IIE Transactions*, **28**(8): 631-642. 1996.

[3] Davis, L., Job Shop Scheduling with Genetic Algorithms, *Proceedings of the 1st International Conference on Genetic Algorithms*, Pittsburgh, PA, 136-140. 1985.

[4] Dimopoulos, C., Zalzala, A., Recent Developments in Evolutionary Computation for Manufacturing Optimization: Problems, Solutions, and Comparisons, *Transactions on Evolutionary Computation*, **4**(2): 93-113. 2000.

[5] Eberhard, R.C., and Kennedy, J., A New Optimizer Using Particle Swarm Theory, *Proceedings. of the Sixth International Symposium on Micro Machine and Human Science*, Nagoya, Japan, 39-43. 1995.

[6] Fenton, P., Walsh, P., Improving the Performance of the Repeating Permutation Representation Using Morphogenic Computation and Generalized Modified Order Crossover, *The 2005 IEEE Congress on Evolutionary Computation*, **2**: 1372-1379. 2005.

[7] Giffler, B. and Thompson, G.L. Algorithms for Solving Production Scheduling Problems, *Operations Research*, **8**(4): 487-503. 1960.

[8] Goncalves, J. F, Mendes, J.J, and Resende, M.C.G., A Hybrid Algorithm for the Job Shop Scheduling Problem, AT&T Technical Labs Report TD-5EAL6J. 2002.

[9] Jackson, J. R., Scheduling a Production Line to Minimize Maximum Tardiness, Research Report 43, Management Science Research Projects, University of California, Los Angeles, USA. 1955.

[10] Jackson, J. R. Simulation Research on Job-Shop Production, *Naval Research Logistics Quarterly*, **4**: 287-295. 1957.

[11] Jain, A.S. and Meeran, S., A State-of-the-Art Review of Job-Shop Scheduling Techniques, European Journal of Operations Research, **113**: 390-434. 1999.

[12] Lian, Z., Gu, X., Jiao, B., A Similar Particle Swarm Optimization Algorithm for Permutation Flowshop Scheduling to Minimize Makespan, *Applied Mathematics and Computation*, 2005.

[13] Liu, J., Zhong, W., Jiao, L., A Multiagent Evolutionary Algorithm for Constraint Satisfaction Problems. *IEEE Transactions on Systems, Man, and Cybernetics—Part B: Cybernetics*, **36**: (1): 54-73.2006.

[14] Muth J.F., Thompson G.L., Industrial Scheduling, Englewood Cliffs, New Jersey, Prentice Hall. 1963.

[15] Pang, W., Wang, K., Zhou, C., Dong, L., Liu, M., Zhang, H., Wang, J., Modified Particle Swarm Optimization Based on Space Transformation for Solving Traveling Salesman Problem. *Proceedings of the Third International Conference on Machine Learning and Cybernetics*, Shanghai, **4**: 26-29. 2004.

[16] Pirlot, M., General Local Search Methods, *European Journal of Operational Research*, **92**: 493-511. 1996.

[17] Rowe, A. J., Jackson, J. R., Research Problems in Production Routing and Scheduling, *Journal of Industrial Engineering.*, **7**: 116-121. 1956.

[18] Smith, W. E., Various Optimizers for Single Stage Production, *Naval Research Logistics Quarterly*, **3**: 59-66. 1956.

[19] Song, M., Gu, G., Research on Particle Swarm Optimization: A Review, *Proceedings of the 2004 Iternational Conference on Machine Learning and Cybernetics*, **4**: 26-29. 2004.

[20] Lawrence, S., Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques (Supplement), *Graduate School Ind. Adm*.. Pittsburgh, PA, Carnegie Mellon Univ., 1984.

[21] Tasgetiren M. F., Yun-Chia Liang, Sevkli M., Gencyilmaz G., Particle Swarm Optimization Algorithm for Makespan and Maximum Lateness Minimization in Permutation Flowshop Problem, *4th International Symposium on Intelligent Manufacturing Systems*, Sakarya, Turkey, 431-441. 2004.

[22] Taşgetiren M. F., Sevkli M., Yun-Chia Liang, Gençyılmaz G., Particle Swarm Optimization Algorithm for Single-machine Total Weighted Tardiness Problem, *Congress on Evolutionary Computation*, Portland, OR, **2**: 19-23. 2004.

[23] Wang, K., Huang, L., Zhou, C., Pang, W., Particle Swarm Optimization for Traveling Salesman Problem. *Proceedings of the Second International Conference on Machine Learning and Cybernetics*, **3**: 2-5. 2003.

[24] Weijun, X., Zhiming, W., Wei, Z., and Genke, Y., A New Hybrid Optimization Algorithm for the Job Shop Scheduling Problem. *Proceeding of the 204 American Control Conference*. Boston, MA, **6**: 5552-5557. 2004.

[25] Xia, W., Wu, Z., An Effective Hybrid Optimization Approach for Multi-Objective Flexible Job-Shop Scheduling Problems. *Computers and Industrial Engineering*, **48**: 409-425. 2005.

VITA

Brian Patrick Ivers

Candidate for the Degree of

Master of Science

Thesis: JOB SHOP OPTIMIZATION THROUGH MULTIPLE INDEPENDENT PARTICLE SWARMS

Major Field: Electrical Engineering

Biographical:

Personal Data: Born March 19, 1981 in Oklahoma City, Oklahoma. Son of Phil and Michele Ivers.

Education: Graduated from Notre Dame High School, Cape Girardeau, Missouri in May 1999. Attended 1 year of college at Southeast Missouri State University, 1999 – 2000. Transferred to Oklahoma State University in August 2000. Graduated with Bachelors degree in Electrical Engineering Technology in December 2003. Completed requirements for Master of Science Degree in Electrical Engineering at Oklahoma State University in December 2006.

Experience: Employed as a Work Study student at the Oklahoma State University Theater Department from August 2002 – January 2004. Worked as a Teaching Assistant during the Spring Semester of 2004. Interned at Noranda Aluminum in New Madrid Missouri, Summer 2005. Employed as a Work Study Student at the ITLE Department at Oklahoma State University, August 2005 – May 2006.

Name: Brian Patrick Ivers Date of Degree: December, 2006

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: OPTIMIZATION THROUGH MULTIPLE INDEPENDENT PARTICLE SWARMS

Pages in Study: 110 Candidate for the Degree of Master of Science

Major Field: Electrical Engineering

Scope and Method of Study: This study examines the optimization of the Job Shop Scheduling Problem (JSP) by a search space division scheme and use of the meta-heuristic method of Particle Swarm Optimization (PSO) to solve it. The Job Shop Scheduling Problem (JSP) is a well known huge combinatorial problem from the field of Deterministic Scheduling. It is considered the one of the hardest in the class of NP-Hard problems. The objective is to optimally schedule a finite number of operations to a finite number of resources while complying with ordering constraints. The Particle Swarm Optimization Algorithm (PSO) is a new meta-heuristic optimization method modeled after the behavior of a flock of birds in flight. It initializes "particles' in the search space of a particular problem assigns them a velocity and position. They fly through the search space with out direct control, but are given both a cognitive personal component and a global or social component of the best positions (thereby solutions) in space. The PSO algorithm is considered a very fast algorithm and is emerging as a widely studied widely used algorithm for optimization problems. This study uses this meta-heuristic to solve the JSP by assigning each machine in the problem an independent swarm of particles.

Findings and Conclusions: It turns out that the PSO can be applied to successfully solve many JSP problems. However, the success depends largely upon the type of schedules being constructed. The idea of space division according to machine shows promise indeed, but could benefit from a custom made scheduling algorithm to accompany it.

ADVISER'S APPROVAL: Dr. Gary Yen